5-6-2020

QoS in SDN-Based Large-Scale Networks

Haitham Ghalwash

University of Connecticut - Storrs, haitham.ghalwash@uconn.edu

Follow this and additional works at: https://opencommons.uconn.edu/dissertations

Recommended Citation
Ghalwash, Haitham, “QoS in SDN-Based Large-Scale Networks” (2020). Doctoral Dissertations. 2482.
https://opencommons.uconn.edu/dissertations/2482
Abstract

Recent research has shown that the Software-Defined Networking (SDN) technology is a promising architecture providing abstraction and programmability of modern networks and enables a more efficient solution to many of the security, performance, management, and QoS issues. The SDN is also proven to meet the growing demands of new applications for an application-oriented network. This dissertation researches and enriched SDN with an added level of QoS for network applications. Starting by investigating the potential of an SDN-based large-scale networked system, two topologies widely used in modern data centers, namely, Fat-tree and BCube, are considered. Their behavior and performance under different network scales, traffic loads, and traffic patterns are studied. Experimental results indicate the superiority of a Fat-tree network as it scales up. The potential of SDN in supporting Big-Data applications is subsequently investigated, using a Hadoop cluster with the Fat-tree interconnection. Experimental results in terms of throughput and execution time for the read/write and sorting operations demonstrate the superiority of the SDN controller over the normal forwarding mechanism. In addition, a framework adopting externally developed modules to enrich SDN capabilities for forwarding, metric retrievals, and congestion control is proposed. A QoS level is guaranteed for traffic classification, metric-based route selection, or congestion detection and control. Noticeable features of the proposed framework include (1) facilitating dispatching applications over different paths based on the application type; (2) metric-based monitoring and rerouting, and (3) congestion detection and control to ensure balanced traffic flow. The behavior and the performance of different traffic types, namely, UDP, TCP, VOIP, and a Big-Data application, are investigated. Experimental results via such metrics as delay, jitter, and packet loss substantiate the advantage of having the developed modules on top of the controller for all traffic types.
The proposed framework reduces the overall average delay, jitter, and packet loss by 54%, 32%, and 51%, respectively. Moreover, the average utilization of a monitored port is reduced by 22%.

Keywords: SDN, QoS, Fat-tree, Large-scale networks, Mininet, OpenDaylight, Congestion Control
QoS in SDN-Based Large-Scale Networks

Haitham Atef Ghalwash

B.S., Tanta University, Egypt, 2006
M.S., Tanta University, Egypt, 2013
M.S., University of Connecticut, USA, 2018

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

at the

University of Connecticut

2020
Doctor of Philosophy Dissertation

QoS in SDN-Based Large-Scale Networks

Presented by
Haitham Atef Ghalwash, B.S., M.S.

Major Advisor __________________________________________________
Chun Hsi-Huang

Associate Advisor _________________________________________________
Reda A. Ammar

Associate Advisor _________________________________________________
Sanguthevar Rajasekaran

University of Connecticut
2020
ACKNOWLEDGMENTS

This thesis would not have been possible without the help, support, and patience of my wife, my parents, and my parents in law. I want to thank them for their outstanding support and patience.

I also wish to sincerely thank my Major Advisor, Dr. Chun Hsi-Huang, for his continuous supervision, encouragement, support, and patience. He continuously supported me on both the technical and the personal level. This work would have never been possible without his thoughtful guidance and keen insight. I also would like to thank my advisors, Dr. Reda Ammar and Dr. Sanguthevar Rajasekaran, for their support and guidance. I am also grateful to my beloved wife, who supported me during my Ph.D. journey. Many thanks for her understanding, patience, and prodding.

I am also grateful to all my family, especially my beloved parents, parent-in-law, brothers, and sister. They continually supported me without any hesitation. Without their support, none of this would ever be possible.

Dedication

My Ph.D. thesis is dedicated to my wife, sons, parents, and parents-in.

Haitham Atef Ghalwash
## Contents

List of Figures .............................................................................................................. vii
List of Tables ................................................................................................................ ix
List of Abbreviations .................................................................................................. x

Chapter 1: Introduction ............................................................................................. 1

1.1. Introduction ........................................................................................................ 1
1.2. Benefits of Software-Defined Networking ......................................................... 3
1.3. Traditional Network vs. SDNs ........................................................................... 5
1.4. SDN Architecture ............................................................................................ 6
   1.4.1. SDN Controllers ....................................................................................... 9
   1.4.2. SDN Applications .................................................................................. 10
1.5. Quality of Service (QoS) in SDNs ................................................................... 11
1.6. Network Topologies ....................................................................................... 12
1.7. Research Motivation ..................................................................................... 13
1.8. Research Contribution ................................................................................. 15
1.9. Dissertation Organization ............................................................................ 16

Chapter 2: A Review of Related Work .................................................................... 17

2.1. Datacenter Topologies (Fat-tree vs. BCube) .................................................... 17
2.2. SDN benefits for applications in Data Centers ............................................... 18
2.3. SDN and QoS through application classification and flow rerouting .......... 20
2.4. SDN and QoS through Congestion control ................................................... 21

Chapter 3: Modern technologies and Network Modeling ....................................... 25

3.1. Introduction ..................................................................................................... 25
3.2. Mininet Emulator ........................................................................................ 26
3.3. OpenFlow Protocol ....................................................................................... 26
3.4. OpenDaylight Controller .............................................................................. 28
3.5. Distributed Internet Traffic Generator (D-ITG) ........................................... 30
3.6. Hadoop Big-Data Application ....................................................................... 31
3.7. Docker Containers ......................................................................................... 32

Chapter 4: Investigation of Software-defined large-Scale Networks ...................... 34

4.1. Introduction .................................................................................................... 34
4.2. Investigating SDN-based Network topologies .............................................. 34
   4.2.1. Experimental setup and Configuration .................................................. 37
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Tradition networks vs. Software-Defined Networks</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>SDN Architecture</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Internal vs. External Applications</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Network topologies</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>OpenFlow Switch components</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>OpenFlow Rule fields</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>OpenDaylight Architecture</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Simplified Hadoop cluster</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>VM vs. Container Architecture</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>BCube architecture for K=1</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>BCube-like architecture</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Fat-tree architecture</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Different scales of 3-level Fat-tree</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>Different scales of 3-level BCube-like</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Average Delay of BCube-like and Fat-tree over different scales and traffic</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Min, Max and Avg Delay of BCube-like and Fat-tree over different scales and</td>
<td>41</td>
</tr>
<tr>
<td>4.8</td>
<td>Average Jitter of BCube-like and Fat-tree over a different scale and traffic</td>
<td>42</td>
</tr>
<tr>
<td>4.9</td>
<td>Packet drop of BCube-like and Fat-tree over a different scale and traffic</td>
<td>42</td>
</tr>
<tr>
<td>4.10</td>
<td>Different scales of 3-level Fat-tree</td>
<td>45</td>
</tr>
<tr>
<td>4.11</td>
<td>4-port scale fat-tree topology with a Hadoop cluster</td>
<td>46</td>
</tr>
<tr>
<td>4.12</td>
<td>Execution time Read/write all scale</td>
<td>48</td>
</tr>
<tr>
<td>4.13</td>
<td>Throughput Read/write all scales</td>
<td>49</td>
</tr>
<tr>
<td>4.14</td>
<td>All scales DFSIO average Execution time</td>
<td>49</td>
</tr>
<tr>
<td>4.15</td>
<td>All scales DFSIO average Throughput</td>
<td>50</td>
</tr>
<tr>
<td>4.16</td>
<td>Sort Execution time</td>
<td>51</td>
</tr>
<tr>
<td>4.17</td>
<td>Sort Execution time overall scales</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>QoS framework Architecture</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>Fat-tree 4-port 3 levels topology and controller</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>Delay (UDP, TCP, and VOIP)</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>Packet dropped (UDP, TCP, and VOIP)</td>
<td>66</td>
</tr>
<tr>
<td>6.4</td>
<td>Average Jitter (UDP, TCP, and VOIP)</td>
<td>67</td>
</tr>
<tr>
<td>6.5</td>
<td>Monitoring port#1 on switch#6</td>
<td>67</td>
</tr>
<tr>
<td>6.6</td>
<td>All Switches traffic over time UDP, TCP, and VOIP traffic</td>
<td>68</td>
</tr>
<tr>
<td>6.7</td>
<td>testDFSIO throughput</td>
<td>70</td>
</tr>
<tr>
<td>6.8</td>
<td>testDFSIO Execution time</td>
<td>70</td>
</tr>
<tr>
<td>6.9</td>
<td>Test DFSIO traffic form Monitoring Module</td>
<td>71</td>
</tr>
<tr>
<td>6.10</td>
<td>Fat-tree topology (4-port switch) and a controller</td>
<td>72</td>
</tr>
<tr>
<td>6.11</td>
<td>Min, Max and Average Delay of different application and odl-l2switch, SP, and QoS-SP forwarding modes</td>
<td>75</td>
</tr>
</tbody>
</table>
Figure 6.12: Packet Dropped for different application and odl-l2switch, SP, and QoS-SP forwarding modes ................................................................. 75
Figure 6.13: Average jitter for different application and odl-l2switch, SP, and QoS-SP forwarding mode ................................................................. 76
Figure 6.14: Ports Utilization for odl-l2switch, SP, and QoS-SP forwarding modes ................................................................. 77
Figure 6.15: Active metric experiment topology ................................................................. 79
Figure 6.16: Delays (traffic scenario “1”) ................................................................. 81
Figure 6.17: Packet Drop (traffic scenario “1”) ................................................................. 82
Figure 6.18: Average Jitter (traffic scenario “1”) ................................................................. 82
Figure 9.19: Delay (traffic scenario “2”) ................................................................. 83
Figure 6.20: Packet Drops (traffic scenario “2”) ................................................................. 84
Figure 6.21: Average Jitter (traffic scenario “2”) ................................................................. 84
Figure 6.22: Fat-tree topology (4-port switch) and a controller – congestion control experiment ........ 86
Figure 6.23: Max Delay improved percentage for TCP, UDP, and VOIP ......................................... 88
Figure 6.24: Average Delay improved percentage for TCP, UDP, and VOIP ......................................... 89
Figure 6.25: Average Throughput improved percentage for TCP, UDP, and VOIP ......................................... 89
Figure 6.26: Average jitter improved percentage for TCP, UDP, and VOIP ......................................... 90
Figure 6.27: Overall Improved percentage for TCP, UDP, and VOIP ......................................... 90
Figure 6.28: 4-port switch Utilization of each level in a Fat-tree topology (ODL) ......................................... 91
Figure 6.29: Shortest-path port utilization for an experiment ......................................... 92
Figure 6.30: Congestion Control (80%) port utilization for an experiment ......................................... 93
Figure 6.31: Congestion Control (70%) port utilization for an experiment ......................................... 93
List of Tables

Table 1.1: SDN vs. traditional network................................................................................. 6
Table 1.2: Comparison of Controllers.................................................................................. 10
Table 4.1: Topology Size ........................................................................................................ 37
Table 4.2: Experiment Topology Size ................................................................................... 37
Table 4.3: SDN-Fat-tree experimental topology for application support................................. 45
Table 4.4. Average Improvements......................................................................................... 50
List of Abbreviations

API: Application Programming Interface
ARLD: Automatic Rerouting with Loss Detection
ASIC: Application-Specific Integrated Circuit
BW: Bandwidth
CDPI: Control-to-Data-Plane Interface
APIC: Application Policy Infrastructure Controller
CLI: Command Line Interface
CPU: Central Processing Unit
DITG: Distributed Internet Traffic Generator
DNS: Domain Name Servers
DOM: Document Object Model
DSCSD: A Dynamic Scheduling and Congestion control across data centers
ECMP: Equal-Cost Multi-Path
EoR: End of Rack
GB: Gigabyte
HDFS: Hadoop Distributed File System
HP VAN: HP Virtual Application Networks
HPC: High-Performance Computing
ICMP: Internet Control Message Protocol
IDS: Intrusion Detection System
IoT: Internet of Things
IP: Internet Protocol
IPS: Intrusion Prevention System
IPv4: Internet Protocol version 4
IPv6: Internet Protocol version 6
IT: Information Technology
MAC: Media Access Control Address
MD-SAL: Model-Driven SAL
NAT: Network Address Translation
NBI: Northbound Interface
Netconf: Network Configuration Protocol
NFV: Network Functions Virtualization
ODL: OpenDaylight
OED: Oxford English Dictionary
OF: OpenFlow
ONF: Open Networking Foundation
ONOS: Open Network Operating System
OS: Operating System
OSA: Optical Switching Architecture
OSGi: Open Service Gateway Initiative
OVS: Open vSwitch
OVSDDB: Open vSwitch Database
QA: Quality Assurance
QoE: Quality of Experience
QoS: Quality-of-Service
QoS-SP: Shortest Path route with Quality-of-Service
RAM: Random Access Memory
REST: Representational State Transfer
RPC: Remote Procedure Call
RTT: Round-Trip Time
SAL: Service Abstraction Layer
SB: Southbound Interface
SDCC: Software-Defined Congestion Control
SDN: Software-Defined Network
SSL: Secure Sockets Layer
TLS: Transport Layer Security
Src: Source
TASDN: Time-Aware SDN
TCP: Transmission Control Protocol
Telnet: Terminal Network/Telecommunications Network
ToR: Top of Rack
UDP: User Datagram Protocol
VLAN: Virtual Local Area Network
VLB: Valiant Load Balancing
VM: Virtual Machine
VoIP: Voice over Internet Protocol
VSC: Virtualized Services Controller
YANG: Yet Another Next Generation
ZB: Zettabytes
Chapter 1

Introduction

1.1. Introduction

Nowadays, there is an urgent need to increase the efficiency in communicating networks due to the growing user’s applications’ needs. Networks today are rapidly growing in terms of scale and traffic volume. A forecast from Cisco white paper [1] stated that, by the year 2021, the number of devices connected to IP networks would be three times as high as the global population. The annual global IP traffic will reach 3.3 zettabytes (ZB), i.e., 275 Exabytes per month. In 2016, global IP traffic was 1.2 ZB per year. Moreover, the rise of cloud services, visualization, wireless networks, IoTs, and big-data applications are hardening the need for software-oriented network architecture.

Data centers are now storing and hosting large-scale service applications and data. The emerging applications are shifting away from traditional client-server traffic models to newly invented models that involve more machine-to-machine data transfer. Moreover, freshly created technologies, such as server virtualization and cloud computing, are changing traffic behavior in data centers. According to what was reported, around 70% of the traffic flows internally in the data center, while 30% is external to it. Subsequently, the significant problems arising in data centers are the intra-data center traffic bottlenecks [2], the inefficient scalability in terms of cost and performance, the increased network complexity, and the difficult and error-prone ways to
fulfill security and quality-of-service (QoS) policies across hundreds or thousands of network devices. Many Protocols were considered for solving such problems. However, the invented protocols tend to resolve the issues separately. According to the ONF paper [3], each protocol is “solving a specific problem and without the benefit of any fundamental abstractions.” Also, new network architectures and technologies are considered. New network topologies, such as the Fat-tree [4], VL2 [5], DCell [6] and BCube [7], implementing novel traffic engineering mechanisms such as the Hedera [8], MicroTE [9] and D3 [10] are currently being considered to accommodate the increasing demand of high traffic.

Recently, the integration between the applications and network configurations has gained a great interest in both industry and academia. The emerging technologies are gradually forcing the traditional hardware-centric data network to software-based network functionality. Software-Defined Network (SDN) is a newly proposed network paradigm that brings a lot of new capabilities and solves many problems in today’s networks. It brings all benefits by merely decoupling the control plane away from the data plane. It provides directly programmable, incredibly dynamic, easily used, and adaptable networks. An SDN logically centralized control-plane offers full control, management, and status information of the whole network. It proved its efficiency in enhancing applications’ performance, including Big-data applications. With SDNs, the network functionality scaling can be improved and deployed more flexibly, quickly, and efficiently. The traffic engineering is easier, efficient, and achieves better resource utilization. Furthermore, network reconfiguration and security are easily applied and maintained.
1.2. **Benefits of Software-Defined Networking**

Nowadays, the growing application’s traffic demands are suffering from lack of information, security, management, and QoS issues. Network Functionalities, in traditional networks, are implemented separately in each network device such as switches, routers, firewalls, and other dedicated hardware. This paradigm raised many problems, such as slower deployments, hard administration, low agility, error-prone, and low efficiency. Moreover, the widespread of new evolving technologies such as virtualization, cloud computing, Big data, and IoT are revealing more challenges for traditional networks. These Limitations are mandating the need for a new architecture. SDN provides simple, centralized, responsive, agile, programmable, fast deployed, easy upgraded, automated configuration, and reduces networks’ cost. The following illustrates some benefits provided through SDNs:

- **Simplification and centralized network management**: SDN provides a centralized view of the entire network. It enables the full control of every entity in a network through a centralized management tool. It can smoothly accelerate deployment and scale network functionality and service delivery.

- **Automation, responsiveness, agility, and programmability**: SDN provides several mechanisms for defining traffic rules. The SDN controller pushes out rules and policies as demanded to shape traffic according to the network application’s needs. It enables applications to request services from the network and dynamically responses accordingly. It also provides quickly accessible API references for vendors to develop applications to control network traffic. Moreover, SDN helps to keep up with the frequently changing demands and availability of the network. It supports the dynamic movement, replication, and allocation of virtual resources. Lastly, SDN reduces the time and complexity of
provisioning services and resources. It provides discriminated services and isolation for various traffic.

- **Better and more granular security:** SDN networks are much simpler to be secured. Policies, rules, and isolations are centralized and implemented across all network entities through the SDN controller. A chance of a misconfigured device is reduced and thus dropping security breaches. SDNs allow security enforcement to the application, endpoint, and device levels. However, the centralized SDN controller, being one entity, raises the security concern against possible attacks.

- **Cost reduction:** Many network services, features, and operations can be combined as software’s running on a fewer number of network entities. Data centers can be less dependent on proprietary hardware and dedicated appliances. A less expensive device, that provides an open interface, e.g., OpenFlow, to the SDN controller can be used. Moreover, SDN controllers provide more efficient administration and better utilization of server and network elements, which reduces the operational cost.

- **Support for cloud abstraction:** Cloud computing is rapidly growing and evolving into a unified infrastructure. SDN helps unify cloud resources. The massive networks’ entities in data centers can be controlled and managed through a centralized controller.

- **Support for network functions virtualization (NFV):** NFV is a new network technology that implements network functions such as routing, firewalls, IPS, and IDS as virtual services instead of having a dedicated hardware appliance. SDN can assist in managing and controlling the NFV entities to get the best overall benefit. In other words, NFV can provide essential networking functions that take advantage of SDN in controlling and orchestrating them for specific uses.
1.3. **Traditional Network vs. SDNs**

Network traffic is rapidly growing and connecting more applications with higher demands. The enormous number of added devices require a decent effort to provide connectivity, manageability, security, and QoS. The traditional networking shows many limitations and is unable to fulfill the growing demands for high performance, agile, and customized networks. Although networking protocols are evolving to solve specific problems, it failed to provide a full abstracted view of the network. However, the way traditional networks are set up, configured, and managed is quite challenging, time-consuming, error-prone, and requires a higher level of expertise for Multi-vendor boxes.

A device, in traditional Networks, encompasses a control, management, and data plane. The control plane is responsible for route decisions by determining how/where/when to send packets. The management plane facilitates the configuration and management of different alarms and backups. The data plane, sometimes called a forwarding plane, is in charge of executing the control plane decision for packet forwarding. In a traditional network, control, management, and data plans are linked together, with a level of intelligence, in each device, and supported by the adjacent devices. Nowadays, SDNs abstract the network control and management away from the hardware forwarding plane/device, see figure 1.1. A Logically centralized control, separated from the Hardware-specific forwarding element, takes care of configuring, managing, and controlling the underlayer forwarding elements as well as providing an abstraction view to the upper applications running over the network. The new architecture allows easier IP address allocation, Policy enforcement, Routing changes, Bandwidth allocation, and end-to-end reachability. Table 1.1 shows a summarized comparison between traditional networks and the SDNs.
(a) Traditional Networks  
(b) Software-Defined Networks

Figure 1.1: Tradition networks vs. Software-Defined Networks

<table>
<thead>
<tr>
<th>Software-defined network</th>
<th>Traditional Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoupled control (software) from the data plane (hardware)</td>
<td>Data and Control planes tied together in a single device</td>
</tr>
<tr>
<td>Intelligence (control plane) logically centralized</td>
<td>Intelligence (control plane) distributed</td>
</tr>
<tr>
<td>Control plane can run on servers to implement network functionalities</td>
<td>Network functionality implemented in a dedicated appliance (ASIC)</td>
</tr>
<tr>
<td>Configured using centralized open software, with Global network view abstraction</td>
<td>Configured separately per device using Command Line Interface (CLI)</td>
</tr>
<tr>
<td>Programmable during deployment time or later stage based on the requirements.</td>
<td>Static and inflexible networks.</td>
</tr>
<tr>
<td>Easy update, automated, custom and much less error-prone</td>
<td>The configuration is manual, device by device, slow, and prone to errors.</td>
</tr>
<tr>
<td>Responding quickly to feature requests</td>
<td>They possess little agility and flexibility</td>
</tr>
<tr>
<td>Easy management with universal standards through the controller</td>
<td>Multi-vendor boxes require a high level of expertise</td>
</tr>
</tbody>
</table>

1.4. **SDN Architecture**

The SDN architecture is a new layering model that decouples the network control away from the forwarding functions, thus enabling the network to become directly programmable and abstracting the underlying infrastructure for applications and network services. According to
Open Networking Foundation (ONF) [11] definition, “Software-Defined Networking (SDN) is the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices.” ONF is a non-profit consortium serving several industry carrier related projects. Among these projects is OpenFlow, which is the first standard communication interface between the SDN controller and the forwarding layer. The SDN architecture and arrangement of control & management delivered centralized network intelligence features such as Directly Programmable, Agile, and Vendor-Neutral Standards. These features enable solving many of the security, performance, management, and QoS issues. According to the ONF white paper [3] in 2012, SDN architecture presented in figure 1.2, compromises of three distinct planes, namely, Application Plane, Control Plane, and Data Plane.

- **Application Plane**: is the top layer, that comprises all applications, implementing the desired network behavior. It is the layer where all the features, services, and policies are defined. Applications can be developed either internally in the controller or externally using an open interface for communication (e.g., REST). It merely generates decisions for traffic
forwarding, network behavior, and security enforcements.

- **Control Plane**: is logically centralized and may have one or more controllers. It is responsible for mapping all desired network actions from the top layer (Application Plane) to the lower layer (Data Plane), via open interfaces. It is also responsible for keeping track of the topology and events in the Data-Plane, as well as providing abstracted information about the network elements, including statistics and events, to the requesting applications. A Controller offers one or more Northbound Interface (NBI) Agents to communicate with the applications in the upper layer. It also provides Southbound Interfaces, sometimes called Control-to-Data-Plane Interface (CDPI) driver, that supports the communication with data path elements in the lower layer.

- **Data Plane**: is the infrastructure layer or sometimes called the SDN Data path. It comprises all types of physical/virtual network elements and devices, which exposes visibility and uncontended control over its data processing capabilities. The controller interacts with all network devices through the southbound interface using dedicated control messages and APIs, e.g., OpenFlow [12] protocol.

As the network scales up, a single controller may not be able to manage the entire network [13]. Using a single controller may result in a single point of failure and performance bottleneck problems. This motivated the innovation of new scaling techniques through partitioning, aggregating, clustering, or even replicating controllers. Several approaches were proposed that mostly fell in one of two categories, hierarchical and fully distributed structure. In the hierarchical technique, e.g., Kandoo [14], controllers operate on a portion of the network, with a logically centralized root controller for the network-wide knowledge. In the distributed approaches, e.g.,
ONIX [13] and HyperFlow [15], controllers operate on their local view and may exchange messages to enhance their knowledge.

1.4.1. SDN Controllers

A controller is the core/brain of an SDN network. It is responsible for manipulating the network elements in the infrastructure layer to fulfill the running application’s needs. Over the past few years, many commercial and open-source controllers were developed with different features and capabilities. Examples of some industrial controllers are, (1) Cisco Application Policy Infrastructure Controller (APIC), (2) HP Virtual Application Networks (VAN), (3) NEC Programmable Flow, (4) Nuage Networks Virtualized Services Controller (VSC), and (5) VMware NSX Controller. On the other hand, for open-source Controllers, (1) NOX [16] is the simplest and most popular controller with standard APIs for managing network resources. (2) POX [17] has emerged as a successor of NOX in Python. (3) Ryu is Another available controller under Apache 2.0 license. (4) Floodlight, a recent controller, an Apache-licensed OpenFlow controller that is widely used in research. (5) OpenDaylight (ODL), and (6) The Open Network Operating System (ONOS) are the most recent. The ODL is a Linux Foundation collaborative project, written in Java and is highly supported by various companies. ONOS, on the other hand, is a “private” development effort (managed by ON.Lab). It is mainly designed for service providers for wide area networks (WAN). Table 1.2 summarizes the strengths and weaknesses of the open-source controllers mentioned above [18][19]. ODL is adopted in this research for being more generic and serving a wide range of applications. A study in [20] also revealed the superiority of ODL over ONOS in terms of scalability.
### 1.4.2. SDN Applications

SDN applications can be designed to be either reactive or proactive. Reactive applications are dynamic that set transient rules on devices that may frequently change, based on the forwarded packets to the controller. In Proactive applications, forwarding rules are set based on initial configuration and external change events. SDN application code is developed either internally or externally to the SDN controller. Internally developed applications run inside the container hosting the SDN controller. It should be written in the controller’s native language considering the internal controller structure and design. An internally developed application requires a complete understanding of the controller’s interior layout, execution constraints, dependency, and available classes and interfaces. It is usually faster than being externally developed. On the
contrary, externally developed applications can be written in any language and may run on a separate host outside the container hosting the controller. External SDN usually uses REST API provided by the controller to access the controller services. An Internal vs. external application programming structure is illustrated in figure 1.3.

![Diagram showing Internal vs. External Applications](image)

**Figure 1.3: Internal vs. External Applications**

### 1.5. Quality of Service (QoS) in SDNs

One of the significant issues in today’s networks is the Quality of Service (QoS). Network QoS is simply the level of service agreed upon, to be delivered to network applications/users. It is measured through bandwidth, end-to-end delay, jitter, and packet loss. It reflects the capability of a network to provide better service to selected network traffic. Active or passive measurement methods are usually applied. Active measurements often inject additional packets in the network, e.g., ping packets, and metrics such as the latency, loss, and jitter are recorded. On the other hand, a passive measurement does not impose any overhead on the network.

SDN architecture is the key to solve current traditional network QoS delivery limitations. In traditional networks, QoS is usually updated after the degradation in the Quality of Experience (QoE). For SDN, the centralized controller and the available network status helps optimize the administrative decision towards enhancing network efficiency. Moreover, it helps network
administrators to smoothly and efficiently create automated QoS management frameworks by considering resource reservation, queue management, and traffic scheduling. Furthermore, monitoring is easily provided through SDN controllers. Detecting threats and performance issues become a handful in nearly real-time, with the possibility of predicting the future behavior that profoundly influences the network operation.

### 1.6. Network Topologies

A data infrastructure always considers a network topology for interconnecting servers and provides agility and reconfigurability. In data centers, networks primarily interconnect top of rack (ToR) switches through aggregation switches, end of rack (EoR), which are, in turn, connected via core switches. Topologies, in data centers, are either of fixed or flexible architectures [21]. In fixed, contrary to flexible architectures, a network topology can’t be modified after deployment. Fixed architectures are of two categories: the tree-based architectures (such as Fat-tree [4] and VL2 [5]) and recursive topologies (such as DCell [6] and BCube [7]). On the other hand, flexible architectures such as cThrough [22], Helios [23], and OSA [24] enable reconfigurability of their network topologies. See figure 1.4

![Figure 1.4: network topologies](image-url)
Based on the literature review, considering fixed networks’ architectures, Fat-tree as a tree-based architecture and BCube as a recursive topology proved to be good performing topologies compared to others. Hence, in this research, an investigation is carried out for both topologies under an SDN environment [25], using different network scales, traffic loads, and traffic patterns. The topology with a better performance in terms of delay, jitter, and packet drops is adopted in the present research for further investigations. Fat-tree and BCube topologies are further detailed in section 4.2.

1.7. Research Motivation

Nowadays, networks become a vital part of the critical activities of our daily life, e.g., electricity networks, communication, social media, IoTs, etc. The growing demands and novel challenges for the evolving technologies and applications are becoming more defiance, tricky, and more stringent. Also, such growing needs are hardening the task of management and configuration in traditional networks. Deploying new protocols, services, resources’ optimization, and traffic differentiation using the conventional network is slow, complex, and error-prone. Data centers dictate the need for a higher level of intelligence and performance control. The performance is profoundly affected by the topologies, which are playing an essential role in providing multiple ways for hosts to reach one another. The forwarding mechanism is another significant component affecting the overall performance. Moreover, slow and error-prone network management and configuration profoundly affect traffic performance.

Hence, this urged the need for SDN as a new promising solution that is capable of solving the mentioned traditional network problems through an abstracted centralized controller. Therefore,
it is valuable to investigate the performance and present the value of using SDN for different scales of data centers. Moreover, various topologies could be a critical factor in how SDN can support data centers. The growing application’s traffic demands are suffering some performance and scalability issues. This mandate re-considering the network mounted software and provided functionalities in such networks.

To mark SDN as a successful candidate for industrial networks, QoS and security issues need to be well-structured and better solved. Many pieces of research investigated the application’s security, Quality of Service (QoS), and monitoring. For traffic orchestration, QoS is one of the main issues in nowadays networks. QoS deals with providing end-to-end guarantees for applications and users. SDNs permit seamless traffic flow optimization and management. Traditionally, even before the advent of SDN, operators used a data-plane load-balancing technique, e.g., equal-cost multi-path routing (ECMP) and Valiant Load Balancing (VLB), which dispatches traffic by assigning several paths for each flow. However, the proposed techniques did not react well in some situations, such as link failure and long-running flows on the same link, which give rise to underutilized or congested networks.

SDN is a hot topic that raises many research aspects to be investigated. For example, there has been much research in the areas of (1) Service Function Chaining, (2) Security, (3) Network Slicing, (4) Internet of Things (IoT), (5) Scalability and Hierarchical Controllers, (6) Quality of Service, (7) Energy efficiency, (8) Load Balancing, (9) Wireless and vehicular Ad-hoc SDNs.
1.8. **Research Contribution**

This research aims to introduce a new paradigm offered through Software Defined Networking to provide QoS for different applications. As mentioned above, network topology plays a vital role in its performance. So, the presented research starts by investigating the relative performance of two widely used topologies, namely, fat-tree and BCube, bordered by the SDN environment. Both topologies are compared for different network scales, traffic loads, and traffic patterns. Experimental results project the performance trend when the two topologies continue to scale up.

Next, the support of SDN for big-data applications traffic, running in large scale networks, is investigated. A Hadoop [26] multi-node cluster is adopted, as a big-data application connected through a fat-tree topology and running either in SDN or normal switching mode. The two operating modes are compared under different network scales and traffic patterns. Experimental results are executed to record the performance of both SDN and normal switching operating modes for reading, writing, and sorting operations as the network scale-up.

Finally, the research presents a QoS orchestration framework for enhancing the performance of a standalone SDN controller. The main features of the proposed framework focus on the ability to provide a dynamic level of QoS through either application classification or route selection based on active/passive measured metrics. Moreover, the proposed framework monitors and analyses the status of the running network to detect congested parts. The framework automatically controls the identified issues and optimizes the load distribution over different parts of the network. The framework relies on the features provided through the SDN network to monitor the status and program the network hitting for better performance.
1.9. Dissertation Organization

The dissertation is organized as follows: In Chapter 2, an overview of the related literature work in SDN big-data, congestion control, traffic rerouting, and QoS are presented. Chapter 3 provides a descriptive background of the modern tools, technologies, and protocols used for the experimental part. Chapter 4 gives a detailed investigation along with the supporting experimental results for the performance of SDN-based networks for different topologies, applications, and traffic loads. Also, a study is presented for SDN when supporting big-data applications over different network scales. In chapter 5, the proposed framework building blocks and functionality is presented. Chapter 6 contains the proposed QoS framework supporting experiments and results. Finally, chapter 7 has the conclusion and future work.
Chapter 2

A Review of Related Work

2.1. Datacenter Topologies (Fat-tree vs. BCube)

The emerging data center network designs are seeking better performance by adopting network topologies with higher bandwidth, larger bisection capacities, and more data-paths. Hierarchical multi-rooted network topology, such as the Fat-tree [4], VL2 [5], DCell [6], and BCube [7], are commonly used in modern Data Centers. Fat-tree topology is widely used for its simplicity and has been well-thought-out in many new HPC data centers [27]. It is typically adopted as two to three levels [28] of switches and performs better than DCell and three-tier networks [29]. BCube is also a promising topology that provides a higher network capacity compared to Dcell. It is a server-centric topology that is mostly used in modular data centers and can be easily extended and built using commodity switches [7]. In general, related literature research categorized both Fat-tree and BCube as high throughput topologies, with BCube being slightly better [29], [30]. Both topologies provide nearly the same bisection bandwidth [31]. Fat-tree has a higher average number of paths over BCube [30] and proved to operate better in the case of server failures [32], while BCube performed well in case of simultaneous failure of both servers and switches [7]. In terms of scalability, Fat-tree is well suited for building large-scale high-performance data centers, providing scalable bandwidth at a moderate cost [30]. BCube scales better than Fat-tree for the same number of ports per switch. However, BCube uses a more significant amount of switches
and wires, which may lead to a higher cost [31]. A part of this research is to nominate a target network topology to be the seed for further SDN investigations when supporting applications’ performance.

2.2. SDN benefits for applications in Data Centers

The Software Defined Networks (SDNs) have recently emerged as a promising technology for future datacenters. An SDN controller is the brain of the network that has a global view of the whole network. Using SDN proved its efficiency over traditional networks over different topologies. A study in [33] considering torus and hypercube topologies demonstrated a 45% gain in throughput by SDN over traditional networks when tested for around 256 servers. Also, the SDN environment reduced the packet loss rate, jitter, and latency.

SDN's centralized vision can directly assist network applications by writing network programs in a way to fulfill applications’ specific needs. A controller can detect the changes in network status and dynamically optimizes resources to enhance network performance. Big-data applications are some of the many applications that can instantly make use of the global view provided by the SDN controller. Supporting Big-Data applications with SDNs has the potential of improving the performance for both. Incorporating Hadoop Map/Reduce jobs in SDN networks have raised significant attention in the research community and the literature, supporting the network in the shuffle and reduce phases. A work in [34] proposed an interface between a centralized coordinator of big-data applications and an SDN control plane. The interface passed the traffic characteristics to the SDN controller before the shuffle phase starts. In [35], an SDN-aware version of a Hadoop application was set up to use prioritized queues in the underlying OpenFlow switches. As a result, critical Hadoop traffic was routed first before other less critical traffic, and the Hadoop jobs were
completed faster. A similar study in [36] proposed crosspoint queued (CICQ) switches to schedule packets for several Big-Data applications. The switches schedule packets based on the bandwidth provisioning table that is set by the controller, for different big-data applications, [37] proposed to run monitoring agents at the data center hosts. These monitoring agents periodically monitored the log files produced by big-data applications, e.g., Hadoop, and tried to predict the traffic pattern before it started. A benchmark of Hadoop sorting a 10 GB job recorded about a 35% reduction in job completion time while re-routing only 6% of the flows. Another similar monitoring was HadoopWatch [38] in the form of passive agents attached to the Hadoop cluster to monitor the meta-data and logs of Hadoop jobs. The collected data was used to forecast application traffic before its start. HadoopWatch proved to predict traffic with almost 100% accuracy for small-scale testbeds. As SDN shows benefits to big-data applications, big-data can assist traffic engineering in SDN as well. Typical objectives of traffic engineering include balancing network load and maximizing network utilization. In [39], a dynamic traffic engineering system with SDN architecture and big-data was described. The SDN controller aggregated and summarized information regarding the traffic in big-data applications. This information was analyzed by a specialized big-data application and passed guidance back to the controller, which shaped the traffic policies and reconfigured the switching decisions. The controller could accordingly set up the flows to avoid congestion and enable the application to make a more informative scheduling decision. Other work presented in [40] proposed a time-aware SDN architecture (TASDN) based on the requests arriving at the data center. TASDN can coordinate Big-Data applications by introducing a time-aware service scheduling strategy. Also, TASDN scheduled the data center services with different delay requirements that optimized the big-data application resources.
2.3. SDN and QoS through application classification and flow rerouting

Recently, QoS through SDN is gaining the interest of researchers. QoS is encountered in SDN networks via flow routing, resource reservation, QoE-aware mechanisms, monitoring mechanisms, queue management and scheduling, and some other QoS-oriented mechanisms [41].

A work in [42], developed an application identification technique based on the SDN controller to determine the QoS levels for different types of applications. The application flows were queued with different priorities based on the application type. Experimental results showed a 28% reduction in the average delay. Another work in [43], developed a framework to guarantee QoS for a specific flow. Traffic was classified as it gets into the edge switch and automatically rerouted, based on a queuing technique that satisfies the required service level, when the network was congested. In [44], HiQoS was proposed using multiple paths between source and destination. It also presented queuing mechanisms to guarantee QoS for various types of traffic. In HiQoS, A modified Dijkstra algorithm [45] assisted in assigning multiple paths that satisfy certain QoS constraints. Experimental results showed a reduction in packet delay and an increase in throughput. In [46], an SDN orchestrator was discussed that periodically evaluated the availability status of the network and, accordingly, adapted service chain paths to recover from congestion and to preserve QoS performance. The SDN orchestrator monitored the state of the network and assigned service chain paths based on predetermined QoS requirements. In [47], SDN was used to control the end-to-end IoT traffic. The proposed solution enabled identifying the routing path that possessed a minimum delay in which the critical data traffic was routed. For monitoring the latency, a probe packet was sent over each path, and its trip time was calculated. Similar work in [48] introduced how collected traffic be used to develop statistics to pinpoint the available bandwidth on each link in the network.
In another work [49], Policy-Cop was proposed to provide aggregated, and per-flow QoS guarantees. Policy-Cop enabled implementing and monitoring a high-level network-wide described policy. It periodically collects network traffic data to detect policy violations and forwarded the detected events for manual action or to the policy autonomic adaptation module. Routes were calculated based on control rules in an internal rule DB, network topology, and a collection for network statistics. However, Policy Cop was not designed to provide deterministic QoS. Another work in [50] was introduced to allocate the requested bandwidth for specific flows; the proposed framework configured three default queues on each port with different priorities for the control traffic, high-priority traffic, and the best-effort traffic. It used the Type of Service (ToS) field in the packet header to differentiate the high-priority traffic from the best-effort traffic. Routes were determined using standard routing protocols such as OSPF and BGP. The proposed framework managed the minimum and maximum bandwidth allocations for different QoS levels, with no delay considerations. Another work in [51] offered an SDN-based QoS control framework by introducing multiple queues, with different QoS levels, for each link in the network topology. The QoS queues are proactively set using the link resources (such as transmission rate) and QoS metrics (such as delay budgets). A standard reactive QoS routing algorithm, such as delay-constrained least-cost (DCLC) routing [52] is used to find routes that meet the QoS requirements.

2.4. SDN and QoS through Congestion control

A lot of research was carried out to enhance the QoS via queuing, prioritization, flow management, or congestion control. Network congestion problems are usually solved either by using an end-to-end approach or a network side approach. In the end-to-end approach, the proposed mechanisms focused on the end terminals to relieve congestion. Modifying TCP parameters in the end-to-end applications proved its efficiency in [53]–[55]. While in-network
side approach, the congestion control can be either avoided by providing larger switch buffers or using flow scheduling/re-routing [56]. In this research, a congestion mechanism was proposed that follows the latter category. Providing congestion control through scheduling/rerouting is supported through datacenter topologies that provide multiple redundant paths between any pair of communicating nodes. The flow rerouting mechanism and the network statistics are available through the SDN controllers, which facilitates efficient re-routing between the source and the destination.

In [57], a method for detecting the congested links via monitoring all ports statistics was proposed. Flows in any congested link were rerouted into new paths with free resources. The congestion detection module issued queries, consolidated, and stored the statistics from all OpenFlow switches, which was used to compute the load in various links. Once the transmitted packets were higher than 70% of the link capacity, the controller module computed all short paths $P$, and the total cost of each path was calculated from a total variable cost of links $l_k \in p_i$, each $l_k$ has a predetermined fixed weight, a link load, and like capacity. The full path cost was the addition of fixed and variable costs. Experiments used IPref traffic in a Mininet topology with a floodlight controller.

Another work in [58], proposed a flow congestion avoidance algorithm through the SDN controller. In the proposed scheme, the link utilization flows are the metrics for rerouting without considering the congested switch. All the routes passing through the congested switch are passed into new routes bypassing that switch. Different thresholds were tested, and the optimization point of the Congestion Avoidance Algorithm was 70%. A video stream was tested using D-ITG in a Mininet environment using an ONOS controller. The results showed an 11% enhancement in the throughput and a reduced RTT. In [59], a congestion avoidance mechanism for specific
predefined hosts was proposed. The algorithm calculated the shortest path assisted by the topology information and redirected the traffic based on the obtained traffic details. The results showed an increase in throughput and available bandwidth. Another work In [60], where a model for congestion detection to reduce the packet transmitted between the controller and the switch was proposed. The model was tested and analyzed in Mininet and Floodlight. The aim was to reduce the packet loss and to provide suitable bandwidth for the link between the controller and switch to other applications. In [61], A Dynamic Scheduling and Congestion control mechanism across data centers with a multilevel feedback queuing mechanism was proposed (DSCSD). The traffic was labeled as low or high priority flow as it arrives at the network. If the shortest path is occupied with the high traffic flows, a second shortest path is assigned for the lower priority traffic. Otherwise, the flow should go into the congestion control with multilevel feedback queues, given all routes have no available bandwidth to be used.

A work in [62], proposed an Automatic Re-routing with Loss Detection (ARLD). The idea was that link congestion is the most common cause of packet drops. Once a packet drop was detected, the SDN controller treated that link as congested and invoked a module for other route computing. Another work [63] proposed a proactive flow monitoring method to reduce congestion and packet loss in SDN. The scheme pre-computed additional resources for the path and automatically re-routed congested link’s traffic through the alternate route. The port monitoring component periodically collected port statistics on each switch. The re-routing module was responsible for updating virtual network topology and alternative route computation. The re-routing module removed the bottleneck links from the topology and recalculated routes based on the new network topology. Simulation in Mininet using OpenIris controller and D-ITG packet generator showed a reduction in the average packet loss leading to an overall increase in network efficiency.
In [64], a software-defined congestion control (SDCC) algorithm for network IP was proposed. The algorithm gained global information and regularly monitored the network for congestion, then found out whether there were congested links based on the TCP and UDP flows and characteristics. Upon discovering a congested connection, it is removed from the topology, and the shortest path is calculated for one or more appropriate data flows to reroute. Bearing in mind that the UDP data flows always emphasize short transfer time and high-efficiency requirements, the algorithm selected the TCP data flows for the rerouting.
Chapter 3

Modern technologies and Network Modeling

3.1. Introduction

Modeling and simulation are recent technologies that are applied in different science, engineering, and other application fields. Computer-assisted simulation can model hypothetical and real-life objects and activities to help study and analyze system functions and behavior. Network simulators are typically programs that run on a single computer, get an abstract description of network traffic, and yield performance statistics. Researchers, developers, engineers, and quality assurance (QA) personnel can use them to design various kinds of networks, simulate and then analyze the effect of multiple parameters on the network performance. Network Emulator, on the other hand, differs from a simulation in that an emulated network appears as a physical network, allowing end-systems to be attached to the emulator and will behave as if they are connected to a real network.

For experiments, an open-source controller and a container-based emulation “Mininet” [65] are used to simulate the SDN environment. The OpenDaylight “ODL” controller is a collaborative open-source project hosted by Linux Foundation. Mininet 2.2, supported by OpenFlow protocol, emulates the network elements, such as the host, switches, and links to form any arbitrary network. Distributed Internet Traffic Generator (D-ITG) [66] and a real Hadoop cluster benchmark are used to generate traffic patterns. The scenarios of Hadoop clusters may be set using virtual machines hence require a considerable number of CPUs and
RAMs. Instead, the presented design relies on lightweight software Linux containers, named *Dockers*. Therefore, Hadoop nodes are implemented on separated Docker containers and connected to the Mininet edge switches [67]. The network hosts attached are either Mininet hosts or Docker containers of Hadoop nodes.

### 3.2. Mininet Emulator

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. It is an open-source project for rapidly prototyping any arbitrary network on single or multiple machines. It can run any arbitrary network built from a collection of virtual hosts, switches, routers, and links. It allows the formation and configuration of any network through a set of Python scripts and commands. Also, it enables customizing topologies, link parameters, e.g., link speed, and packet delay & loss. Mininet's virtual hosts, switches, links, and controllers appear as real-time elements that can be used to test any arbitrary network. A distinguishing feature for Mininet is its use in Software-defined networks. It provides an internal POX controller for the created network with the option to use an external controller.

### 3.3. OpenFlow Protocol

OpenFlow protocol is the first defined standard for communications between the control and the infrastructure layers in SDN. OpenFlow allows direct access, for the SDN controller, to manipulate and configure any physical or virtual forwarding element. An OpenFlow-enabled switch uses the OpenFlow protocol to communicate with an external controller. The switch consists of an OpenFlow secure SSL or TLS channel for communication, a group table, and several flow tables, see figure 3.1. Lookups and forwarding are based on configured flow entries in the switch’s flow tables and group table.
Chapter 3: Network Modeling and Technical tools

The controller relies on the OpenFlow protocol to collect information about the connection, topology, nodes, and traffic statistics. An OpenFlow-enabled switch communicates with the controller via OpenFlow messages (such as packet-received, packet-sent out), to modify-forwarding-table or get-stats. On the arrival of a packet at the OpenFlow-enabled switch, the switch looks up the frame in one or more flow-tables. These flow-tables can implement services such as firewalls, NAT, and QoS or collect statistics for network management. Each entry in the flow-tables consists of a rule to match, an action to apply, statistics to obtain, and some timers, see figure 3.2. The rules match an incoming frame based on the incoming port, Source/destination IP address, Source/destination Port number, Source/destination MAC, VLAN tag, type of packet, or other extensions. When a rule matches a frame, an action is applied to that frame, such as
forwarding to one or more ports, forward to the controller, drop, join a queue, or modify frame fields. Each time a flow rule catches a match, the switch updates the statistics/counters associated with that flow entry. There are counters per tables, flows, ports, and queues. Finally, timers are updated to reflect the last flow activity. A received packet with no matching entry is forwarded to the controller. The controller decides either to drop the packet or to add a new entry at the corresponding switch to handle similar packets in the future.

3.4. **OpenDaylight Controller**

The OpenDaylight (ODL) is an open-source project, hosted by Linux Foundation, to provide an open, modular, and flexible SDN platform. The project aims to enhance SDN through community-lead and industry support. The ODL controller is written in Java and uses the YANG modeling language. It is based on Apache Maven, to facilitate and manage building, reporting, and documenting a project. ODL architecture, presented in figure 3.3, consists of three layers. The top layer has ODL applications, which can be written either externally to the controller, using any language, or internally, using Java, inside the controller’s OSGi using the available abstractions, classes, and interfaces.

![Figure 3.3: OpenDaylight Architecture](image)

Chapter 3: Network Modeling and Technical tools
At the middle layer, OpenDaylight core Controller exposes open northbound and southbound interfaces to communicate with the upper applications layer and the lower infrastructure layer (devices). The northbound interface provides Java APIs for any internal application (consumers and producers), and REST APIs for external applications (consumer). The northbound APIs enable the applications to use the controller in collecting data and creating new rules to reflect the desired applications' behavior. The southbound interface implements protocols to manage and control the underlying networking infrastructure, e.g., OpenFlow, Netconf, OVSDDB, etc.

The Service Abstraction Layer (SAL) is the heart of the ODL controller. SAL enables matching the lower layer protocol plugins with the network service function modules. Network services function modules provide a set of components to manage topology, statistics, forwarding rules, and other features. ODL is now following the Model-Driven SAL (MD-SAL). The MD-SAL maps the service request from network service function modules to the appropriate southbound plugin to interact with a particular network device. In MD-SAL, all device status and information are represented and stored as a document object model (DOM), whose interaction is processed and managed by the MD-SAL. Data is incubated in two main “data trees,” namely, the “operational” and the “configurational” data trees. The operational data tree temporary stores, frequently changing, runtime status of the network components. As a result, an abstraction view becomes available for applications to oversee status, changes, and updates of network elements on a real-time basis. On the other hand, the configuration data tree stores the intended status/configuration of the network, which is usually populated through the applications.

In MD-SAL based applications, YANG models are used to define all necessary data and APIs. The OpenDaylight YANG tools generate Java-based APIs for each YANG defined model. The adopted YANG modeling schema provides a hierarchical organization of operational and
configurational data trees. MD-SAL components interact, in the form of ‘consumer’ or ‘producer’ in a given communication. For example, a protocol plugin can be either a producer of information for the underlayer or a consumer of the application instructions provided through the SAL. SAL joins the producers and consumers with the local data store for data exchange. A producer updates the data store and can generate notifications that are published by components for listeners. A producer implements an API and API data that are saved in the data store. A consumer consumes the API through RPC and reads the API’s data from the local data store.

3.5. Distributed Internet Traffic Generator (D-ITG)

D-ITG is a platform that can generate both IPv4 and IPv6 traffic. Generated traffic patterns are defined by traffic protocol, rate, packet size, and inter-departure time between packets. D-ITG supports a variety of protocols, such as TCP, UDP, ICMP, DNS, Telnet, and VoIP. Different probability distributions are also available for sending packets, such as Constant, Uniform, Exponential, Pareto, Cauchy, Normal, Poisson, and Gamma. Traffic information is stored at the sender and receiver sides for each generated traffic flow. A log server can be configured to receive and store the log information for multiple senders and receivers. The D-ITG provides calculated statistics for Delay, Packet Loss, Jitter, and Throughput.

A traffic flow generation involves a traffic sender, receiver, and log servers. ITGSend, the sender component of D-ITG, operates in single or multiple flows sending modes and generates the traffic towards the receiver (ITGRecv). ITGRecv is a continuous listening daemon for any intended connection to the receiver. A thread is generated to handle each flow received. The log server, ITGLog, collects and stores the log information from multiple senders and receivers. The stored traffic information is analyzed by ITGDec tool to provide statistics about the packet delay, packet
loss, jitter, and throughput for each generated flow. The end-to-end delay is calculated as the average time a packet takes to traverse the network from one endpoint to another. Jitter is the variation in the end-to-end delay of sequential packets. Throughput is measured as the average packet/bit rate successfully delivered between the communicating hosts. Finally, the port utilization is represented by the number of recorded bytes passing through the switch port at any given time.

3.6. **Hadoop Big-Data Application**

According to the Oxford English Dictionary (OED) definition, big-data is the “data of a very large size, typically to the extent that its manipulation and management present significant logistical challenges.” The characteristics of big-data are commonly known as four Vs, i.e., Volume for the vast amounts of data, Variety for different forms of data collected, Velocity for rapid generation and frequent moving, finally, Veracity for the uncertainty due to data inconsistency and incompleteness. Recently, the fifth V was added for Value, which is due to the massive value in very low-density data. Big-data applications are now storing and processing a large amount of unstructured, semi-structured, and structured data varying from terabytes to petabytes and up to exabytes. To efficiently manage such data, parallel processing, and data partitioning among multiple cluster nodes are used, which typically requires high traffic volume to distribute data.

![Figure 3.4: simplified Hadoop cluster](image_url)
Nowadays, the industry widely uses Hadoop in big-data applications. MapReduce and Hadoop Distributed File System (HDFS), see figure 3.4, are on top of the Hadoop ecosystem. MapReduce is a software framework for processing and generating distributed data from a cluster. MapReduce is two task parties, namely, Map and Reduce. Map converts a given set of data into another set of data, in the form of (key/value) tuples. Receiving these tuples from Map, Reduce merges these data tuples into a smaller set of tuples. HDFS, the storage part in Hadoop, mainly consists of a single master node, NameNode, and many slave nodes, DataNodes. HDFS divides the data into fixed-size blocks and stores them across all DataNodes in the cluster. The NameNode handles job requests, divides jobs into tasks, and assigns each task to a DataNode. DataNodes are responsible for internally managing and executing the assigned job.

For the network traffic, TestDFSIO and TeraSort are two widely used tools for benchmarking single/multi-node Hadoop clusters. The TestDFSIO benchmark is a read/write test for Hadoop file systems. It is helpful for the discovery of performance bottlenecks in the network. The TeraSort benchmark sorts any amount of data as fast as possible. It is a benchmark that combines testing the HDFS and MapReduce layers of the Hadoop cluster. A full TeraSort benchmark involves three partial steps as follows: generating the data, then sorting the data, and finally validating the sorted data.

3.7. Docker Containers

Docker containers are efficient, lightweight, self-contained systems. A container is a lightweight virtualization that wraps up the code and all its dependencies to be deployed quickly and reliably on different computing environments. A Docker container image is a lightweight package that includes all required code, runtime, system tools, system libraries, and settings for the running
application. Docker images become containers at run time when they run over the Docker engine. Containers virtualize the operating system and provide isolation and allocation benefits to the running application. A virtual machine is an abstraction of the physical hardware that virtualizes one hardware server to multiple servers. Containers are the abstraction of the operating system that packages code and dependencies together. It does not require a full copy of the operating system, application, and libraries. The same machine shares the OS kernel with all containers, with each running in a separate userspace. As a result, containers take up less space than VMs, and can handle more applications and require fewer resources. An architecture for VM and containers is shown in figure 3.5

![Figure 3.5: VM vs Container Architecture](image)

(a) Virtual Machines
(b) Docker Container
Chapter 4

Investigation of Software-defined large-Scale Networks

4.1. Introduction

This chapter investigates the potential of using SDN for big-data applications running in large scale networks. In this context, experiments comprise three components, 1) a big-data application, 2) a connecting network topology, and 3) a packet forwarding mechanism. Hadoop [26] multi-node cluster is the chosen example of a Big-Data application. The Fat-tree network topology is selected since it records a promising result in the investigation of SDN behavior and performance compared to BCube. The Fat-tree and BCube topologies are investigated for different sizes, traffic load, and traffic types, see section 4.2. Finally, the packet forwarding mechanism will be either through an SDN controller or the normal switching mode. Both operating modes are compared at different network scales and for different traffic patterns. The experimental results record the performance of both SDN and normal switching operating modes for reading, writing, and sorting operations as the networks scale-up, see section 4.3.

4.2. Investigating SDN-based Network topologies

Based on the literature review, Fat-tree and BCube are suitable candidates for the present work. Hence, both topologies are investigated under an SDN environment [25], using different network scales, traffic loads, and traffic patterns. The topology with better performance in terms of packet delay, jitter, and packet drops is adopted for further investigations. A BCube-like topology is
considered in our SDN environment due to some limitations in the Mininet host. It is worth to be noted that BCube-like and BCube behave the same when replacing server’s connectivity with OpenFlow switches to enable a full network view for the controller. In a BCube topology, as shown in figure 4.1, every server is prepared by interfaces that tie the number of switch levels, acting as a relay node, and are recommended to be less than four [7]. For a BCube\(_k\) of \((k\text{-levels})\) and \((n\text{-ports})\) switches, the basic cell BCube\(_0\) consists of \(n\) servers, equipped with \((k+1)\) interfaces; each is connected to one \((n\text{-ports})\) switch. BCube\(_1\) contains \(n\) BCube\(_0\) and \(n\) switches, as shown in figure 4.1. The \(i\)\(^{th}\) port in the \(n\)\(^{th}\) switch in the newly added level is connected to \(k\)\(^{th}\) +1 interface of the \(n\)\(^{th}\) server in the BCube\(_k\). During the experimental tests for BCube, the controller should be aware and has a full view of all nodes relaying the packets. So, servers, as packet relaying nodes, need to be part of the communicating OpenFlow entities. Consequently, the BCube-like topology is proposed, by replacing each server in the BCube with an OpenFlow switch, as shown in figure 4.2. This approach enables the controller to have the entire view of the topology, more hosts for the same topology architecture, and fewer ports per server. The servers are no longer relaying packets, and hence the BCube is converted from a server-centric to a switch-oriented topology.

![Figure 4.1: BCube architecture for K=1](image)
The Fat-tree topology, on the other hand, figure 4.3, comprises \( n \)-pods, each with two layers of \( n/2 \) of \((n\text{-ports})\) switches and \( n^2/4 \) servers. Each edge switch in the lower layer is directly connected to \( n/2 \) servers and \( n/2 \) aggregation switches. Each aggregation switch is further connected with \( n/2 \) \( n\text{-ports} \) core switches. There is a total of \( n^2/4 \) of \( n\text{-ports} \) core switches, where each core switch has one port connected to each pod. The \( i^{th} \) port of any core switch is connected to pod \( i \) so that the consecutive port in the aggregation layer of each pod is connected to a core switch on \( n/2 \) strides. A 4-pod Fat-tree network with four hierarchical layers, comprising, from the bottom, hosts, edge switches, aggregation switches, and core switches, is shown in figure 4.3.
4.2.1. Experimental setup and Configuration

The main objective of this part is to adopt one of the two promising, widely used topologies, namely Fat-tree or BCube, for further SDN investigations. Different scales of Fat-tree and BCube-like topologies are compared during the experiments. The small-scale topology is the 4-port capacity switches organized in 3 levels, with a total of 16 servers and 20/24 switches in each topology. The network is then scaled up for both topologies by increasing the port density per switch to 6 and 8 ports, consecutively. Both topologies scale up to accommodate 128 servers and 80 switches. The BCube-like topology can afford more servers, as shown in table 4.1 and table 4.2, but we consider only a reduced number of servers connected per edge switch to have the same size for both topologies. The investigated topologies are illustrated in figure 4.4 and figure 4.5, as presented by the web management interface of the ODL controller.

<table>
<thead>
<tr>
<th>Table 4.1: Topology Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
</tr>
<tr>
<td>Fat-tree</td>
</tr>
<tr>
<td>BCube</td>
</tr>
<tr>
<td>BCube-like</td>
</tr>
</tbody>
</table>

n: number of switch ports
k: number of switch levels

<table>
<thead>
<tr>
<th>Table 4.2: Experiment Topology Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology (3-level)</td>
</tr>
<tr>
<td>Fat-tree</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>BCube-like</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
For traffic patterns, $D$-$ITG$ is used. Three modes of communications are generated for testing both topologies, namely, one-to-one (Unicast), one-to-many (Multicast), and one-to-all (Broadcast). In these experiments, the packet size is set to 1K bytes, with a constant inter-departure time of 1K, 2K, 3K, 5K, and 10K packets per second. The default execution time is nearly 10 secs per connection. Every type of communication, single- or multi-flow, is tested for all bit rates over all traffic patterns. This work adopts the calculated packet losses, minimum delay, maximum delay, average delay, and average jitter, from the ITGDec, as performance metrics. Online cloud-based experiments are conducted, using Mininet 2.2
running on, Ubuntu server of 16 GB RAM and 8 CPUs. ODL “Beryllium” Controller runs on a separate Ubuntu server of 8 GB RAM and a 4-core processor.

4.2.2. Experimental Results

Recorded results of the five-performance metrics, namely, average delay, minimum delay, maximum delay, average jitter, and the packets dropped, are shown in figures 4.6 - 4.9. Each figure displays the results of 4-ports, 6-ports, and 8-ports switches. For every level of scale, different traffic patterns, e.g., one-to-one, one-to-many, and one-to-all are shown for both BCube-like and Fat-tree topology.

Figure 4.6 shows the average minimum and maximum delay on the same plot. For each network scale, the average delay, jitter, and packets dropped increased with the traffic pattern type (e.g., from unicast to broadcast). In figure 4.6(a), for the low-level scale (4-ports switches), BCube-like demonstrates a better average delay performance over all traffic patterns. For the medium-size network (6-port switches) and the one-to-one communication pattern, see figure 4.6(b), both topologies show nearly the same performance, but as the traffic changes to one-to-many and one-to-all, Fat-tree demonstrates better performance. For the large-scale of (8-ports switches), figure 4.6(c), Fat-tree performs better over all traffic patterns.
Figure 4.6: Average Delay of BCube-like and Fat-tree over different scales and traffic patterns
Figure 4.7: Min, Max and Avg Delay of BCube-like and Fat-tree over different scales and traffic patterns

Figure 4.7 shows the minimum, maximum, and average delay trend as the network scales up. In figures 4.7 [a-c], as the network scales up, Fat-tree recorded a consistent behavior in terms of minimum, maximum, and average delay, while delays for BCube-like topology increases more rapidly than that of Fat-tree. For small and medium-scale networks, considering
maximum and average delays, BCube-like slightly outperforms Fat-tree for nearly all one-to-
x traffic patterns, figures 4.7(b) and (c). Fat-tree, however, starts performing better than the
BCube-like at the largest network scale. Figures 4.7 [a-c] show that the delay increases rapidly
over scale in BCube-like topology. On the contrary, Fat-tree steadily outperforms overall
traffic patterns for all scales.

In terms of the average jitter of small-scale, as shown in figure 4.8, BCube-like performs better
over all traffic patterns. When it comes to medium-scale, both topologies perform about the
same, with Fat-tree being slightly better in one-to-one and one-to-many, while BCube-like is
better over the one-to-all communication pattern. For the large-scale network, Fat-tree is
slightly better in the case of unicast and much better in multicasting. Finally, in figure 4.9, the
packet drop percentage is better for BCube-like topology on the small-scale of 4-port switches.
Both topologies are comparable to each other on the medium-scale of 6-port switches. The *Fat-tree* gets a slightly better performance over *BCube-like* at larger scales.

In general, at the small scale of 4-ports switches, *BCube-like* topology outperforms the *Fat-tree* topology for the one-to-x communication patterns and is a favorable 8-ports choice at such scale. As the network scales up to 6-ports and switches, with more added switches and servers, *Fat-trees* show a better performance. For all performance metrics, the performance degradation in both topologies is primarily due to the scaling up of the network. It is worth noting that the degradation in the delay, jitter, and packet drop is either the same or less than the scaling factor of the number of hosts (2.8 on average) or the scaling factor of the number of switches (1.7 on average). The experimental results reveal that *BCube-like* performs better on a small scale, but its performance degrades more rapidly, compared to *Fat-tree*, as the network scales up. Accordingly, *Fat-tree* topology will be sustained in the next part of the experiments, with the big-data application in SDN and non-SDN environments. It is worth noting that the largest scale (8-ports switches) with a one-to-all traffic pattern shows a segmentation error when generating over 112 traffic flows. The error may be due to some limitations imposed by the traffic generator software. The same machine configuration performs well for all other traffic patterns.

### 4.3. SDN support for applications (Big-Data application)

In this subsection, experiments are conducted to study how SDN supports the traffic of big-data applications over different network scales [68]. So, a Hadoop big-data application is installed in Docker containers with a *Fat-tree* topology and OpenFlow switches. Although the Hadoop cluster may be set using virtual machines, a considerable number of CPUs and RAMs are
necessary for such scenario. Hadoop nodes are implemented on separated Docker containers that are connected in a Fat-tree topology. The packet forwarding is operating in one of two modes of operation, namely, “normal” and “odl-l2switch” modes. In the “normal” mode of operation, the OpenFlow switches operate on MAC-Port mapping as any L2 layer switch. OpenFlow switches are configured for “normal” mode operation by adding a normal mode flow entry. When the incoming flows hit this entry, the MAC table maintained by OpenFlow switch will be referred for forwarding decisions. Also, the Rapid Spanning Tree Protocol (RSTP) is enabled to avoid loops in the proposed topology. However, in the “odl-l2switch” mode of operation, the centralized SDN controller is responsible for setting and managing all flows through all OpenFlow switches. The SDN controller is OpenDaylight “Boron” release, running on a standalone server, with enabled “odl-l2switch” features.

4.3.1. Experimental Configuration

Different scales are considered in the experiments. Switch capacity is increased as we scale up the network size, starting by 4-, 6-, up to 8-, and 10-ports per switch. Expanding the network size allows hosting 16, 54 up to 128, and 250 hosts, respectively. The number of switches is increased from 20 up to 125 switches from the small scale to the largest 10-ports capacity switches, respectively. A three-level Fat-tree topology is used for all switch capacities. Figures 4.10 [a-d] show, as presented by the ODL web interface, the network topological views for different scales. A Hadoop cluster is also scaled up accordingly, starting with a cluster of 8 nodes, up to 18, 32 nodes, and finally 50 nodes for the largest scale network. For each network scale, the same file size is used for read/write and sort tests. Hadoop cluster nodes are organized in such a way that each node is attached to exactly one edge switch. The packet forwarding in each topology scale is either a normal mode of layer-two switching or
using the L2 switching in ODL. Table 4.3 summarizes different network sizes, Hadoop clusters, and the capacity of switches and servers.

![Diagram of network topology](image1)

(a) 4-ports
(b) 6-ports
(c) 8-ports
(d) 10-ports

Figure 4.10: Different scales of 3-level Fat-tree

<table>
<thead>
<tr>
<th>Topology</th>
<th>Total ports (n)</th>
<th>Levels (k)</th>
<th>Total Switches (\frac{n^k(k-1)}{2})</th>
<th>Edge Switches (\frac{n^k}{2})</th>
<th># max server (\frac{n^k}{4})</th>
<th>Hadoop nodes</th>
<th>File size (R/W/SORT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>20</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8x 1GB</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>3</td>
<td>45</td>
<td>18</td>
<td>54</td>
<td>18</td>
<td>18x1GB</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
<td>80</td>
<td>32</td>
<td>128</td>
<td>32</td>
<td>32x1GB</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>3</td>
<td>125</td>
<td>50</td>
<td>250</td>
<td>50</td>
<td>50x1GB</td>
</tr>
</tbody>
</table>

Table 4.3: SDN-Fat-tree experimental topology for application support
Figure 4.11 illustrates how Hadoop nodes participate in the small-scale network. The *Fat-tree* topology accommodates OpenFlow switches, running OpenFlow1.3 protocol. Mininet 2.2 is used for emulating the network elements by predefined Python scripts. The Hadoop Docker containers are attached to the created Mininet topology, as Mininet hosts, by subclassing the original host class in Mininet. Network traffic is generated by benchmarking the Hadoop cluster using two widely used tools, namely, *TestDFSIO* and *TeraSort*. *Throughput* in *Mbps* and the *execution* time in *sec* are the metrics used for performance testing of *TestDFSIO*. The *execution* time in *sec* for completing the sorting task is the performance metric of the *TeraSort*. All experiments for both the *TestDFSIO* and the *TeraSort* are repeated 10 Times where the *average, the minimum, and the maximum execution* times and *throughputs* are recorded and presented in figures 4.12 - 4.17.

Figures 4.12 - 4.17 show the performance metrics for the reading, writing, and sorting tests over the various scales; the small scale starting by the 4-, 6- and 8-ports switches, up to the largest scale of 10-ports switches. For each scale, the same file size is used for reading,
writing, and sorting operations. For the smallest scale of the 4-ports switch, an 8-GB file size is used. As the network scales up, the file size is increased to 18GB, 32GB, and 50GB for the scales 6-, 8- and 10-ports switches, respectively. The performance metrics for each scale are recorded in both “normal” and “odl-l2switch” modes of the network operations.

For the 4-, 6-, and 8-ports scale experiments, all switches, hosts, and Docker containers run on an online cloud server, with 64 GB RAM and an 8-core processor. The server is then upgraded to 128 GB RAM and a 16-core processor to handle the creation of the 10-ports scale topology, which accommodates a total of 125 switches and 50 Docker containers. The controller is hosted on a stand-alone 8 GB RAM server with a 4-core processor, which is upgraded to 16 GB RAM and eight cores for the enhanced largest scale version (10-ports scale). It is worth noting that the presented single controller will not be able to maintain the same efficiency level with large-scale networks. A study is conducted to analyze how upgrading or clustering the SDN controller enhances the efficiency of the controller when managing very large-scale networks. An upgraded controller with Single Node Clustering added feature, namely, “odl-mdsal-clustering,” is tested for the largest 10-ports scale.

4.3.2. Experimental results

Figures 4.12 - 4.17 show the performance metrics for each scale in both “normal” and “odl-l2switch” modes of the network operations. Figure 4.12 shows the execution times of reading and writing over different scales. The “odl-l2switch” mode shows better performance over the normal switching mode for all metrics (average, minimum, and maximum execution times) over nearly all scales. In a write operation, the execution time of ODL was improved on average by 5.5%, 9.3%, and 20% for the small scales of 4-, 6- and 8-ports switches, respectively. This improvement was not noticeable for the largest scale, 10-ports switches.
with 125 switches. This is due to the inefficient handling of a single controller when facing a large amount of traffic generated from the write/read testing at the 10-ports scale. The latter case was improved when upgrading the controller to a higher configuration and adding a clustering feature, “odl-mdsal-clustering,” over the single node. Figure 4.12 (d) shows the performance metrics when the upgraded controller is considered. Better performance is achieved for the newly configured SDN controller over the normal forwarding with 7.2% and 8.3% improvements on the average execution time for the write and read operations, respectively.

![DFSIO Read/Write Execution Time](image)

**Figure 4.12: Execution time Read/write all scale**

For the same scales of the network, figures 4.13[a-d] illustrate the throughputs. The average throughput, using “odl-l2switch” mode shows an improvement of 10.3%, 18%, and 38% over the “normal” mode for the scales 4-, 6- and 8-ports, respectively. Similarly, better performance in throughput is achieved using the upgraded controller when the largest scale of 10-ports switches is considered. The average throughput in writing recorded nearly 11.4%.
Figure 4.14 and figure 4.15 summarize the execution time and throughput, overall network scaling, for the read and write operations.
For the sorting test, the execution time for each scale is shown in figure 4.16. Using ODL reduces the average and maximum execution times over all scales. The improvements in the average execution times are 2.2% and 8% for the 4- and 6-ports switch topologies, respectively. As we test the larger scale of 8- and 10-ports, using ODL still records some improvements over the normal mode. However, using a single controller slightly affects the improvement at such scales. A large amount of traffic generated, at these scales, needs a more powerful clustered feature of controllers is recommended to maintain the increase in performance. Table 4.4 summarizes the improvement results. Figure 4.17 summarizes the overall average execution time for the sorting operation as the network scales up.
Chapter 4: Investigation of Software defined large Scale Networks

Figure 4.16: Sort Execution time

Figure 4.17: Sort Execution time overall scales
Chapter 5

The Proposed QoS Framework

5.1. Introduction

Quality of Service (QoS) is the capability of a network to provide better service to designated network traffic. Two necessary components are essential for delivering QoS, namely, monitoring and management facilities. For monitoring traditional traffic, either an *active* or a *passive* measurement method is usually applied. For *active* measurements, additional packets are typically injected in the network; commonly, ping packets are used to determine the latency loss and jitter. On the other hand, a *passive* measurement does not impose any overhead packets on the network. However, *passive* measurement is not always feasible and may not be accurate for some network metrics.

In an SDN environment, the centralized controller facilitates the monitoring and control of all network elements. OpenFlow is usually the chosen protocol for communicating the controller with network devices. OpenFlow enables collecting statistics about switches, ports, and flow tables of network switches, that showed its usefulness in reflecting the network status. However, other valuable metrics are not directly available as *port utilization*, *delay*, and *jitters*. *Port utilization* can be calculated using some port statistics, while the *delay*, *jitter*, and *path losses* need some added features to be measured.
5.2. **Proposed SDN Framework - Functional Overview**

The proposed framework measures different transmission metrics based on the centralized controller. Also, it adopts probing agents for measuring some parameters that are not provided by the controller. The proposed framework automatically captures the event of any initiated communication and calculates the forwarding paths using the externally developed modules. The externally developed modules provide route selection, monitoring, and congestion detection and control functionalities.

The route selection is based on either the unweighted Dijkstra shortest-path or a QoS-enabled route selection. A single/multipath route is provided for classification and metric-based routing functionalities. Any simple/complex algorithm can be implemented for shortest-path detection. In the *multipath* route calculation, the module calculates all possible short paths between the source and the destination, compares them using the "SequenceMatcher" class from the "difflib" python module, and finally considers the two paths with the least similarity ratio value. The two paths are selected with minimum overlapping hops, maximum disjoint links. This minimizes the shared links and reduces the congestion between communicating nodes. The metric-based route selection is based on the shortest-path that has the best performing active/passive measured metric, namely, end-to-end delay and path utilization. Any other active/passive measured metric can be adopted for selecting the best path, such as jitter, packet rates, and packet losses.

The monitoring modules use the port utilization metric to reflect the network status and traffic metrics. It is also responsible for extracting the network graph information from the ODL controller. This information is used for forwarding and monitoring activities. The ODL controller continuously updates the network graph status that reflects the utilization of different links and ports. All ports' statistics from the controller are extracted at fixed intervals. All statistics and...
information are maintained locally in matrix forms and saved time stamped into files. The monitoring module also identifies the edge switches to be solely monitored. Edge switches monitoring helps to identify the ingress and egress traffic to and from the network. It also provides a full traffic matrix for all communicating pairs in the network. Congestion detection and control are also provided through the monitoring module. A subset of the network switches is monitored for excess port utilization, which is exceeding a predefined threshold, e.g., 70%. If a congested link is detected, the module selects a subset of the traffic passing through the congested link to be rerouted on a less utilized alternative path. The route selection module is triggered to assign a new metric-based route for the selected traffic subset. The subset flows are selected provided that they will not transfer the congestion to another part in the network.

The proposed framework automatically detects any initiated communication based on some registered event notification in the operational data store of the ODL controller. For forwarding decisions, multiple candidates are compared based on the calculated active/passive measured metric. The selected QoS-enabled path is compared to the unweighted Dijkstra shortest path and “odl-l2switch” provided through OpenDaylight. The “odl-l2switch” forwarding acquires information about the nodes, hosts, and links (via ARP & LLDP) packets. It removes loops and allocates the flow on the switches based on any “packet_in” message, indicating a pair of communicating nodes. Also, the proposed framework helps set the topology of switches by configuring the needed forwarding rules. The developed “route preparation” module prepares the required flow rules for configuring the best paths. Flow rules are pushed to the controller to be placed on the designated switches. Rules are set with different priorities and timers that match specific host/application requirements. Actions are taken based on the matching part of the flow rule. The proposed framework can fit any topology that provides multiple paths between any pairs
of nodes. Fat-tree is an example that is widely used in today’s data center. It showed its efficiency in previous work [3] when tested over different scales and traffic patterns.

5.3. Proposed SDN Framework Building Blocks

In this section, we describe the main building blocks of the proposed framework. The proposed framework enhances the SDN controller forwarding and monitoring capabilities. It uses the internal network statistics provided through the ODL controller to calculate the path utilization metric. Also, it uses probe agents on the communicating hosts, to estimate the end-to-end communication delay needed for forwarding decisions. The presented modules are externally developed on top of the ODL controller. They are written in python and use REST API for communicating with the controller.

![Figure 5.1: QoS framework Architecture](image)
Figure 5.1 presents the three layers of the framework. The developed modules are the top layer of the framework. The ODL controller, in the middle layer, communicates with the upper and lower layers via REST and OpenFlow plugin, respectively. The controller performs the basic functions such as topology discovery, managing/collecting network statistics, and mapping the application requirements down to the OpenFlow switches. The lower layer has hosts connected through the edge switches forming the network topology. The main functions of the external modules are described as follows:

1. **Event Notification listener**: A module through which notifications are received from the controller’s REST API. It handles notifications indicating a detected pair of communicating devices. Accordingly, it triggers the “Route Selection” module to start calculating the forwarding paths. The event notifications are generated by the OpenDaylight controller based on some registered change-event notifiers. These notifiers are attached to some parts of the controller’s operational data store.

2. **Route-selection**: is the forwarding decision-maker module. It calculates the forwarding path between any two nodes, using the shortest path algorithm either with or without QoS considerations. Shortest path candidates are initially based on the hop counts. The best shortest path with QoS consideration can either consider the delay or the utilization metric. The path with lower-cost is considered for forwarding the traffic. For the delay metric, the end-to-end delay is considered as the cost of every single path. In the port utilization metric, the short path is selected based on the graph information extracted from the controller, and the port utilization as in equation (5-2). A short path between two nodes is calculated as follows, assume a path $P$ between the source and the destination nodes. A path $P$ in the network is made up of several links $l$, $l \subset P$. If the link utilization is $U_l$ then
the total path cost $C_p$ that has $x$ links can be calculated using equation (5-1). In other words, the highest utilized link along the path reflects the path cost.

$$C_p = \max_{i=1\rightarrow x} U_i \quad (5-1)$$

The route selection module is triggered upon receiving a notification, indicating a pair of communicating nodes. It uses the network graph information, extracted from the SDN controller, and the current state of the network, e.g., delays and port utilization, to select the best path between the source and destination, the module pseudo code is shown in Algorithm 5.1.

**Algorithm 5.1: Route Selection Module**

*Get Two Communicating nodes from Notification listener*

begin

.....

g = get_graph()
Candidates = getCandidatePaths (g, Src, Dst)
Dijkstra path = getDijkstra (g, Src, Dst)
If mode == UtilizationQoS:
   Then:
      UtilizMatrix = GetUtilizationMatrix()
      PathsCost = GetUtilizationCost(UtilizMatrix, Candidates)
      Best path = Candidates [Min pathsCost]
      sendtoRulePrepModule (Best path, Src, Dst, flowID, priority)
   elseif mode == delayQoS:
      Then
      ports list, paths list = setProbFlowRules (Candidates)
      Path labels = createLabeledList (ports list, paths list)
      Delays = ProbLinks (Path labels)
      Min index = get index (min (Delays))
      Best path = paths list [Min index]
      sendtoRulePrepModule (Best path, Src, Dst, flowID, priority)
   else:
      sendtoRulePrepModule (Dijkstra path, Src, Dst, flowID, priority)
end

The module also initiates a probe request for updating the delay statistics of all paths between any two communicating nodes. The request is sent to the “Resource Monitoring” module to scan the delays of all candidate paths between the given pairs of nodes. Finally,
the selected best QoS path is passed to the “Rule preparation” module to be pushed to the controller. The module also provides a multipath route calculation, for traffic classification, the module calculates all possible short paths between the source and destination, compares them using the “SequenceMatcher” class from the python “difflib,” and finally considers the two paths with the least similarity ratio value. The two paths are selected with the minimum overlapping hops, maximum disjoint links. This minimizes the shared links and reduces the congestion between communicating nodes.

3. **Rule preparation:** This module is responsible for the switch flow programming. The selected routes for communication are analyzed, and the required rules for traffic forwarding are prepared. The module decomposes each path to primary forwarding nodes, identifies the ingress and egress ports, prepares the needed rules to be configured in these nodes, and finally sends these rules to the controller to be set on the designated switches, see Algorithm 5.2. The rules are placed in the controller configuration store. Then, the controller’s flow programming service writes all flows down into the active operating OpenFlow switches, and packets transparently start going through the newly installed path. The pre-configured rule priorities and timer are used to balance how traffic is forwarded at any given time.

**Algorithm 5.2: Rule Preparation Module**

```plaintext
Get path route from Route Selection Module
begin
    ....
    for i in range (1, len(path) - 1, 1):
        edge_egress = find_edge (path[i], path[i + 1])
        port_egress = getIndex (edge_egress)
        nodeID = getnodeID (path[i])
        newFlow, Url = prepareRule (NodeID, FlowID, src, dst, timeout, priority,
                                   port_egress)
        sendtoController (newFlow, Url)
end
```
4. **Resource Monitoring**: A module that is responsible for frequently updating metrics stored in the local “Data Store. The operational ports and packets statistics are queried from the controller operational data store at fixed intervals. The port utilization matrix is calculated and updated based on the total number of bytes transmitted and received on each port. It is updated every 4 seconds (fixed intervals) to reflect the current status for all ports and links. In this work, the utilization is calculated, as in equation (5-2), based on the total Bytes received during the time interval (s). Where $S_i$ and $R_i$ are the count of sent and received bytes in port $i$, and $t$ is the recording time.

\[
\text{Port Utilization} = \frac{(S_i^t - S_i^{t-\Delta t}) + (R_i^t - R_i^{t-\Delta t}) \times 8}{\text{Port Speed} \times \Delta t} \tag{5-2}
\]

Link and port-utilization matrices, and flow statistics are periodically saved time stamped in external files. Moreover, this module acquires the end to end link delay from the probe agents installed at the end host devices. The delay is calculated on demand for multiple paths as soon as any pair of communicating nodes are detected. The monitoring module also extracts the nodes’ topology and link’s information from the controller and stores them in the local “data store” to be used by other modules.

5. **Data Store**: is the data repository for the network graph, information, and statistics. It holds the topology description, collected network utilization status, and the end to end delays of different paths.

6. **Congestion detection and Control**: A module that considers port utilization as a metric for congested links. It continuously monitors the aggregation layer of the fat-tree topology, seeking for a congested port that surpasses a certain predefined utilization threshold (e.g., 70%). Upon detecting a congested port, it selects the best candidate subset flow for rerouting. Then, the route selection module is triggered to discover new routes.
for a subset of the flows passing through that congested port. The “Route-selection” module calculates the best candidate route based on the provided QoS measured metrics. Once a congested port is detected, with the SDN controller having a full view of the port status, the new calculated path cost should exclude the congested link, provided that a short path exists with a better cost. A newly calculated best-utilized path will be assigned as a new path between the source and the destination of the rerouted flow, as long as such rerouting will not transfer congestion to another port.

An optimal choice to guarantee the efficiency of the traffic subset rerouting phase is considered. At any given time $t$, The congested port $p$ on the switch $s$ has several passing flows $(f_1, f_2, f_3, \ldots, f_k)$ with utilization costs $(c_1, c_2, c_3, \ldots, c_k)$. Each traffic utilization cost is a portion of the overall utilized link. These costs are pulled out from a continuously maintained traffic matrix for all communicating nodes. The candidate list of flows is checked one flow at a time from the costly flows down to less costly flows. A condition to reroute a traffic flow $f_k$ with cost $c_k$, is that the cost of the selected path $C_{P_{new}}$, after adding the flow $(f_k)$ cost $(c_k)$ should be lower than the utilization of congested port. At any given time, $t$, the controller gains access to the information of all flows passing through port $p$. The cost of the selected path $C_{P_{new}}$ after rerouting one or more traffic flows is the key by which the path is selected by flow $f_k$ and whether or not a rerouting process would be valuable. Any selection criteria or algorithm can be applied to get the best load distribution among different paths. In the proposed algorithm, the flows are checked, starting with the costly flow and ending when none fulfills the condition. The highest costly flow has the highest probability to be rerouted, provided that the condition is satisfied. Otherwise, the next costly traffic is checked for meeting the
condition, as presented in Algorithm 5.4. The stated condition guarantees that no rerouted flow will transfer the congestion to another port in the network. That is to say, the most significant possible passing flow, through the congested port, is nominated for rerouting without transferring the congestion to any other link. The rerouting is based on selecting the shortest path that owns the least utilized cost, $C_p$.

**Algorithm 5.4:** detecting the congested port

```python
def monitor_aggregation():
    while 1:
        portUtilization = get_utiliz_agg_sw()
        Cport, switchID, Cportcost = get_cong_port_cand(portUtilization)
        if Cport > Threshold:
            flowList, flowCost = portFlowInformation(switchID, Cport)
            while (len(flowCost) != 0):
                CandidateFlowIndex = max(flowCost)
                Remove_candidate_flow_from_list
                Csrc, Cdst, Cost = getFlowInfo(CandidateFlowIndex)
                adjSw = get_adj_sw(graph, switchID, Cport)
                CbestPathList, CpathCostList = get_Candidatepaths(Src, Dst, adjMatrix)
                selectedPathIndex = min(CpathCostList)
                selectedPath = CbestPathList[selectedPathIndex]
                selectedPathCost = CpathCostList[selectedPathIndex]
                if len(selectedPath) > 5:
                    CandidatecommonLinkCost = Ccost
                    if selectedPathCost + Cost < Cportcost:
                        set_new_path(Src, Dst, selectedPath)
                        break
                    elif (adjSw $\in$ graph.CoreSwitch) and (Cost<$\text{Cportcost}$) and (switchID $\in$ selectedPath) and (selectedPathCost == CandidatecommonLinkCost):
                        set_new_path(Src, Dst, selectedPath)
                        break,
                    elif (adjSw $\in$ graph.edgeSwitch):
                        del temp_best_path_list[selectedPathIndex]
                        del cost_list[selectedPathIndex]
```

It is worth to be noted that, SDN controller, based on the utilization information of the aggregation layer switches, can predict any congestion at any part of the network, e.g., core, edge, and the aggregation layer switches. Ports, in the aggregation layer switches, can either connect edge or core switches. As described before in section 4.2, a fat-tree topology can hold up to $n^3/4$ hosts, connected to $n^2/2$ edge switches. The total number of core switches is
(n/2)^2 and the aggregation layer includes n^2/2 of n-port switches. The aggregation switches represent only 40% of the total number of used switches. Thus, monitoring the aggregation switches only should reduce the overhead of detecting the congested port by 60%.

7. **Probe Agents**: are agents installed on the end hosts that are triggered on demand to identify delays over multiple paths. They use several threads pinging different paths to get the delay of each path. Each agent aims to measure the real-time latency for any requested path. The agents receive ports’ numbers corresponding to each path and returns the delay of that path, Algorithm 5.3.

```
Algorithm 5.3: Probing agents
begin
    conn, addr = acceptSocketConnection()
    dstIP, portlist = receivedata()
    for i in portlist:
        worker = Thread (target=ping, args= (i, dstIP,))
        for i in range (0, len(ports), 1):
            Delays = prepareDelayReplies()
            conn.send(Delays)
    conn.send(Delays)
end
```
6.1. Introduction

This chapter shows, via experiments, how SDN can provide a level of QoS based on application classification, active/passive measured metrics rerouting, and congestion control. The proposed framework is tested for measuring different transmission metrics based on the centralized controller and probe agents distributed at the end host devices. The proposed framework modules capture the event of any initiated communication and calculate the forwarding paths. The route selection is based on either the unweighted Dijkstra shortest-path or a QoS-enabled route selection. The QoS-enabled route selection is based on the short-path that has the best performing active/passive metric, namely, end-to-end delay or path-utilization. Other active/passive measured metrics can be adopted for choosing the best path, such as jitter, packet rates, and packet losses. The monitoring modules use the port utilization metric to detect network status and traffic metrics. The selected QoS-enabled path is compared to the unweighted Dijkstra shortest path and "odl-l2switch" provided through OpenDaylight.

6.2. QoS through application classification

This section shows how SDN can be used to provide a level of QoS through application classification [69]. The traffic is classified based on the application type. The "Route-calculation" module computes the appropriate routes based on the application's port numbers and
Chapter 6: Experimental results for Proposed Framework

the IP of the source and destination node. Paths with the minimum overlapping hops, maximum disjoint links, are selected for traffic forwarding. Finally, the “Flow preparation” module, based on the paths chosen, defines the forwarding nodes and ports, prepares the required forwarding rules, and notifies the controller to install these rules into the designated switches.

6.2.1. Experimental setup and Configuration

As shown in figure 6.1, the experiment set up considers the developed external application modules on top of the OpenDaylight controller. The fat-tree of 4-pods is the connecting topology with twenty (4-ports) switches organized in 3 levels and a total of 16 hosts. Different types of traffic are observed; namely, UDP, TCP, VOIP, and a Big-data Hadoop read/write traffic. Figure 6.1 shows the nodes engaged in each type of traffic generated.

A two phases experiment is conducted. In the first phase, only the UDP, TCP, and VOIP traffic are considered. In the second phase, the Hadoop Big-data application is added to the previously considered traffic types. D-ITG is used to create a total of 3 flows, a flow of each UDP, TCP, and VOIP traffic type,
from the sender to each receiver. The network has four receiver agents, one in each pod, forming a total of 12 flows in the network. The packet size is set to 512K bytes, with a constant inter-departure time of 2K, 4K, and 8K packets per second. The execution time is set to 200-seconds per run with a total of 10 runs. The delay, jitter, and packets dropped are calculated for each communication flow. In the second phase, UDP, TCP, and VOIP traffic are added to Hadoop running read/write benchmark. For the network traffic, the TestDFSIO read/write test is used for Hadoop file systems. Mininet 2.2 running on a Ubuntu machine of 32 GB RAM with a Xeon processor is used for simulation. The ODL Controller “carbon” release, running on a separate Ubuntu personal machine of 8GB RAM, communicates with OpenFlow switches in the fat-tree topology. A total of 16 hosts are connected to the edge switches. The hosts are either, Hadoop nodes, D-ITG traffic sender, or D-ITG traffic receiver.

6.2.2. Experimental results

Results for the five performance metrics, namely, average delay, minimum delay, maximum delay, average Jitter, and packets dropped, are shown in figures 6.2-6.4. Each figure shows the traffic of UDP, TCP, and VOIP applications. Each application is tested with different packet rates of 2K, 4k, and 8K packets per second. The average overall rates for every metric is considered for plotting. The three modes of packet forwarding, namely, odl-l2switch, single-path, and multipath forwarding, for each traffic type, are tested.
Figure 6.2: Delay (UDP, TCP, and VOIP)

Figures 6.2 [a-c], shows the average, minimum, and maximum delays on the same plot. The average, minimum, and maximum delays are improved as the forwarding mode changes from odl-l2switch to single-path and multipath forwarding with the best performing multipath forwarding. The improvements are noticed over all traffic rates and for all types of traffic. It is worth to be noted that, at a low rate (2K), all forwarding modes record nearly a comparable maximum delay. In figures 6.3[a-c], using single-path and multipath routing outperform the open-daylight normal switching, odl-l2switch, in terms of packet dropped for all traffic rates and types. Multipath routing again records the best performance overall traffic rates and types. In Figures 6.4[a-c], multipath forwarding recorded nearly the best average jitter overall traffic types and rates.

Figure 6.3: Packet dropped (UDP, TCP, and VOIP)
Meanwhile, the average jitter for single-path forwarding outperforms odl-l2switch for all traffic rates and types.

![Figure 6.4: Average Jitter (UDP, TCP, and VOIP)](image1)

![Figure 6.5: Monitoring port#1 on switch#6](image2)
Figure 6.6: All Switches traffic over time UDP, TCP, and VOIP traffic
The Monitoring module is used to analyze the port status during the experiments, as shown in figure 6.5 and figure 6.6. Figure 6.5 shows how the traffic load is reduced on port#1 of switch#6 when using multipath over single-path forwarding. Figures 6.6[a-c] show the switches' traffic load over all ports in bytes for the three forwarding modes odl-l2switch, single-path, and multipath. As we can see from the figures, figure 6.6(a) shows the highest traffic in the network switches when using the odl-l2switch mode, recording 338M bytes on average, and a maximum of 695M bytes traffic overall switches traffic. In figure 6.6 (b) and figure 6.6 (c), using the single-path and multipath forwarding reduced the overall average bytes to around 161M and 177M bytes, respectively. The maximum bytes are reduced from 695M in the odl-l2switch to 458M in the single-path and down to 440M in the multipath. The monitoring module can also be used to analyze the traffic per port, as illustrated in figure 6.6.

For the second phase of the experiment, a Big-data Hadoop traffic is added into the network. The testDFSIO benchmark is used for writing a 4GB file over the eight different Hadoop cluster nodes. The execution time and throughput are recorded for both reading and write operations, as shown in figure 6.7 and figure 6.8. The testDFSIO is used to test the Hadoop read and write traffic either solely or with the existence of other traffic types in the network.

In figures 6.7[a-b] and figures 6.8 [a-b], the three left bars, show Hadoop traffic running solely (without UDP, TCP, or VOIP traffic) in the network for each of the three forwarding modes odl-l2switch, single-path, and multipath. In this case, for the multipath forwarding, a subset of Hadoop node's traffic is routed on the second path. For the testDFSIO Hadoop traffic running solely, figures 6.7[a-b] and figures 6.8 [a-b] show nearly the same throughput and execution-time for all forwarding modes. Next, Hadoop traffic is tested along with TCP, UDP, and VOIP. The read/write throughput and execution time are again calculated as the Hadoop is running with other...
traffic in the network. Figures 6.7[a-b] and figures 6.8 [a-b] show throughput and the execution time (the three right bars) for the odl-l2switch, single-path, and multipath forwarding modes for the read/ write operations. The read and write throughput increases as the forwarding mode changes from odl-l2switch to single-path and multipath, see figures 6.7[a-b]. The execution time, figures 6.8[a-b], is also reduced when using the multipath over the single-path and odl-l2switch.

![Figure 6.7: testDFSIO throughput](image1)

![Figure 6.8: testDFSIO Execution time](image2)

Figures 6.9[a-c] show the monitored forwarded traffic over ports of every switch for Hadoop traffic only (without UDP, TCP, or VOIP). Using the developed module reduces the traffic from an average of 100K bytes with odl-l2switch forwarding to around an average of 18K bytes with the single-path and the multipath forwarding. Moreover, the maximum port's traffic of a switch is reduced from an average of 219K bytes in the odl-l2switch to nearly 108K bytes for the single-path and multipath modes. Figure 6.9 (b), also pointed out that the traffic in core switch #4, aggregation switch #6 and the edge switch #7 have higher averages compared to other switches. Using the multipath forwarding reduced the average traffic for switches #4 and #6, as shown in
Figure 6.9 (c). It is worth mentioning that Hadoop is showing lower overall traffic in this experimental scale, compared to the first phase of UDP, TCP, and VOIP traffic.

6.3. SDN QoS through Active monitoring measured metrics

In this section, active monitoring measured metrics are used for evaluating the proposed framework [70]. Probe agents on the communicating hosts calculate the end-to-end communication delay that leverages the capability of the SDN controller to enhance the forwarding decisions. Accordingly, forwarding decisions could be based on the minimum delayed links, thus providing a better QoS. A Fat-tree topology, with a basic building block of 4-ports OpenFlow-switches, holding up to 16 hosts is used for the experimental tests, see figure 6.10.

![Diagram of SDN QoS measurement](image-url)
Figure 6.10 is a snapshot of the Fat-tree topology, with source h1 communicating with destination h16. All possible shortest paths, p1, p2, p3, and p4 between h1 & h16 are highlighted along with the corresponding link delays. The controller received a message from the h1 edge switch indicating a new flow between h1 and h16. Based on the internal odl-l2switch feature, the controller reacts by setting flow routes between the communicating nodes h1 and h16. Simultaneously, a notification event generated by the controller is received via the “Event Notification listener” to the “Route-selection” module of the proposed framework, figure 5.1. The “Route-selection” module selects a path to forward the packets based on either the unweighted shortest path (SP) mode or the weighted QoS shorted path (QoS-SP) mode. For the SP mode, the unweighted Dijkstra calculates the shortest path between the communicating nodes. For the other forwarding mode, QoS-SP, the module calculates all possible shortest paths, initiates a probe request to the “Resource Monitoring” module to scan the delays of the different paths,
and picks up the path with the minimum delay. For both forwarding modes, the final selected paths between the source and the destination are sent to the “Rule preparation” module, which prepares all necessary forwarding rules for the designated intermediate hops between the source and the destination and passes them to the controller. Being informed with the new configuration rules, the controller installs each rule on the designated OpenFlow switch.

6.3.1. Experimental Setup and Configurations

In this part, experiments are set up using OpenDaylight and Mininet. A Fat-tree topology, with twenty 4-ports Open-Flow-switches, holding up to 16 end hosts. Some links are configured with a 2ms delay while others have a 1ms delay, as shown in figure 6.10. The end hosts are Mininet hosts running a Linux system. The forwarding path selection is either based on the odll2switch feature of the OpenDaylight controller or using the Route-selection module of the proposed framework. The odll2switch was configured to act reactively to a new flow, with no proactive flooding rules being installed. The Route-selection module operates in either the unweighted shortest-path (SP) or the QoS-enabled shortest-path (QoS-SP) forwarding mode. The end-to-end delay in a path was considered as a QoS metric.

In other words, three forwarding modes are pulled for comparison: (1) odll2switch built in the open-daylight, (2) Shortest-Path unweighted Dijkstra’s shortest-path algorithm through the external module, and (3) weighted QoS Shortest-Path delay-based shortest-path enabled through the presented module. Different traffic applications are investigated for each forwarding mode. Experiments are set up that considered pairs from different pods in the Fat-tree topology. The averages of all experimental results are recorded for calculating the performance metrics of three applications, namely, TCP, UDP, and VOIP. The traffic of tested applications is generated using
Chapter 6: Experimental results for Proposed Framework

*D-ITG*, under the three forwarding modes, and for different pairs of communicating hosts. Different packet rates of 1000, 2000, and 4000 packets/s are injected for the TCP and UDP applications, and the averages of all test results are recorded for comparison. The run time for each experiment is bounded to 200 seconds, and each trial with the same configuration is repeated five times, for each rate and each communicating pair. For the communicating nodes, all possible pairs from two pods are tested, and the average of all test results are recorded. The considered performance metrics are *packet losses, minimum delay, maximum delay, average delay*, and *average jitter*. Open daylight controller “Nitrogen” release and OpenFlow switch nodes are used. Hosts are running on Ubuntu laptop with Xeon processor and 32GB RAM.

### 6.3.2. Experimental Results

The recorded experimental results for the five-performance metrics, namely, *average delay, minimum delay, maximum delay, average jitter*, and the *packets dropped*, are shown in figures 6.11-6.13. Using the proposed framework showed its efficiency over the standalone controller for all performance metrics. The *minimum* and *maximum delay* values are presented alongside with the *average delay* on the same plot for different forwarding modes. Figures 6.11[a-c] show TCP, UDP, and VOIP delays for the three tested forwarding modes, namely, *odl-l2switch, Shortest path (SP)*, and the weighted *QoS shorted path (QoS-SP)* modes.

Figures 6.11 [a-c] show that the average and minimum delays decrease as the forwarding mode changes from *odl-l2switch*, to *SP* and the *QoS-SP* modes. The enhancements in the average delay for both *SP* and *QoS-SP* modes over the *odl-l2switch* mode considering TCP, UDP, and VOIP applications are 77%, 44%, and 50%, respectively. The overall *average delay* enhancement
recorded 57% for all applications. It is worth noting that the above-mentioned \textit{average delays} are much closer to their corresponding \textit{minimum delays}, figures 6.11 [a-c].

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{delay_graph.png}
\caption{Min, Max and Average Delay of different application and odl-l2switch, SP, and QoS-SP forwarding modes}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{packet_drop_graph.png}
\caption{Packet Dropped for different application and odl-l2switch, SP, and QoS-SP forwarding modes}
\end{figure}
The packet drops for TCP, UDP, and VOIP applications are shown in figures 6.12[a-c]. The enhancements in the packet drops reduction using SP or QoS-SP mode over the odl-l2switch mode considering TCP, UDP, and VOIP applications are 46%, 77%, and 78%, respectively. The overall packet drops reduction is 67% for all applications. Finally, the jitter presented in figures 6.13[a-c], shows a steady decrease when applying the proposed framework on TCP and VOIP applications. For the UDP traffic, the jitter is comparable to the three modes. The average jitter for all applications shows a 25% enhancement.

![Average Jitter](image.png)

Figure 6.13: Average jitter for different application and odl-l2switch, SP, and QoS-SP forwarding mode
Chapter 6: Experimental results for Proposed Framework

Figure 6.14: Ports Utilization for odl-l2switch, SP, and QoS-SP forwarding modes

The port utilization for all switches, over the three tested packet rates (1K, 2K, and 4K packet/s), is shown in figure 6.14. Using the proposed framework, reduced the average ports utilization from 23.3M for odl-l2switch mode to around 15.9M for SP and QoS-SP. For all experiments, the overall average port utilization was reduced by 32%. For a fair comparison, in figure 6.14, the averages are presented with and without excluding the un-utilized (zero-values) switches. Figure 6.14 (a) and figure 6.14 (b), indicated more idle switches when using the odl-l2switch and the SP forwarding. Using the QoS-SP enabled fairly redistributing the load over all switches and making use of the un-utilized (idle) switches. This helps minimize the burden over congested ports,
causing a reduction in the overall average port utilization. It is also worth noting that the average port utilization of all switches is much closer to the average port utilization of individual switches, see figure 6.14 (c).

6.4. SDN QoS through passive monitoring metrics

QoS based on passively measured metrics does not require imposing any additional packets in the network. The centralized SDN controller provides information that can be used to calculate other QoS metrics, such as the port utilization, data rates, and packet losses, see equations (6-1), (6-2), and (6-3). Where $S$ and $R$ are the count of sent and received bytes, $i$ and $j$ are port numbers, $t$ is the recording time.

\[
\text{Port Utilization} = \frac{(S_i^t - S_i^{t-\Delta t}) + (R_j^t - R_j^{t-\Delta t}) \times 8}{\text{Port Speed} \times \Delta t} \times 100 \quad (6-1)
\]

\[
\text{Data Rate} = \frac{(S_i^t - S_i^{t-\Delta t}) \times 8}{\text{Port Speed} \times \Delta t} \times 100 \quad (6-2)
\]

\[
\text{Packet losses} = \frac{(S_i^t - S_i^{t-\Delta t}) - (R_j^t - R_j^{t-\Delta t})}{\text{Port Speed} \times \Delta t} \times 100 \quad (6-3)
\]

Other experiments are conducted to test the efficiency of the proposed framework considering the path utilization as a metric for path selection [71]. The path utilization cost is calculated based on the port’s utilization, equation (6-1). At any given time, the utilization of all ports along the path is calculated, and the highest utilized port in a path represents the cost for that specific path. In simple words, the presented framework is fishing for the most unobstructed path for traffic forwarding.

Figure 6.15 shows the fat-tree topology during the experiment. Hosts h1-h8 are senders/receivers for the traffic under investigation, while hosts h9-h16 are used for generating fixed background traffic during the whole experiment. For any initiated communication, the “Event Notification
“listener” module receives a notification of the active pairs. It triggers the “Route-selection” module to calculate the best path based on the link utilization. All possible shortest paths are calculated, and the path with the least utilized cost is selected for forwarding. The chosen path is then sent to the “Rule preparation” module, for preparing rules and configuring switches.

6.4.1. Experimental setup and Configuration

For performance evaluation, experiments are set up using OpenDaylight “Nitrogen” release and “Mininet” with Mininet end hosts and D-ITG sender/receiver agents. The three considered forwarding modes are: (1) odl-l2switch built-in feature in the open-daylight, (2) Shortest-Path unweighted Dijkstra’s shortest-path algorithm through the proposed framework, and (3) QoS-Shortest-Path based on the path utilization also provided through the proposed framework.

For the tested traffic, we adopt two traffic scenarios. In both scenarios, the network is initiated with a “background traffic” generated by h9-h16, see figure 6.15. The hosts create a total of 6 UDP flows, sending fixed packets size of 512 bytes, and a constant packet rate of 3K pkt/s. The
"background traffic" is sustained running with other traffic under investigation. The investigated traffic is either "customized traffic" flows generating different types of applications, packet sizes, and packet rates, or a "real traffic" example captured from real internet traffic over a private network.

In the "customized traffic" scenario, the D-ITG hosts are configured to produce a total of 20 flows, including different application packets, namely, TCP, UDP, and VOIP packets. The generated packets’ sizes vary from 100 bytes to 1 KB, and the packet rates vary uniformly between 1K and 2K pkt/s. In the "real traffic" scenario, the D-ITG flows are generated based on a downloaded data file [72] that contains packets captured from real traffic in a busy private network. The downloaded data file “.pcap” is mapped to the D-ITG flow script using ditg_to_pcap [73]. The captured data file was analyzed, and the src/dst IP’s addresses were extracted and assigned to the Mininet hosts. It is worth noting that part of the flows was removed to fit in our limited resources. The "real traffic" scenario tested 686 flows that have varying durations between 1s and 38s, and different packet counts between 2 packets up to 2517 packets. For all traffic scenarios, the average of all traffic flows for all packet sizes, and application types were considered for plotting. The performance metrics were presented in terms of minimum delay, maximum delay, average delay, average jitter, and packet losses.

6.4.2. Experimental results

The recorded experimental results, for both scenarios, of the five-performance metrics, namely, average delay, minimum delay, maximum delay, average Jitter, and the packets dropped, are shown in figures 6.16-6.21. The minimum, maximum, and average delays of the investigated "customized traffic" and "background traffic" are shown in figure 6.16(a) and figure 6.16(b),
respectively. For both figure 6.16(a) and figure 6.16(b), the minimum, maximum, and average delay recorded better performance as forwarding mode changes from odl-l2switch to SP and QoS-SP modes. For both the investigated and “background traffic,” the enhancement in the average delays for the SP and QoS-SP over the odl-l2switch recorded 43% and 60%, respectively. Moreover, the standard deviation records a decrease of 54% and 70% for both SP and QoS-SP, respectively. As per figure 6.17(a) and figure 6.17 (b), using SP and QoS-SP modes outperform the odl-l2switch, in terms packets dropped. The recorded enhancement is 44% and 63% for the SP and QoS-SP, respectively. figure 6.18 (a) and (b) show a reduction in the average jitter by 44% and 62% on average when applying the SP and QoS-SP compared to odl-l2switch.

**Figure 6.16: Delays (traffic scenario “1”)**
The port utilization for these experiments using the monitoring module showed an overall average reduction in port utilization from 19.7% for the “odl-l2switch” to 13.2% and 12% for SP and QoS-SP, respectively. In other words, the utilization was reduced by 33% and 37% for the SP and QoS-SP, respectively. The maximum port utilization for both SP and QoS-SP was also reduced by around 2% on average.

Figure 6.17: Packet Drop (traffic scenario “1”)

Figure 6.18: Average Jitter (traffic scenario “1”)
Chapter 6: Experimental results for Proposed Framework

Considering the “real traffic” scenario, figures 6.19-6.21 show the delay, packet drops, and jitter for both “real traffic” and the “background traffic.” The results of the experiments again proved the superiority of applying the proposed framework in an SDN environment. For the investigated “real traffic,” figure 6.19(a) shows an enhancement in minimum, maximum, and average delay when using both SP and QoS-SP. The average delay recorded a reduction of 48% and 53% when applying the SP and QoS-SP, respectively. Figure 6.20 shows the packet drop in both the investigated “real traffic” and the “background traffic.” The “real traffic” shows nearly no packet losses for all forwarding modes, and the “background traffic” shows a reduction from 2.5% to 0% when applying either SP and QoS-SP forwarding mode. For the average jitter, figure 6.21(a) shows a reduction in the “real traffic” by nearly 23% on average for both the SP and QoS-SP. It is worth noting that the experiments record almost no change in the delay and jitter for the “background traffic” for all trials overall forwarding modes.

![Delay - "Real Traffic" scenario](image)

Figure 6.19: Delay (traffic scenario “2”)
6.5. Congestion Detection and Control

Congestion control is one of the critical factors enhancing QoS in a network. Congestion, in the context of networks, refers to a network state where a node/link holds so much data that is affecting the network QoS. Network congestion can be reduced through traffic monitoring, network segmentation, topology design, explicit congestion notification, TCP-settings reconfiguration, backpressure routing, and traffic prioritization. This section shows how the proposed framework monitors the network for congestion detection, then re-routing a subset of the flows passing through the congested ports. The proposed mechanism is currently customized for a Fat-tree SDN network. However, it can be modified easily to fit any other topology by specifying the key switches to be monitored.
The proposed framework monitors a subset of the network switches to detect congested parts, automatically reroutes subset of the traffic without interrupting the flows, and ensures the rerouted packets will not reposition the congestion in other parts of the network. The proposed mechanism reduces the maximum link utilization for the whole network and avoids congestion by applying an efficient re-routing mechanism. The congestion control module performs two main functions. First, the module selects the best candidate subset flow for rerouting. Then, it calculates the best candidate route based on the port utilization. Once a congested port is detected, with the SDN controller having a full view of the port status, the new calculated path cost should exclude the congested link, provided that a short path exists with a better cost. The newly calculated best-utilized path will be assigned as the new path between the source and the destination of the rerouted flow, as long as such rerouting will not transfer congestion to another port. The most significant possible passing flow, through the congested port, is nominated for rerouting without moving the congestion to any other link. The rerouting is based on selecting the shortest path that owns the least utilized cost, $C_p$. It is worth to be noted that, SDN controller, based on the port utilization information of the aggregation layer switches, can predict any congestion at any switch of the network eg., core, edge, and the aggregation layer switches. The monitoring overhead required for congestion detection is reduced by 60%.

6.5.1. Experimental setup and Configuration

In the given test environment, as a congested link is discovered, a subset of the traffic is rerouted on a separate calculated path that has the lowest utilization cost. The selected subset data flow will be rerouted provided that; no rerouted flow will transfer the congestion to another port in the network. To validate the effectiveness of the presented modules, experiments are set up using OpenDaylight and “Mininet.” An efficient fat-tree topology is used. The tested environment,
shown in figure 6.22, contains twenty 4-ports Open-Flow switches, holding up to 16 Linux based Mininet end hosts.

The tested traffic is generated using the Distributed Internet Traffic Generator (D-ITG) [66]. Half (50%) of the total hosts, housed in the topology, are involved in traffic (sending/receiving) with one host being selected per edge switch. As shown in figure 6.22, host $h1$ is the source of the traffic flows communicating with $h2$-$h8$ hosts. The calculated shortest path between host $h1$ and all other hosts will produce a congested node. The proposed congestion detection module oversees the ports in the aggregation layer with a predefined congestion threshold. As was recommended in the literature, experiments are set up with 70% and 80% thresholds for port utilization. Many
literature reviews recommended a 70% port utilization as a threshold level at which the port is experiencing congestion [58] [57]. Other publications considered 80% port utilization threshold for their experiments [74]. The tests consider averaging the results gained when setting 70% and 80% port utilization thresholds. Different flow transmission rates, e.g., 0.25K, 0.5K, 1k, 2k, 3K, 4K, 5K, and 8K packets per second, are considered. Various experiments are carried out, each with added TCP and UDP flows ranging from 0.6K pkt/s and up to 5.4K pkt/s (2K pkt/s on average).

In figure 6.22, the Congestion is detected at the aggregation switch a5 and the congested port #3 is selected for flow rerouting. The controller detects all flows passing through this port and selects the most significant flow based on the traffic size. The developed modules find out a new path with minimum capacity utilization between the end pairs of the selected flow. The flow traffic is rerouted, providing that the new cost of the calculated path will not exceed the cost of the congested port. Otherwise, the next candidate traffic flow is selected from the candidate list for rerouting. The path chosen for the rerouted traffic is installed in the switches, and the packets start moving through the new path. To validate the proposed mechanism, the shortest-path routing algorithm is tested with and without the existence of the proposed congestion control modules. Different traffic application types are investigated, namely, TCP, UDP, and VOIP. Different transmission rates are also considered for two sets of experiments with 70% and 80% thresholds for the link capacity. Different positions for communicating hosts are selected. The averages of all experimental results are recorded for calculating the overall performance metrics.

### 6.5.2. Experimental results

The recorded experimental results of the six-performance metrics, namely, *average delay*, *minimum delay*, *maximum delay*, *average jitter*, *throughput*, and *port utilization*, are shown in
figures 6.23-6.31. Using the proposed congestion control modules show their efficiency compared to using only the shortest path algorithm for route determination [75]. In sections 6.2, 6.3, and 6.4, the shortest path algorithm proved its efficiency over the built-in forwarding, “odl-l2switch”, in the OpenDaylight controller. Three communication protocols are considered for the comparison TCP, UDP, and VOIP. The average communication traffic per experiment ranges from 0.6 K to 5.4K pkt/s. The shortest path algorithm is tested with and without the proposed congestion control mechanism for all performance metrics.

Figures 6.23-6.26, display the percentages of improvements in the delay, jitter, and throughput for different traffic rates. For all metrics, TCP shows the best improvement percentage over UDP and VOIP. In figure 6.23 and figure 6.24, UDP and VOIP show a noticeable improvement in delay when low traffic rates are considered. As the traffic rate increases (e.g., higher than 3.2K traffic rates), TCP shows a better performance. Meanwhile, UDP and VOIP show low or no enhancement at such rates.

![Maximum delay (TCP, UDP, VOIP)](image)

Figure 6.23: Max Delay improved percentage for TCP, UDP, and VOIP

Chapter 6: Experimental results for Proposed Framework
Chapter 6: Experimental results for Proposed Framework

In figure 6.25, TCP shows a big improvement in the average throughput at high traffic rates while other protocols (UDP and VOIP) show no improvements along with all traffic rates. In figure
Chapter 6: Experimental results for Proposed Framework

6.26, UDP and VOIP show a comparable improvement in jitter with TCP during low traffic rates (under 3.2K). The latter (TCP) recovered its superiority as the traffic rates increased (over 3.2K).
Figure 6.27 shows that with the proposed congestion control mechanism TCP recorded the most considerable improvement, VOIP comes second, then UDP. In summary, TCP shows improvements by 22.4% in the average delay, 18.6% for the maximum delay, 15.3% in jitter, and 21.3% in throughput.

Figure 6.28: 4-port switch Utilization of each level in a Fat-tree topology (ODL)

Traffic ports utilization is shown in figures 6.28-6.31. Figure 6.28 shows the average reduction in port utilization in the core, aggregation, and edge switches. The presented usage is the utilization of the shortest-path without congestion control compared when using the proposed congestion control mechanism with thresholds 80% and 70%. At the core switch level, when averaging the
maximum port utilization, it is reduced from 22.1MBytes to 19.2MBytes and 17.2MBytes with congestion control thresholds 80% and 70%, respectively. Thus 13% and 22% improvements are achieved. At the aggregation layer, the maximum port utilization is also reduced from 25.3MBytes to 22.6MBytes and 20.4MBytes with congestion control thresholds 80% and 70%, respectively, and Thus 11% and 19% improvements are achieved. Finally, when it comes to the edge switches, no much improvement in the port utilization is realized when new unutilized ports and switches start participating in the traffic forwarding (reduced with around 3%). It is worth to be noted that ports at this level are directly connected to the source and the destination hosts and no further routes could be possible. The results mentioned above show that using the proposed congestion control mechanism improves the maximum port utilization percentage in the core, aggregation, and edge switches by 18%, 15%, and 3%, respectively.

Figures 6.29-6.31 show the timely reduced maximum port utilization for all aggregation and core switches. As shown in figure 6.29, the maximum port utilization, using a short path algorithm only, remains at its maximum value during the whole experiment. On the other hand, with the presented congestion control mechanism, the maximum port utilization is reduced just after detecting the first utilization peak, as shown in figure 6.30 and figure 6.31. The congestion control mechanism rebalances the traffic load

![Figure 6.29: Shortest-path port utilization for an experiment](image-url)
by engaging new switches and ports in the forwarding process, thus reducing the overall maximum port utilization and pulling down the probability of future congested ports.

Figure 6.30: Congestion Control (80%) port utilization for an experiment

Figure 6.31: Congestion Control (70%) port utilization for an experiment
Chapter 7

Conclusion and Future work

7.1. Conclusion

In this work, we investigated how an SDN can enhance performance by solving many of the traditional network issues. Experimental results revealed that SDNs are a promising foundation for more scalable, faster converging, and better-performing networks. Firstly, the potential of using SDNs in large scale data and communication networks was investigated. Two widely used topologies in today’s large-scale networks, namely, Fat-Tree and BCube, were the seeds of the research. Scaling in terms of the number of hosts and switches in a given topology and degradation in its performance was proven to be related and changed shoulder to shoulder. Also, the experimental results showed that small scale BCube-like topology has a better performance in terms of packets delay, jitter, and dropped packets. This behavior continued, in some cases, when the topology scaled up to medium size. On the contrary, Fat-tree topology proved to be able to maintain a sustainable performance as the network scaled up to a large extent. Having a better performance, the Fat-tree was the adopted topology in all subsequent experiments and investigations.

On the track, the support of an SDN for big-data applications was investigated, as the Fat-tree network topology scales up to extreme sizes. The traffic behavior for “Hadoop,” big-data application, was tested using the normal layer two (L2) packet switching and through the OpenDaylight “Boron” release (ODL) controller. Using an SDN controller to support Big-Data
applications showed a better performance in terms of execution time and throughput for almost all scales of the tested topology. Moderate improvement was noticed with a small-scale network, but a significant increase in performance was recorded as the network scaled up. A single controller in OpenDaylight proved to be eligible to maintain a sustainable performance as the network scaled up. For very large-scale networks, a more powerful controller node, with an added clustered feature, was considered, to sustain the improvements in network performance.

Next, a framework for providing QoS for different applications was proposed. The framework extended the SDN capability and centralized view for monitoring the network and enhanced the forwarding decisions. It provided QoS through (1) classification of application type, (2) metrics-based routing, or (3) Congestion detection and control. The framework modules were implemented in python language that used the REST API to communicate with the controller. The framework provided the following modules: (1) an Event Notification listener for detecting any communicating nodes, (2) A metric-based Route-calculation, (3) A Flow-preparation for programming flows on switches, (4) A Monitoring module for collecting, storing and analyzing statistics from the controller’s operational data store, and (5) a congestion detection and control module to relax congested links, reroute traffic and equalize traffic load.

Moreover, the proposed framework offered a scheme to dynamically detect the initiated flows and assign suitable routes based on the required QoS. Traffic flows were automatically rerouted to a lower-cost path based on generated events of newly discovered flows, a detected congested link, or preconfigured policy rules. The proposed technique redistributed the traffic considering underutilized links and thus enhancing the packet delay, jitter, packet drops, and port utilization. It made the best use of network throughput and guaranteed the requested QoS level for applications. The selected QoS routes were based on either the link delays measured through
probe agents or, the link utilization calculated through the SDN provided statistics. Since the traffic flow requirements changed over time, the mentioned scheme dynamically detected the initiated flows or congested links and assigned the needed routes based on the required QoS.

Experimental tests were conducted to prove the efficiency of the proposed framework. The network performance for three types of applications, namely, TCP, UDP, and VOIP, was evaluated in terms of improved delay, jitter, throughput, port utilization, and packet loss. Firstly, some experiments were set up to test the traffic based on the application type. The “Route-calculation” module rerouted a subset of the traffic on a separate link. This reduced the load over the original link and provided better performance for all running applications. We set apart the traffic flows over the available paths based on the application type. Traffic flows based on application type were injected in the network under different modes of operation, namely, odl-l2switch, single-path, and multipath. The experimental results showed the superiority of using the presented modules over the odl-l2switch in terms of delay, jitter, and packet dropped, with the best-recorded performance for the multipath operation.

Secondly, the framework was tested for providing QoS metrics-based route determination. The “Route-calculation” was based on either the end-to-end delay or path utilization. Different traffic forwarding modes were considered, namely, odl-l2switch, Shortest-path, and QoS-Shortest-path. The performance metrics for various applications and different traffic scenarios were presented in terms of delay, jitter, and packets dropped. Once more, the proposed framework showed its superiority compared to the standalone controller's overall performance metrics. Using the proposed framework modules, active/passive experimental setup, for metrics-based forwarding, reduced the total packet delay by 54% on average. The packets drop was reduced by 51% on
average, and the average jitter was decreased by 32% for all application types. Moreover, considering port utilization showed an overall reduction of 22% on average.

Finally, congestion detection and control were added features for the framework. The proposed congestion control mechanism was in charge of monitoring port utilization for detecting congested links. Two components were implemented, namely, congestion detection and congestion control. The congestion detection component continuously monitored the aggregation layer of the fat-tree topology for congested ports (exceeding a certain threshold). Upon detecting a congested port, the congestion control component was triggered to search for new routes for the subset of the flows passing through the congested port. The re-routing selection was a two steps process. First, the best candidate subset flow for rerouting was selected. Then, the best candidate route was nominated. To immune the rerouting process, the selected subset data flow to be rerouted over the new routes would not congest these routes due to the newly added traffic flows.

Experiments were set up in the Mininet environment with the ODL controller and D-ITG for traffic generation. Once more, three communication protocols were engaged for verification, namely, TCP, UDP, and VOIP. Various traffic rates, types, and thresholds were also considered. The recorded results for six-performance metrics, namely, average delay, minimum delay, maximum delay, average Jitter, throughput, and port utilization revealed the efficiency of the proposed congestion control mechanism when using the shortest path algorithm for route determination. TCP application shows a noticeable improvement in the selected metrics, over all the traffic rates. With TCP, the average delay, maximum delay, jitter, and throughput recorded an improvement by 22.4%, 18.6%, 15.3%, and 21.3%, respectively. Moreover, using the proposed modules, a reduction was recorded in the maximum port utilization in the core, aggregation, and edge switches by 18%, 15%, and 3%, respectively.
7.2. Future work

For the future work, we are planning to apply a more sophisticated algorithm for route selection that considers a combination of multiple QoS metrics as route costs. Based on the application type, different applications will be considered for diverse QoS metrics path selection. Moreover, the congestion control mechanism can use a different algorithm for selecting the best fitting flow for rerouting. Also, a load balancing scheme will be considered to reduce the load on the single chosen path.

Then, as wireless networks are rapidly growing, it is essential to investigate how accurate the network information will be acquired and the efficiency of controlling such networks, especially when different hosts are interacting together. The presented framework will be extended to cope with the new challenges offered by wireless networks. Finally, security and scalability issues to be considered; the proposed framework will be enhanced to support larger-scale networks and efficiently manage a large amount of received information from the controller. Some added security modules will be considered to address some of the security challenges in today's SDN networks.
References


[40] Y. Zhao, J. Zhang, H. Yang, and X. Yu, “Data Center Optical Networks ( DCON ) with OpenFlow based Software Defined Networking ( SDN ),” in 8th International Conference on Communications and Networking in China (CHINACOM), 2013, pp. 771–775.


