# Experimental Categorization of SDN Controllers using Open-Source Benchmarking Tools

By

**Ghulam Bahoo**

**Fall-2020-MS-IT 00000329486 SEECS**

Supervisor

**Dr Salman Ghafoor**

**Department of Computing**

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Information Technology (MS IT)

In

School of Electrical Engineering & Computer Science (SEECS) ,

National University of Sciences and Technology (NUST),

Islamabad, Pakistan.

(December 2023)

# Approval

It is certified that the contents and form of the thesis entitled "Experimental categorization of SDN controllers using open-source benchmarking tools" submitted by Ghulam Bahoo have been found satisfactory for the requirement of the degree

Advisor : Dr. Salman Abdul Ghafoor

Signature: [illegible]

Date: 04-Dec-2023

Committee Member 1:Dr. Ahmad Salman

Signature: [illegible]

06-Dec-2023

Committee Member 2:Dr. Rizwan Ahmad

Signature: [illegible]

Date: 04-Dec-2023

# Thesis Acceptance Certificate

Certified that final copy of MS/MPhil thesis entitled "Experimental categorization of SDN controllers using open-source benchmarking tools" written by Ghulam Bahoo, (Registration No 00000329486), of SEECS has been vetted by the undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/M Phil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature: [illegible]

Name of Advisor: Dr. Salman Abdul Ghafoor

Date: 04-Dec-2023

HoD/Associate Dean: [illegible]

Date: 03-Jan-2024

Signature (Dean/Principal): ____________

Date: ____________

# Dedication

This thesis is dedicated to all the deserving children who do not have access to quality education especially young girls.

# Certificate of Originality

I hereby declare that this submission titled "Experimental categorization of SDN controllers using open-source benchmarking tools" is my own work. To the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics, which has been acknowledged. I also verified the originality of contents through plagiarism software.

Student Name:Ghulam Bahoo

Student Signature: [illegible]

# Acknowledgments

I wish to express my profound gratitude to **Allah Almighty** for His continuous guidance and blessings. I owe a debt of gratitude to my **parents** whose unwavering support and fervent prayers have been the cornerstone of my achievements. I extend sincere appreciation to my academic supervisor **Dr Salman Abdul Ghafoor** and the esteemed members of the Guidance Committee for their invaluable guidance and scholarly input, which have significantly enriched the quality of my research. Their expertise and commitment to academic excellence have been instrumental in shaping the outcome of this work. Lastly, I acknowledge the supportive role of my friend and colleague, **Nimra Noreen**, whose companionship and shared dedication to scholarly pursuits have added a meaningful dimension to my academic journey. Each of these entities has played a pivotal role in my academic success, and for their contributions, I am sincerely thankful.

**Ghulam Bahoo**

# Abstract

Software Defined Networking SDN has increased network programability and administrative ease. SDN provides centralised and fine-grained network control; it has now become a trend in industry and is being used in different fields of networking. Controllers are crucial to the network's stability and scalability in SDN, and their performance is critical. This research looks at three SDN controllers: RYU, POX and NOX. Quality of Service QoS measures such as Flow Setup Latency, Initial Ping Delay , Round Trip Time , Throughput, and TCP and UDP Bandwidth are used to evaluate the controllers' performance. Different sized networks are emulated using the Mininet SDN simulator by adjusting the number of switches in linear and tree topologies and hosts in a single topology. In compared to POX and NOX, my research demonstrates that RYU's performance is quite consistent and exhibits little variation as the number of network devices and network traffic increases.

# Contents

# List of Figures

# List of Abbreviations and Symbols

## Abbreviations

| | |
|---|---|
| **SDN** | Software Defined Networks |
| **TCP** | Transmission Control Protocol |
| **RTT** | Round-trip time |
| **DCN** | Data Center Network |
| **Cbnech** | Controller Benchmarking |
| **UDP** | User Datagram Protocol |
| **IPD** | Initial Ping Delay |
| **API** | Application Programable Interface |
| **CSV** | comma-separated values |
| **OFDP** | OpenFlow Discovery Protocol |
| **LLDP** | Link Layer Discovery Protocol |
| **IETF** | Interfnet Engineering Task Force |
| **NIC** | Network Interface Card |
| **QoS** | Quality of Service |

CHAPTER 1

# Introduction and Motivation

Software Defined Networks (SDN) is an approach to networking that separates the network's control plane and data plane operations in order to make networks more flexible, agile, and programmable. Control (deciding how to manage traffic) and data forwarding (real data packet movement) occur within the same device in classical networking.

SDN decouples these activities, concentrating network intelligence and control in software-based controllers or applications, while the data plane continues to deliver traffic in accordance with the controller's instructions. This separation enables administrators to govern network behaviour through software, making it easier to optimise and manage traffic flow, enforce policies, and adapt to changing network conditions. There are many SDN open-source controllers that are available:

- NOX
- POX
- RYU
- Floodlight
- OpenDayLight
- ONOS

The **Data Plane** also known as the forwarding plane, is in charge of forwarding data packets as they pass through a network device. It evaluates the headers of incoming packets, decides where these packets should be forwarded based on their destination addresses, and then transmits them over the relevant output interfaces. The data plane, in essence, moves data packets according to predetermined rules and configurations without engaging in complex decision-making processes.

**Control Plane** The network device's intelligence resides in the control plane. It is in charge of deciding how data traffic should be routed through the device. This plane is in charge of running routing protocols, obtaining network topology information, exchanging routing information with other devices, determining best pathways for data transfer, and updating forwarding tables utilised by the data plane. The control plane establishes the rules, policies, and configurations that control the data plane's behaviour. Elmoslemany, Mohamed M et al. [1] compared

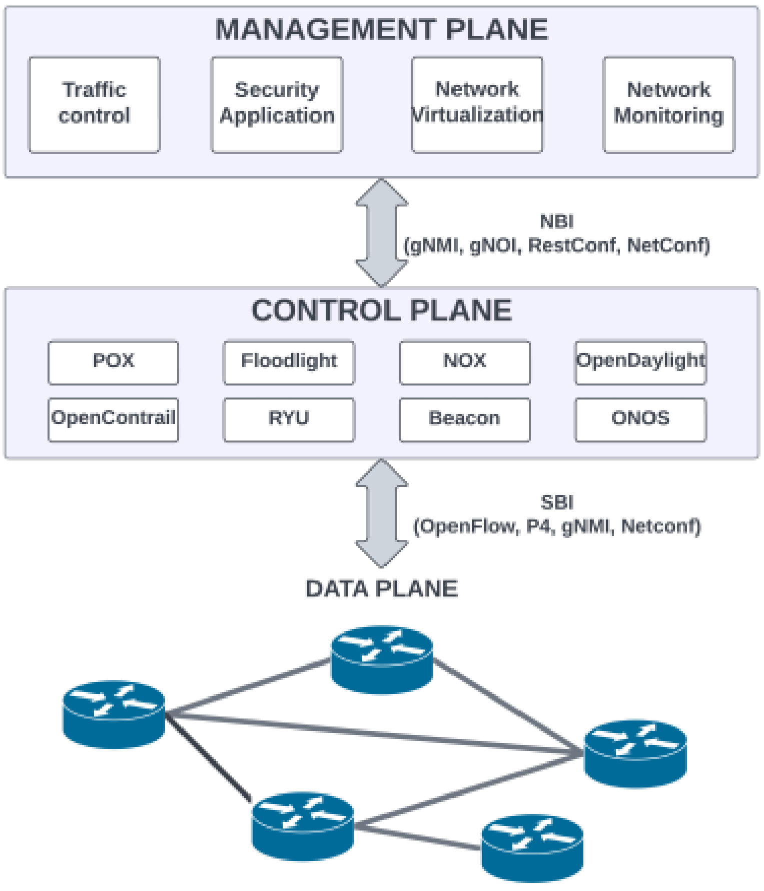


**Figure 1.1: SDN Architecture**

using Cbench, ONOS, ODL, POX, and RYU throughput and latency. Ali, J. et al. [2] POX and

RYU were tested in terms of Transmission Control Protocol (TCP) bandwidth and Round-trip time (RTT) for single, linear, tree, and Data Center Network (DCN) topologies with a set number of hosts and switches. Danijel et al.'s [3] study of POX and RYU's RTT and total latency [4] was carried out using tree topologies with 2, 4 and 8 switches. The authors of [5] compared RTT and throughput for POX and Floodlight using built-in mininet topologies. Floodlight also surpasses RYU in terms of throughput and latency, according to [6]. When ONOS, Floodlight, and RYU were tested against open commercially available switches, ONOS fared the best in the flow setup test, while RYU discovered the topology in the quickest amount of time, according to [7]. The results reported in [8] show that OpenDaylight outperforms ONOS as workload increases among distributed controllers, whereas RYU outperforms ONOS among centralised controllers for large-scale network management. According to the data in [9], when utilising the Cbench tool, the Beacon controller outperforms POX, RYU, NOX and Floodlight. The authors of [10] thoroughly investigated numerous OpenFlow controllers in a detailed analysis. They used benchmarking tools such as Cbench, PktBlaster, and OFNet to assess the performance of nine controllers. The study found that distributed, multi-threaded controllers outperform centralised, single-threaded controllers, but at the penalty of using more physical resources.

In this study, I thoroughly examined the performance of RYU, NOX and POX for a variety of SDN network topologies using Mininet. To evaluate the performance of these controllers. I evaluated seven metrics: throughput, flow setup latency, round-trip time , inter-packet delay, jitter, and TCP/UDP bandwidth. CBench is used to evaluate SDN controller flow setup latency and throughput. Ping, Iperf, and Mininet are used to investigate additional performance characteristics. SHell scripts were used to automate the entire experimenting process, which improved the performance of the controller. To the best of my knowledge, no other work has taken into account as many topologies, performance parameters, and scripting techniques as I did in this work.

## 1.1 Tools for Simulation and Benchmarking

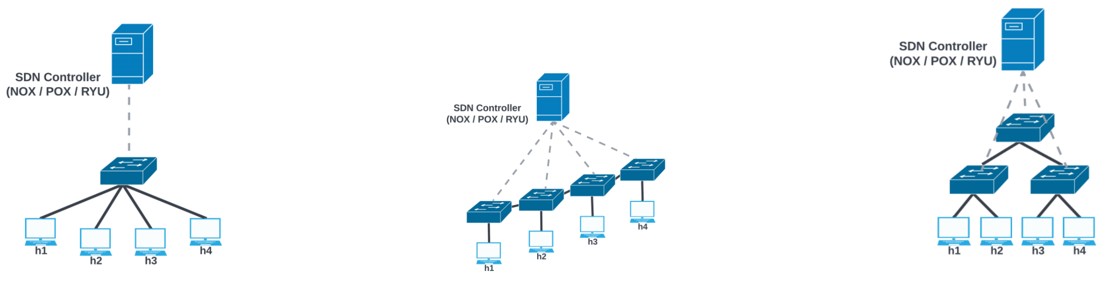
**Mininet** [11] Mininet is an open-source emulator that allows you to create virtual networks within a single computer system. Its principal function is to allow users to simulate and construct software-defined networks (SDNs) without requiring actual hardware. This platform enables the creation of virtualized network topologies, switches, hosts, and controllers, allowing for experimentation, testing, and software development in a controlled, simulated network environment.

**Controller Benchmarking (Cbnech)** It is a specialised benchmarking tool designed to assess the performance of Software-Defined Networking (SDN) controllers. This tool is intended to assess how well an SDN controller manages OpenFlow messages and network management duties under varying workloads and situations. Its major goal is to stress-test the controller by creating a large number of OpenFlow messages and simulating various network traffic scenarios in order to evaluate the controller's responsiveness and efficiency.

**Iperf** IPerf is an open-source command-line application that generates Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams between two endpoints to measure network performance. It aids in determining the maximum attainable bandwidth on IP networks and identifying potential network-related issues through the use of measurements such as throughput, packet loss, delay, and jitter.

## 1.2 Performance Parameters

1. **Flow Setup Latency [10]**: The time it takes to establish and configure a communication link or flow between two network devices within a network is referred to as flow setup latency. It specifically refers to the time required to set up the necessary rules, policies, or configurations in network devices (such as switches or routers) to handle a certain flow of traffic in the context of Software-Defined Networking (SDN) or OpenFlow-based networks.



**(a) Single** **(b) Linear** **(c) Tree**

**Figure 1.2: (a) Star Topology,(b)Linear Topology, (c)Tree Topology**

2. **Throughput [1]**: Throughput is the rate at which data is successfully transmitted or processed over a communication channel or network in a given amount of time. It quantifies

the quantity of data effectively transported between two places in a network or within a system. Depending on the amount of data transfer, throughput in networking or data communication is often stated in bits per second (bps).

3. **TCP Bandwidth [2]**: The maximum data transfer rate possible over a Transmission Control Protocol (TCP) connection between two endpoints is referred to as Transmission Control Protocol (TCP) bandwidth. It indicates the amount of data that can be transmitted across the network per unit of time using Transmission Control Protocol (TCP), taking into account network conditions, latency, packet loss, and protocol characteristics.

4. **UDP Bandwidth**: The maximum data transfer rate possible over a User Datagram Protocol (UDP) connection between two endpoints is referred to as UDP bandwidth. Unlike TCP, UDP is a connectionless and unreliable protocol that lacks flow control, error correction, and congestion control techniques.

5. **Initial Ping Delay (IPD)**: TThe term "initial ping delay" refers to how long it takes for an initial ICMP (Internet Control Message Protocol) echo request, sometimes known as a "ping," to reach its destination and obtain a response.

6. **Round Trip Time (RTT) [2]**: Round-trip time (RTT) is the amount of time it takes for a signal or packet to go from a source to a destination and then back to the source. Round-trip time (RTT) is a networking term that describes the time it takes for a packet to travel from a sender to a receiver and then for an acknowledgment to travel back from the receiver to the sender.

7. **Jitter**: Jitter is the variance in the latency of received packets in a network, particularly in packet-switched networks such as the internet. It depicts the difference in packet arrival timings as they move through the network.

### 1.2.1 Custom Topologies

I developed three distinct topologies using Mininet's Python API. Each topology is described in depth below:

1. **Single Topology (T1)**: In the T-1 topology, I changed the number of hosts devices linked to a single switch from 2 to 64. Figure 1.2(a). depicts a single topology with four hosts.

2. **Linear Topology (T-2)**: I changed the number of linearly connected switches within the T-2 topology, increasing it from 2 to 1024. In this topology, there are two hosts—one connected to the initial switch and the other linked to the final switch. A linear topology with 4 switches is shown in Figure 1.2(b).

3. **Tree Topology (T-3)**: I changed the depth of the binary tree from 1 to 5 in T-3 topology. A tree topology with depth 1 is shown in Figure 1.2(c)

### 1.2.2 Cbench Topology

Using bash scripts, I emulated Cbench topologies , which generated a set of topologies with two hosts per switch and a total number of switches ranging from 2 to 1024, which were directly connected to a distant SDN controller. T-4 topology is what I name these topologies.

## 1.3 Problem Statement and Contribution

### 1.3.1 Problem Statement

Based on the researcher's experience there are still some research gaps such as:

- There is a need to analyze the response of several renowned SDN controllers when subjected to extreme challenging conditions involving an exponential growth in the number of host devices. The primary objective is to identify each controller's behavior regarding its maximum capacity in handling host devices under these varying topological setups.
- No scripting automation techniques are currently utilized in existing solutions.
- The current research in the domain of SDN controllers lacks a systematic classification based on specific network scenarios.
- Study and testing of different performance parameters has not been done adequately . For example Throughput, Latency, TCP/ UDP Bandwidth, Packet Loss etc.

### 1.3.2 Proposed Solution

**Topology Generation**

A Python script will be created to leverage the Mininet API for dynamically generating virtual topologies.

- **Topologies Design**: Custom mininet topology scripts will be used. This simulates common SDN deployment with edge switches connecting user devices.
- **Host Scaling**: The initial number of hosts will be set to 2 connected to the switch. For each iteration, the number of hosts will be doubled - 4, 8, 16 and so on. The maximum number of hosts will be 2048 to stress test at massive scales.
- **Topology Object Creation**: A Mininet topology object will be instantiated in each iteration. The switch will be added along with the required number of host objects.
- **Link Configuration**: Links will be created to interconnect each host to the central switch. TCP queues, link capacities etc. can be customized if needed.
- **Application Programable Interface (API) Calls**: Mininet API functions like addHost(), addSwitch(),addLink() will be leveraged. net.addController() will attach the controller. net.start() will initialize the emulated network.
-

This systematic doubling of hosts using Mininet APIs allows automatated generation of thousands of unique topologies ranging from small to extremely large scales in an structured manner.

**Automation using Shell Scripting**

A bash shell script will be developed to automate the entire testing process from start to end. The key steps involved are:

- **Custom Topology** The script will take the base number of hosts and an increment factor as input. It will call the Python Mininet script in a loop, each iteration doubling the number of hosts to generate the topologies.
- **Controller Connection** The IP/port of the controller to test will be passed as arguments to the script. It will connect each topology sequentially to the controller using the Python API.
- **Performance Testing** Cbench, Iperf and OFCProbe tests will be launched in parallel processes for each topology. The script will monitor and wait for all child processes to complete before proceeding.

- **Results Logging** Output from all tools will be consolidated and important data fields extracted. Results will be written to a JSON file with metadata liketopology details, test timestamps etc.
- **Report Generation** Statistics will be computed from the CSV file and different visual plots/charts will be generated for analysis. Summary logs and reports help compare performance across tests.

This automation approach eliminates human errors and ensures consistent, repeatable execution of thousands of test cases. The comma-separated values (CSV) output enables flexible post-processing of large results datasets.

## 1.4 Aim and Objectives

### 1.4.1 Aim

The key objective of this research is to implement an extensive and in-depth analysis of the performance of SDN controllers, with a focus on RYU, POX, ONOS, and NOX. The purpose of this evaluation is to provide a thorough overview of the controllers' capabilities over a wide range of important performance characteristics. Some of these metrics are throughput, latency, TCP/UDP bandwidth, jitter, and initial ping delay IPD.

The primary goal of this research is to evaluate how the performance of these SDN controllers changes when the network increases in terms of the number of hosts and switches connected to it. This scalability study tries to shed light on how responsive and adaptive each controller is to changing network conditions by analysing each controller's maximum capacity while taking into account real-world limitations and performance constraints.

This study will contribute to a better understanding of how these controllers behave in different real-world circumstances by examining network configurations such as tree, linear and single topologies along with number of parameters which have been underrepresented in earlier studies.

### 1.4.2 Objectives

The key Objectives of this study are:

1. **Performance Evaluation**: To completely analyze the performance of NOX, POX, and RYU SDN controllers, with a focus on performance characteristics such as throughput and latency.

2. **Scalability Testing**: To determine each SDN controller's maximum capacity in terms of the number of linked hosts and switches before they become unresponsive or suffer severe performance deterioration.

3. **Variation in Network Topologie**s: To evaluate how SDN controller performance differs when subjected to varied network topologies, such as tree, linear, and single configurations. This investigation of several topologies will provide insights into the adaptability of the controllers.

4. **Identification of Performance Limitations**: Identify and describe each SDN controller's restrictions and constraints when dealing with growing network size and complexity, providing to a better knowledge of their practical capabilities.

## 1.5 Contributions

This study makes an important scholarly contribution to topic of Software-Defined-Networking. It includes a thorough examination of multiple well-known SDN controllers, including RYU, POX, and NOX. Through rigorous performance evaluation in a variety of network scenarios, the study gives an in-depth understanding of these controllers' capabilities. It delves into their practical limits, response, and scalability, all of which are critical details for network administrators and engineers. This study adds to our understanding of these controllers' real-world performance and offers important insights by analysing various network topologies such as tree, linear, single, and star configurations. Furthermore, this study fills current research gaps and improves theoretical and practical understanding of SDN. It provides researchers and network experts with useful information to help them select and configure SDN controllers to satisfy specific network requirements. The ultimate goals here are to improve network performance, resource efficiency, and general network administration using SDN technology. This study supports the practical implementation of results in the dynamic field of network technology and actively contributes to the ongoing discussion in academia.

## 1.6 Limitations

While the key objective of this study is to properly analyze the performance of multiple SDN controllers, including NOX, POX, and RYU, there are certain inherent constraints to consider. Although the Mininet simulation environment is valuable, it may not accurately replicate the complexities of real-world networks, and resource constraints may restrict the size of network situations. Furthermore, interactions in multiple network scenarios may be overlooked by the study, which focuses solely on individual controller performance. Despite their diversity, the chosen network topologies are still somewhat simplistic, and their lack of dynamic traffic patterns may limit their application in real-world scenarios. Changes made to controllers over time and differences in their settings may have an impact on performance. Finally, while outside the scope of the study, external factors such as hardware limits or network maintenance can have an impact on outcomes. These limitations give important background for interpreting the research findings and identify potential future research initiatives.

## 1.7 Thesis Structure

**Chapter 1: Introduction**

The first chapter is an introduction to the research, providing a thorough description of the research problem, its significance, and the stated aims and objectives.It lays the groundwork for the subsequent chapters and provides the study with a contextual framework.

**Chapter 2: Literature Review**

The second chapter conducts an in-depth examination of existing literature on SDN technology, SDN controllers, and relevant performance evaluation metrics. This chapter seeks to identify gaps in current research and to justify the study's necessity.

**Chapter 3: Methodology**

The third chapter covers the research methodology, including the research design, data gathering methods, and performance evaluation tools. It provides information about the network topologies, variables, and experimental setting employed in the study.

**Chapter 4: Results**

The fourth chapter demonstrates the results of the selected SDN controllers' performance evaluations across various network scenarios. Data, charts, and graphs are used to visually represent the research findings.

**Chapter 5: Discussion**

The fifth chapter is devoted to the analysis the results collected in the previous chapter. It assesses the implications of the results in light of the research objectives and prior literature, while also highlighting the study's flaws.

**Chapter 6: Conclusion**

The final chapter provides a thorough summary of the study, restating the objectives and significant findings. It emphasises the research's significance and finishes with recommendations for further research or practical applications in the field of SDN.

CHAPTER 2

# Literature Review

This research provides a thorough investigation of two commonly used controller implementations, Ryu and POX, including a feature-based comparison as well as a performance evaluation. This study examines how they perform in terms of throughput and latency across different network topologies, such as Simple-Tree-Based, Fat-Tree Based, and regular IP networks. Their findings show that the performance of Ryu and POX controllers is affected by a number of factors, such as the controller's hardware configuration, control algorithm settings, the underlying network infrastructure, OpenFlow switches, the number of connected hosts, threading configuration, and others. In the majority of our simulation trials, they find that when the OpenFlow protocol is enabled, both Ryu and POX controllers consistently beat traditional IP networks in term of latency and throughput. Furthermore, their data show distinctions between the two controllers. POX offers high throughput performance, excelling in particular at promptly completing requests in environments with complicated SDN traffic loads and increasing number of OpenFlow switches. Ryu, on the other hand, excels in latency, making it a preferable choice for delay-sensitive SDN applications and less sophisticated SDN network settings. [3]

The controller is an crucial part of the (SDN) framework, with the capacity to either strengthen or weaken SDN systems. Because SDN controllers can be constructed using a number of open-source and proprietary technologies, the controller used has a significant impact on total output. As a result, there is an urgent need to conduct a full examination and comparison of existing SDN controllers in both the commercial and research realms.

This research compares the performance of two prominent open-source SDN controllers, Open Network Operating System (ONOS) and OpenDaylight (ODL). They install the most recent stable versions, ODL-Nitrogen and ONOS-Nightingale, in a virtual test environment called

Mininet.Within this controlled environment, they rigorously evaluate several critical performance measures, such as IPD, RTT, and TCP capacity i.e. throughput, all of which vary depending on the network architecture deployed in Mininet.

The following conclusions were drawn from the experimental results: When compared to ODL, ONOS has better latency performance. In terms of flow-setup delay, however, ODL exceeds ONOS. The analysis of jitter shows that ONOS is more relieable, resulting in a more reliable network connection. ONOS outperforms ODL in terms of processing power based on measured TCP bandwidth using iperf. In conclusion, this comprehensive investigation reveals that, in their specific experimental environment, ONOS is a more resilient alternative than ODL. [12]

The goal of this article is to thoroughly evaluate ten main SDN controllers against a variety of critical criteria, followed by an assessment of their scalability and performance across various network topologies using Mininet an SDN network simulator. The following are the research's main contributions:

1. A complete list of important characteristics and capabilities that every SDN controller should have has been compiled.
2. Python-based controllers may struggle to achieve high performance and low latency in big networks, whereas Java and C-based controllers display greater scalability and performance.
3. The Multi-factors Decision Making technique is used to decide on the best controller based on many factors.

In essence, this research strives to provide a holistic assessment of SDN controllers, aiding stakeholders in selecting the most suitable controller based on their specific needs and priorities.[13]

The controller is the brain of every SDN network. As a result of the shift from traditional networks to SDN, several controllers have emerged, including Beacon, Floodlight, RYU, OpenDayLight, ONOS, NOX, and POX. Because of the wide range of SDN applications and controller options, selecting the optimal controller has become a situation-specific decision. As a result, the focus of this research is on determining the impact of various SDN controllers on SDN QoS performance. POX and RYU controllers were evaluated, with TCP, UDP, and ICMP traffic utilised to study their performance in terms of QoS parameters such as Throughput, Round-Trip Time, and Jitter. The RYU controller outperformed the POX controller, according to the results. In terms of average throughput, the RYU controller outscored the POX controller by 1.24 percent to 5.35 percent. Furthermore, when compared to the POX controller, the RYU controller had 0.5 to 1 ms less latency and around 0.02 ms less jitter.[14]

The paper presents the creation of an SDN architecture that analyses network traffic using the open-source RYU SDN controller. This project's purpose is to evaluate the performance of a customised network architecture within the SDN framework, with a focus on node-to-node performance parameters such as , throughput, bandwidth and RTT, among others. The results show unequivocally that the proposed custom network topology outperforms the default network configuration widely used in SDN.By utilising the open-source RYU controller for network traffic analysis, this study adds to the enhancement of performance measurements in traffic routing inside the SDN environment. This has far-reaching implications for the future, as the network may now be used to support a wide range of high-end applications. The primary purpose of this study is to make traffic analysis easier by analysing the performance of the RYU controller within the context of SDN. This assessment aims to improve resource utilisation, network performance, data traffic management, reduce costs associated with existing solutions, and encourage field innovation.[15]

The present study aims to conduct an in-depth examination of the complex query mechanism, infrastructure, contributing elements, and numerous challenges encountered during the topology finding process. This research also focuses on recent studies that have successfully addressed and improved on these issues. It also invites open debate on these topics while identifying future research directions. Topology finding services are a crucial challenge under discussion in this work. The SDN controller is responsible of maintaining the network topology up to date in the world of Software-Defined Networking (SDN). The OpenFlow Discovery Protocol (OFDP) is used to find the connections linking SDN-Switches in the data plane. Nonetheless, this technique's performance has significant limitations, particularly in large and dynamic network situations. Furthermore, application layer applications rely on this underlying topology, emphasising the importance of addressing these constraints. This article examines the performance constraints related with the link discovery protocol used in SDN networks, namely OFDP. This method is intended to reduce stress on both components and, as a result, learning time. Furthermore, identifying the right trade-off between SDN-Controller overhead and learning time becomes crucial for proposals that determine the topology on a regular basis. This trade-off should be tailored to the specific environment and topological change rate. For example, in situations with low and modest topological change rates, such as enterprise networks, the discovery interval may be extended. In contrast, it is best to keep the discovery interval as short as possible in dynamic and essential scenarios such as data centres and transportation networks.[16]

The document evaluates the OpenDaylight (ODL) and ONOS SDN controllers' performance for network topology identification and updating in response to changes. These are critical functions for the SDN controller to do in order to maintain an accurate global view of the network topology. The OpenFlow Discovery Protocol (OFDP), which is based on the Link Layer Discovery Protocol (LLDP), is commonly used for topology discovery. OFDP works by having the controller deliver LLDP packets to switches via Packet OUT messages, and the switches respond with the received LLDP packets via Packet IN messages. This enables the controller to detect switch links in both directions. The topology discovery time was measured in the experiments by recording the timestamp of the first Packet OUT and last Packet IN for each topology discovery cycle. Iperf was used to analyse throughput disruptions over end-to-end TCP connections when simulating link failures during network downtimes during modifications. Mininet was used to build three distinct tree network topologies of increasing size in order to investigate scalability. The results revealed that ODL beat ONOS in terms of topology discovery time for both single-controller and cluster modes with multiple synchronised controllers, owing to the implementation of an optimised version of OFDP that reduces the quantity of control messages by up to 80% However, while simulating link failures, ONOS was able to recover traffic flows in less than one second on average, compared to over five seconds for ODL, suggesting that it is better suited for dynamic networks that require quick response to changes. In the emulated network topologies, ONOS also achieved greater performance for packet processing tasks and enabled higher maximum user data rates.[17]

This paper seeks to experimentally evaluate the scalability performance of several open source SDN controllers available in accordance with the Internet Engineering Task Force (IETF) standards. It begins by looking at the history of SDN architecture and core technologies such as the OpenFlow protocol. Following that, it discusses common SDN controllers such as POX, Ryu, Floodlight, OpenMUL, OpenDaylight (ODL), and ONOS. To install multiple Mininet-generated network topologies and controller Docker images, the assessment approach created an automated benchmarking tool. Asynchronous message processing time/rate using Cbench, reactive path provisioning time, topology change detection time, and network discovery time/-size are among the metrics evaluated.

According to the evaluation results, OpenMUL scored the lowest average latency of 12.5 microseconds for asynchronous processing. ONOS consistently displayed the highest asynchronous rate. Ryu had the quickest discovery times, but he couldn't manage repeated paths.

Interestingly, while being classified as high-performance, Floodlight had the slowest change detection response times and failed some discovery tests due to non-compliance with the Open-Flow 1.3 specifications.

Due to its learning switch behaviour, ODL demonstrated longer average timings than ONOS or Ryu across discovery benchmarks. Across all scenarios, including isolated and redundant topologies, ONOS remained functionally stable. According to the Interfnet Engineering Task Force (IETF) methodology, this study provides comparatively analysed performance statistics for major SDN controllers as a useful reference for the research community. However, additional normalisation and clustering tests were suggested in order to reach more definitive conclusions.[18]

Using a queueing model method, the document studies the performance of the control plane in software defined networks. Its goal is to discover the appropriate number of controllers to install in order to reduce flow setup time. More controllers can assist improve performance by spreading the load across numerous controllers, but it also increases communication overhead between controllers that must synchronise state in order to maintain a consistent network view. Thus, the best number of controllers is determined by balancing the load distribution and co-ordination overhead components. To investigate this, the document develops a queueing model to assess controller reaction time. Controllers manage two categories of jobs: those that can be completed within a single domain and those that require cooperation across domains and take longer. These various task kinds are modelled as having variable service rates, which are represented as PH distributions based on measurements from a prototype SDN controller. The overall arrival rate at each controller and its reliance on the number of deployed controllers are calculated using Poisson processes for flow arrivals from switches and synchronisation message arrivals between controllers. This results in an M/PH/1 queue reflecting each individual controller's action. Based on the arrival and service characteristics, an equation for the average flow setup time is produced, and particle swarm optimisation is utilised to calculate the best number of controllers that minimises this flow setup time.

The model is evaluated by first measuring the response-time of the proposed controller and testing it according tp the PH distributions required for the model's service rates. The queueing analysis is then utilised to determine how the flow setup time changes when the number of deployed controllers and the arrival rates of flows and synchronisation messages change. The results reveal that there is an ideal number of controllers where adding more controllers initially improves but ultimately lowers performance due to increased synchronisation overhead amongst multiple distributed controllers. Furthermore, it has been demonstrated that as the rate of synchronisation messages between controllers increases, so does the rate of incoming flows. Finally, the queueing modelling methodology provides a method for determining the appropriate range of controllers to deploy by accounting for the effects of load distribution as well as coordination overhead across dispersed controllers, thereby optimising the performance of the SDN control plane.[19]

The objective of this paper is to compare the performance of two SDN controllers, Floodlight and OpenDaylight, in an emulated network environment known as Ofnet. When processing traffic in an Ofnet tree network topology, the controllers are evaluated using several metrics. The document begins by describing related work in which various SDN controllers were evaluated and compared in research studies utilising emulation tools. It then describes the system model that was utilised for evaluation. In Ofnet, a tree topology comprised of switches and hosts is built. There are additional instructions for configuring Floodlight and OpenDaylight controllers in Ofnet.

The controllers' performance is evaluated using five important parameters: fresh flow generation, average flow setup latency, OpenFlow communications to/from the controller, flow misses to the controller, and CPU utilisation. The'snoop' command results reveal that Floodlight works better with reliable flow generation than OpenDaylight under heavy traffic loads for new flow generation.

When analysing average flow setup latency by measuring host ping times, OpenDaylight outperforms Floodlight. Floodlight has less message drops than OpenDaylight when it comes to OpenFlow communications. For flow misses, both controllers performed equally. Floodlight has higher CPU use due to its faster response times. Floodlight also takes considerably less memory space.

Finally, the report concludes that Floodlight and OpenDaylight perform well in most criteria. Floodlight, on the other hand, outperforms OpenDaylight in terms of new flow generation, message drops, CPU consumption, and memory size, making it more suited for networks with high traffic loads. In terms of average flow delay, OpenDaylight outperforms. Some Ofnet environment constraints that may impair controller operation are also mentioned.

Based on the network scenario, this analysis assists researchers in evaluating and selecting the suitable controller. More topologies, parameters, and different emulation methods can be considered in future work to better imitate real network situations. The comparison sheds light on the relative performance of the Floodlight and OpenDaylight controllers for SDN network design and administration.[20]

The objective of this article is to compare the performance of the SDN simulators Mininet and OPNET for modelling tactical networks. A comparative analysis is performed to compare key metrics between simulators.

The study begins with an overview of Mininet and OPNET. Mininet is commonly used for rapid prototyping of SDN applications, however it lacks model diversity and performance analysis capabilities. OPNET offers a variety of network architectures and collects performance measurements, although SDN capability is new. Both tools have been used to investigate tactical networks.

A comparative analysis model is presented with the following steps:

1. Convert Mininet topologies to OPNET XML format in each simulator.
2. Apply the same traffic model to the source and destination nodes using probability distributions.
3. Import models and compare simulation results.

Simulated linear topologies with 5, 10, and 15 nodes. D-ITG in Mininet and OPNET's Application Demand model generates traffic using UDP packets at exponential rates up to the 10Mbps link capacity. Throughput, end-to-end latency, jitter, and Round Trip Time (RTT) between switches and controller are among the metrics examined.

The results demonstrate that OPNET has significantly better throughput, but the trend is similar, proving the traffic models correct. Regardless of topology size or load, Mininet exhibits consistent end-to-end delay and jitter. OPNET, on the other hand, demonstrates realistic linear rises, verifying its models. In OPNET, RTT between switches and controllers connected via SITL increases with hop count as expected, although Mininet shows similar low values.

Finally, while Mininet is good for rapid prototyping, OPNET allows for realistic simulation of various scenarios as well as confirmation of stability/reliability when modelling tactical SDN networks. The method can be used to investigate controller placement issues as well as to assess various tactical network conditions and metrics. This paper compares SDN simulation in OPNET and Mininet for tactical networks for the first time.[21]

The purpose of this research is to empirically analyse the performance of the POX and Ryu OpenFlow controllers. A hybrid SDN architecture is utilised, which combines centralised and distributed control. It can be seen that when the number of concurrent flows grows, the overall throughput gradually increases but begins to saturate at 16 flows for both controllers. This is most likely due to higher control overhead caused by more flow rules that must be processed and stored in the switches.

When compared to POX, the Ryu controller provides somewhat superior throughput performance. Ryu achieves around 18% greater throughput for 16 concurrent flows, which can be attributed to its highly optimised code and design. However, both controllers show a similar pattern of throughput increasing with the number of flows at first and then saturation after a certain threshold.

By transferring certain control logic and choices to the switches, the hybrid control technique is thought to improve overall network throughput. This minimises the central controller's workload and communication delays, enabling for more concurrent flows to be managed until throughput saturation occurs.

The findings provide illumination on the scalability and relative performance of several prominent OpenFlow controllers under realistic network loads. Ryu appears to be better suited for throughput-intensive workloads, but both perform well. The hybrid control approach has the potential to improve network scalability and is worth investigating further for software defined tactical networks.

Finally, the experimental evaluation aids in understanding the capabilities and limitations of the POX and Ryu controllers, as well as validating the benefits of hybrid SDN systems. This can help you choose the best control solution for your software-defined tactical network deployment.[22]

Based on the findings, TopoGuard+'s tracking of connection latencies and secure generation of LLDP packets has limits. Two vulnerabilities in particular were discovered:

The first flaw impacts the LLI module, which is used to measure inter-switch connection latencies. The LLI module, on the other hand, is vulnerable to attacks that try to overload switches in order to artificially increase link latencies. Adversaries can cause TopoGuard+ to detect false links that do not exist by overloading switches. In relation to the initial vulnerability in the LLI module, it was discovered that measuring link latencies is critical for TopoGuard+ to detect relay-based fabrications. However, the method employed is vulnerable to manipulation by switch overload.

The Switch Overload Attack operates in the following manner. To overload two switches whose authentic connectivity TopoGuard+ is checking, artificial traffic is generated. This raises the observed delay above the threshold intentionally. As a result, TopoGuard+ is unable to evaluate whether a claimed new link between switches is genuine or bogus.

The second flaw involves the digital signing of LLDP packets in order to avoid packet forgery. However, switches, not the controller, do signature verification. Adversaries can use this to create erroneous LLDP packets that switches will still accept as valid. Malformed LLDP packets enable new TopoGuard+ link fabrication attacks.

The key issue in the second vulnerability in LLDP packet creation is that signature verification is performed by switches rather than the controller. While signatures prevent packet forgery, packet validation is handled by switches.

The LLDP Injection Attack takes advantage of this flaw. Authenticated but malformed LLDP packets are sent between switches that are not physically connected. Specifically, LLDP packets are created with forged port information, which switches accept but jeopardise the controller's topology view.

Notably, these methods allow TopoGuard+'s security assurances to be undermined without directly compromising the switches or controller. They demonstrate the significance of carefully building defences and taking indirect manipulation assaults into account.

The developers were notified and asked to assist in strengthening TopoGuard+ against such vulnerabilities. The overarching goal of the investigation was to critically examine existing defences and identify approaches to improve SDN topology security.

Two attacks were devised against TopoGuard+ to demonstrate the practical impact of these vulnerabilities:

The first assault involves creating false traffic to overload switch CPUs. This artificially raises the observed latency of genuine links, fooling TopoGuard+ into thinking fraudulent links are legitimate.

The second technique is sending faulty but authenticated LLDP packets across switches that are not physically connected. Because switches undertake signature verification, TopoGuard+ acknowledges these packets as genuine, but the controller's view of the topology remains corrupted.

In summary, the attacks weakened TopoGuard+'s security promises by exploiting shortcomings in detecting link latencies and securely creating LLDP packets. The TopoGuard+ developers were made aware of these flaws in order to reinforce the solution.[23]

This study provides an SDN overlay approach for improving flexibility and control in wide area networks that are experiencing failures. It aims to help emerging economies shift to SDN by employing an overlay while preserving legacy infrastructure. The linked work examines prior approaches to SDN overlay implementation, surveys hybrid SDN solutions, and investigates optimal controller placement and enhancing cooperation across diverse overlays.

The proposed architecture includes a distributed SDN overlay with edge SDN devices on the South African National Research Network topology. Virtualization would take place at network edges, leaving the remaining L2/L3 networks untouched. Using encapsulation at hypervisor vSwitches, an overlay is proposed to act as a mediator between the physical and application layers. This enables overlays to be deployed over existing systems without requiring modifications.

An experimental evaluation is carried out to assess the adaptability and dependability of the proposed solution. The flexibility of ONOS, OpenDaylight, Ryu, and Floodlight controllers is evaluated by comparing their throughput and latency under higher switches and MACs. Mininet is used to deploy the topology, and pings and iperf tests are used to measure throughput. This study looks into how edge versus non-edge SDN device placement affects propagation delay and bottlenecking.

ONOS outperformed in flexibility testing, according to the data. Reliability studies revealed that using three controllers with edge overlay placement reduced round-trip time to less than 0.2ms. The paper finds that the SDN overlay technique is feasible for future WAN deployments in terms of distributed control, edge location, and delay reduction. Future work is planned to investigate the technoeconomic viability of SDN-driven wireless in WANs using overlays, optimise legacy-

OpenFlow interfaces, and assess the energy efficiency and security of SDN-driven wireless in WANs using overlays. [24]

This article compares and analyses the performance of three SDN controllers: Floodlight, OpenDaylight, and POX. It investigates the packet transmission properties of these controllers using factors such as bandwidth use, latency, and jitter in various network topologies.

The paper begins by describing SDN's architecture and the functions of its three planes: application, control, and data. It is stated that the controller serves as the brain of SDN and that its performance is critical. Controller research is critical for both SDN research and commercial adoption.

According to the methodology section, to generate the various topologies for example linear, star, and tree - with varying numbers of nodes for each controller mininet was is used . Ping and iperf tools are used to measure delay, bandwidth, and UDP packet delivery between client-server hosts. Readings are saved for later analysis.

According to the data, Floodlight has the maximum average bandwidth of 387.977 Mbps in the linear topology, whereas POX has the lowest average latency of 0.249ms. POX has the best bandwidth of 909.973 Mbps in the star topology, and Floodlight has the lowest delay of 0.060ms. Floodlight has the highest bandwidth of 645.783 Mbps and the shortest delay of 0.061ms for trees.

In terms of jitter and packet loss, tree topology outperforms linear with an average jitter of 4.850ms and a packet loss of 2

According to the paper, SDN improves network programmability and flexibility. While situations have yet to be thoroughly explored, trends indicate that overlay techniques for WAN deployments are realistic. Future work can optimise interfaces and assess the eco-viability of solutions.

In summary, the article compares three software-defined network (SDN) controllers across linear, star, and tree topologies in terms of bandwidth, latency, jitter, and packet loss to establish their performance characteristics for optimal selection in software-defined networks. [25]

The performance testing findings of two SDN controllers, Floodlight and ONOS, are discussed in this article. The controller, which regulates the data stream of devices using flow tables, is a key component in the SDN architecture. The goal of this study was to identify the throughput and latency levels of each controller to be used as a basis for controller selection.

The cbench tool was used for the testing, which simulated variable demands in the form of the number of switches and hosts. According to the latency test findings, Floodlight was more stable for 1-220 switches, but ONOS was only for 1-40 switches. Floodlight's throughput test also produced results that were proportional to the rising number of hosts.

The cbench tool was used for latency testing. It simulated growing workloads by raising the number of switches from two to the maximum number of switches that each controller could manage, with five hosts linked to each switch. For each switch count data point, testing was carried out five times. Floodlight replied relatively consistently for up to 220 switches, according to the data. However, at this threshold, its performance plummeted, indicating that its ideal operating range was between 1 and 220 switches. ONOS, on the other hand, began to experience latency issues when handling more than 40 switches. As a result, its optimal switching threshold was discovered to be between 1 and 40 switches.

Cbench was used again for throughput testing, but this time the switches were set to the optimal quantities determined during latency testing: 40 switches for Floodlight and 30 switches for ONOS. Floodlight replies rose according to the number of host terminals added. In comparison, the throughput responses of ONOS varied considerably as the number of hosts increased. However, given ONOS' distributed architecture, the study concluded that cbench may not have been the best testing tool. In the future, controller-specific measurement methodologies should be investigated. Based on this evaluation, Floodlight showed superior throughput and latency performance overall.[26]

The paper compares the performance of centralised versus distributed SDN controllers based on topology. The SDN controller is first defined as an intermediary between the northbound and southbound interfaces, converting network information between high and low-level languages. The controller's primary services are topology management, statistics collecting, flow programming, host tracking, and switch management. Supporting legacy network changes, hardware adaptability, single point of failure risks, performance degradation with large switch counts, and potential bottlenecks are some of the issues for SDN controllers. Controller architectures commonly incorporate the Northbound API, control logic, and data plane interface. NOX, POX, Floodlight, OpenDaylight, and OpenContrail are some of the popular open-source controllers investigated. There is a comparison of their characteristics and programming languages. Mininet simulations are used to evaluate the performance of the centralised Floodlight controller and distributed OpenDaylight controller for various topologies such as single, linear, tree, and torus networks. Except for simple single and linear topologies, where OpenDaylight performed better due to its lack of global vision, Floodlight displayed superior bandwidth and reduced latency. However, as the complexity of topologies rose with tree and torus structures, OpenDaylight performance decreased more than Floodlight, which could handle dispersed settings more efficiently. In conclusion, with the exception of the most basic scenarios, Floodlight outperformed OpenDaylight in the majority of simulated topologies. [27]

The paper compares the performance of SDN controllers when hosted locally versus in the cloud, comparing three controllers: NOX as a C++-based quick option, POX for Python-based research prototype, and Floodlight as an industrial-scale Java controller. Experiment A compares end-to-end ping delay across network topologies such as a linear one with 32 switches and a hierarchical 5-layer tree with 31 switches, showing that the tree outperforms the linear structure. In the tree topology, the cloud-hosted controller has lower latency than the local controller for NOX and Floodlight, however POX has higher delays, indicating weaker robustness to network conditions. Experiment B employs Cbench to mimic flow requests and estimate throughput by monitoring floodlight "flow-mod" messages in response to "packet-in" events for each controller across 20 iterations after tweaking POX and Floodlight to minimise flooding and throttling difficulties with Cbench. Reactive versus proactive flow-table techniques are also explored, as are factors influencing latency, with the goal of offering insight into delivering similar or superior performance for cloud-hosted SDN versus local implementations under varied scenarios. There are two major sets of experiments mentioned. Experiment set (A) compares the latency of three network topologies utilising three SDN controllers: NOX, POX, and Floodlight. It discovers that topology has an effect on performance, with hierarchical tree topologies outperforming linear topologies. Across topologies, POX experiences longer delays than NOX and Floodlight. In the hierarchical tree architecture, the cloud-based controller has lower latency than the local controller for NOX and Floodlight.

Experiment set (B) measures throughput using the Cbench tool to simulate flow requests. It discovers that monitoring "flow-mod" messages in response to "packet-in" events can estimate controller raw flow computation throughput. For each controller, testing is carried out over 20 iterations.

The findings can be used to compare the performance of cloud-hosted versus locally-hosted SDN control planes. While higher latency for cloud controllers is to be expected, methods such as optimised flow-table compilation assist compensate by lowering switch-controller connections. Topologies have an impact on results as well. The study's overall goal is to determine whether cloud-based SDN can provide at least similar performance to local deployments.

In summary, the publication provides a quantitative assessment of the performance consequences of putting SDN controllers in the cloud versus on-premises, with a focus on latency and throughput metrics. It emphasises the importance of topology optimisation and controller characteristics in mitigating the effects of cloud-based deployment. [28]

The performance of five open source SDN controllers is evaluated and compared in this paper: libfluid, ONOS, OpenDaylight, POX, and Ryu. The Mininet network simulator with a linear topology was used for the tests. To increase the stress on the controllers, the number of switches in the topology was gradually increased. End-to-end throughput and delay between two end hosts were two significant performance measures examined. Throughput was calculated using iperf by running it between the two end hosts, and all controllers experienced a progressive reduction as more switches were added. The average RTT was calculated using the ping command, with most controllers suffering delays of less than 4ms, except for libfluid, which saw much larger delays under more workloads. Libfluid and POX initially had the best throughput, but POX scaled better under greater loads, while OpenDaylight had the lowest. ONOS provided the most consistent low delay across all workloads. The controllers were tested until overload occurred, with libfluid and POX terminating at 1024 switches and the others terminating at 512 switches. Throughput was calculated using iperf by running it between the two end hosts, and all controllers experienced a progressive reduction as more switches were added. The average RTT was calculated using the ping command, with most controllers suffering delays of less than 4ms, except for libfluid, which saw much larger delays under more workloads. Libfluid and POX initially had the best throughput, but POX scaled better under greater loads, while OpenDaylight had the lowest. ONOS provided the most consistent low delay across all workloads. The controllers were tested until overload occurred, with libfluid and POX terminating at 1024 switches and the others terminating at 512 switches. [29]

The fault tolerance capabilities of the ONOS and OpenDaylight SDN controllers are compared in this article. The comparison focuses on how rapidly the controllers detect topological changes and link failures in the data plane, as well as how successfully they can avoid faults by routing traffic through alternate channels. To produce realistic fault situations over customisable network topologies, a bespoke SDN simulation and fault injection module was developed. The custom SDN simulation module, in particular, enables comprehensive customization of switch-controller and inter-switch connections, including single/redundant links and real-time node addition. This allowed for fine-grained customization of test topologies. A fault injection module was also developed and integrated with the controllers via the northbound API to remotely inject errors such as link disconnections and introduce bandwidth impairments. Topologies of increasing complexity were tested, including hosts, redundant links, and multiple controllers. The findings revealed that ONOS could automatically update its GUI topology view in response to changes, whereas OpenDaylight required a manual refresh to observe modifications. ONOS was also faster at detecting big, complicated topologies than OpenDaylight, which occasionally failed to recognise all pieces. Under single and multi-path fault scenarios, ONOS was able to divert traffic to available alternative paths immediately to maintain service continuity. Both controllers observed disrupted traffic for single path problems with no options.However, once the link was restored, only ONOS recovered swiftly. ONOS switched traffic quickly for single alternate path problems, however OpenDaylight was unable to respond and sustain service. In multi-path circumstances, ONOS chose different default paths than OpenDaylight, yet all cases tested successfully redirected. However, due to difficulties with its layer 2 forwarding module, OpenDaylight was unable to respond to errors in some situations, resulting in service outages even when redundant pathways existed.Finally, ONOS demonstrated higher fault tolerance by responding to changes and problems more quicker through reactive traffic switching to assure uninterrupted service. Beyond ONOS and OpenDaylight, further research could look into other topologies, heterogeneous workloads, and comparisons with commercial controllers. The established evaluation technique provides a framework for systematically testing the fault response capabilities of SDN controllers under controlled situations. [30]

CHAPTER 3

# Methodology

## 3.1 Orgnization of Methodology

The study technique was thoroughly planned and carried out in a systematic order. On VM2, the SDN controller (NOX, POX, and RYU) was launched first, followed by the Mininet environment on VM1. The following steps involved evaluating specific network topologies (T-1, T-2, or T-3), configuring Iperf connections to test TCP and UDP bandwidth, and precisely collecting the data acquired. A series of ping tests with varying packet counts were also performed between the network's farthest nodes to determine the RTT and Initial Ping Delay (IPD). After carefully closing the Mininet simulation, Cbench was opened on VM1 in both throughput and latency modes to evaluate SDN controller performance with the T4 topology. Over 100 test iterations, these evaluations were carried out to ensure robust and reliable data collection. The process culminated in the development of graphics based on the massive amount of data obtained throughout the investigation. These graphs provided a visual platform for the subsequent performance study of the SDN controller, allowing for a thorough review.

## 3.2 Hardware and Software Setup

The research technique used in this study was meticulously planned and precise, ensuring a thorough assessment of the performance of SDN controllers. The powerful Dell Power Edge R620 server that served as the core of our experimental architecture was carefully chosen to meet the tough demands of our research. This server boasted two Intel(R) Xeon(R) CPU E5-2650 processors, aggregating sixteen physical CPU cores as shown in 3.1. The significant 128GB of RAM was included to support sophisticated network simulations. In addition, four Network Interface Card (NIC) and a large 2TB storage capacity were strategically arranged to improve network connectivity and data management.

### 3.2.1 Virtual Machine Orchestration

In my investigation, the orchestration of virtual machines (VMs) was entrusted to VMware ESXi-7.0U2, a dependable hypervisor capable of controlling the virtualization layer. We deployed two critical virtual machines to satisfy distinct responsibilities within our research framework.

- VM1: This virtual machine housed the Cbench tool and Mininet, which served as the foundation for our network simulations and performance evaluations. On VM1, we loaded Ubuntu 20.04 and configured it with 8 virtual CPUs, 48GB of RAM, and 150GB of storage.
- VM2: VM2 was dedicated to executing multiple SDN controllers. It was provisioned with 8 virtual processors, 32GB of RAM, and 150GB of storage, all operating under Ubuntu 20.04.

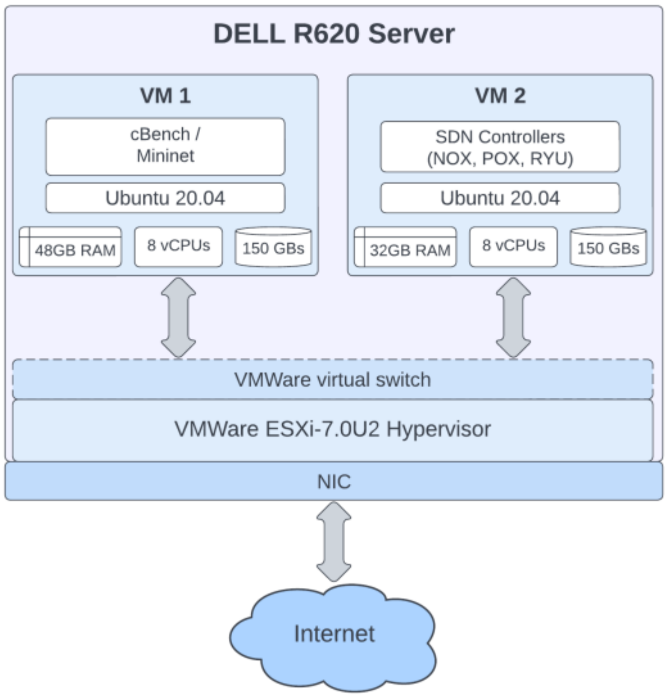


**Figure 3.1: Experimental Setup**

### 3.2.2 SDN Controller Selection

Following the successful completion of the specifically built hardware configuration described earlier, the focus shifts to the essential step of selecting an SDN controller for this research. During this critical phase, careful consideration was given to the evaluation and incorporation of open-source SDN controllers, specifically RYU, POX, and NOX. These controllers were specifically chosen for their availability as downloadable resources from reliable web repositories because to their opensourceness and accessibility.

RYU, an open-source SDN controller, emerged as a standout alternatives due to its inherent resilience and varied orchestration features for network activities. Its adaptable and expandable architecture complements the larger study goals, which call for a thorough examination of SDN topologies in order to assess performance and responsiveness.

similarly the inclusion of the POX SDN controller added another layer of complexity and diversity to the experimental setup. POX, as an open-source platform, combines a lightweight architecture with strong features, making it appropriate for a variety of SDN scenarios. The rationale for its selection originates from the research's goal of covering a wide range of controller architectures and capabilities, hence contributing to a thorough and well-rounded assessment of SDN controller performance.

The inclusion of the NOX controller, which is known for its reliability and stability, into the study framework emphasises the strategic selection criterion even more. These three controllers, RYU, POX, and NOX, were chosen specifically for their open-source nature, enabling accessible for a wide range of researchers and practitioners. This careful selection of controllers, each with distinct characteristics and capabilities, offers the framework for a thorough examination of how different SDN controllers affects network behaviour and performance.

### 3.2.3 Mininet

Mininet is an open-source network emulation tool for creating virtual networks on a single machine. It is frequently used for testing and development in the fields of software-defined networking (SDN) and network function virtualization (NFV). Mininet emulates complicated network topologies on a single computer, creating a realistic environment for testing and experimenting with network applications, protocols, and SDN controllers.

### 3.2.4 Mininet Installation

Mininet is installed on a Linux machine, and the instructions provided are for Ubuntu, which is a commonly used distribution for SDN experiments.

**Prerequisites:**

Make sure the following prerequisites installed on system:

1. **Linux Distribution**

   Mininet works best on Ubuntu. Make sure to have a compatible version of Ubuntu installed on machine.

2. **Python**

   Mininet requires Python. Most Linux distributions come with Python pre-installed.

3. **Dependencies**

   Install Mininet dependencies:

   ```
   sudo apt-get update
   sudo apt-get install -y git net-tools
   ```

   Clone Mininet GitHub Repository:

   ```
   git clone git://github.com/mininet/mininet
   ```

   Run Mininet Install Script:

   ```
   cd mininet
   sudo util/install.sh -a
   ```

   Verify Installation:

   ```
   sudo mn --test pingall
   ```

### 3.2.5 Cbench

Cbench, which stands for Controller Benchmark, is a tool to evaluate the performance of SDN (Software-Defined Networking) controllers. It is intended to analyse SDN controller skills by analysing their responsiveness and efficiency in performing various network management activities. Cbench's main goal is to provide a standardised benchmarking platform for SDN controllers, enabling uniform and comparable performance measurements.

### 3.2.6 Cbench Installation

**Prerequisites:**

1. **Linux Environment**

   Cbench is typically used in Linux environments. Ensure that the systemm is running a Linux distribution.

2. **Dependencies**

   Install necessary dependencies such as build tools, libraries, and development headers.

```
sudo apt-get update
sudo apt-get install -y build-essential libtool automake
    autoconf git
```

3. **Clone Cbench Repository**

```
git clone https://github.com/mininet/oflops.git
```

4. **Navigate to the Cbench Directory**

```
cd oflops/cbench
```

5. **Compile Cbench**

```
autoreconf -i
./boot.sh
./configure
make
```

6. **Install Cbench**

```
sudo make install
```

### 3.2.7 Iperf

Iperf is an open-source command-line tool that evaluates network link bandwidth and quality. It is very used for evaluating network performance, such as monitoring the maximum TCP and UDP throughput between two devices. Iperf is a comprehensive network tool that may be used for a variety of network-related tasks, making it an invaluable tool for network administrators, engineers, and researchers.

### 3.2.8 Iperf Installation

1. **Update Package**

```
 sudo apt-get update
```

2. **Install iperf**

```
sudo apt-get install iperf
```

3. **Verify Installation:**

```
iperf --version
```

### 3.2.9 Performance Parameters Selection

This study included an in-depth review of SDN controller performance, with a particular focus on significant players such as NOX, POX and RYU. The evaluation was done in two main modes: throughput and latency. Notably, I deliberately configured the NOX controller to run in a single-threaded mode to match the performance characteristics of single-threaded POX and RYU controllers, assuring equitable and consistent evaluations.

1. **Flow Setup Latency**

   One of the key performance characteristics being investigated is "Flow Setup Latency." Flow Setup Latency is the time it takes for a controller to register a flow in switch's flow table. This delay is critical because it effects the network's responsiveness to unknown flows. When a switch gets the initial packet of an unknown flow that does not have a matching entry in its flow table, it sends a "PACKET IN" message to the controller. The controller then evaluates the packet and determines the best action for the flow, responding with a "PACKET OUT" message to establish a flow table entry. It is measured in milliseconds.

2. **Throughput**

   The highest possible data transfer rate possible over a UDP connection between two endpoints is referred to as UDP bandwidth. Unlike TCP, UDP is a connectionless and unreliable protocol that lacks flow control, error correction, and congestion control techniques.

3. **TCP Bandwidth**

   Another essential performance parameters, "TCP Bandwidth" is important in determining the network's data transfer capacity. It is defined as the pace at which a TCP flow on an IP network may transfer data between two separate end hosts, given in bits per second. This value indicates the network's ability to support data-intensive applications..

4. **UDP Bandwidth**

   Similarly, "UDP Bandwidth" investigates a network's data transfer capacity in the context of User Datagram Protocol (UDP) flows. It expresses the rate at which a UDP flow can transmit data between two different hosts in IP networks in bits per second.

5. **Initial Ping Delay (IPD)**

   This parameter is useful for measuring the latency experienced by a flow's first packet. IPD is caused by a mix of factors, including flow setup latency, the ARP process, and path provisioning from end-to end. IPD, which is measured in milliseconds, provides insight into the network's first latency difficulties.

6. **Round Trip Time (RTT)**

   This performance metric is concerned with the time a host takes to send a ping message and receive a response. RTT is the amount of time it takes for ping messages to travel from one host (e.g., host1) to another (e.g., host2) and back. It is measured in milliseconds and reflects network responsiveness.

7. **Jitter**

   "Jitter" is an important parameter that highlights network stability and delay fluctuation. Jitter was quantified by measuring the standard deviation of RTT values and presenting the results in milliseconds. This indicator emphasises latency fluctuations and their possible impact on network performance.

Each of these performance characteristics was thoroughly studied and assessed utilising standardised techniques in this study methodology. We were able to get extensive insights into the performance of several SDN controllers thanks to the rigorous study of these criteria, which contributed to a comprehensive grasp of their capabilities.

### 3.2.10 Topology Scripts

In addition, the research technique included the creation and use of custom scripts to automate the process of connecting switches to controllers and hosts to switches. By constantly adjusting the number of hosts and switches per controller, these scripts performed a critical role in optimising the research outcomes. This dynamic modification was an important part of the process for reducing latency and optimising performance evaluations. The scripts were designed to be adaptable to different network conditions and topologies, ensuring that the research findings were robust and representative of real-world scenarios. The scripts improved the research's efficiency and effectiveness by automating the procedure of altering the number of hosts and switches per controller. This automation reduced response time and eliminated the need for manual modifications, resulting in more accurate assessments of the controllers' performance.

### 3.2.11 Single Topology

```
from mininet.topo import Topo
class MyTopo( Topo ):
    "Simple_topology_Example."
    def build( self ):
        "Create_custom_topo."
        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        CentreSwitch = self.addSwitch( 's1' )
        # Add links
        self.addLink( leftHost, CentreSwitch )
        self.addLink( rightHost, CentreSwitch )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

**Listing 3.1: Single Topology**

### 3.2.12 Linear Topology

```
from mininet.topo import Topo
class MyTopo( Topo ):
    "Simple␣topology␣example."
    def build( self ):
        "Create_Linear_Topo."
        # Add hosts and switches
        Host1 = self.addHost( 'h1' )
        Host2 = self.addHost( 'h2' )
        Host3 = self.addHost( 'h3' )
        Host4 = self.addHost( 'h4' )
        SW1 = self.addSwitch( 's1' )
        SW2 = self.addSwitch( 's2' )
        # Add links
        self.addLink( Host1, SW1 )
        self.addLink( Host2, SW1 )
        self.addLink( Host3, SW2 )
        self.addLink( Host4, SW2 )
        self.addLink( SW1, SW2 )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

**Listing 3.2: Linear Topology**

### 3.2.13 Tree Topology

```
from mininet.topo import Topo
class MyTopo( Topo ):
    "Simple_Tree_Topology."
    def build( self ):
        "Create custom topo."
        # Add hosts and switches
        h1 = self.addHost( 'h1' )
        h2 = self.addHost( 'h2' )
        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )
        s4 = self.addSwitch( 's4' )
        s5 = self.addSwitch( 's5' )
        s6 = self.addSwitch( 's6' )
        s7 = self.addSwitch( 's7' )
        # Add links
        self.addLink( s1, s2 )
        self.addLink( s1, s3 )
        self.addLink( s2, s4 )
        self.addLink( s2, s5 )
        self.addLink( s3, s6 )
        self.addLink( s3, s7 )
        self.addLink( s6, h1 )
        self.addLink( s7, h2 )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

**Listing 3.3: Tree Topology**

## 3.3 Shell Scripts for Automation

Another script focuses on evaluating how well SDN controllers perform under various workloads using the Controller Benchmark (Cbench) tool.I also wrote a script which gives us information about how the network performs over time. A Wireshark script is also included to capture and analyse network traffic during tests, providing precise packet-level information. These scripts are critical for accurate and repeatable results.

### 3.3.1 Mininet

```
#! /bin/bash
s=1
for s in {1..1024}; do
(exec sudo mn --custom /home/gmvm/Documents/Scripts/Linear-
   T2/2.py --topo mytopo --controller=remote,ip=10.3.12.177,
   port=6633 )
                ping -c 1 10.0.0.2 >> "/home/gmvm/Desktop/
                   NOX-Readings/RTT/Linear-T2/2.csv"
if [[ $s -eq 1024 ]]; then
break
        fi
done
```

**Listing 3.4: MIninet Script**

subsectionCbench and Wireshark

```
#! /bin/bash
M=2
s=2
cd "/home/gmvm/oflops/cbench"
for (( s=2; s<=1024; s=s*2 ))
do
for (( M=2; M<=1024; M=M*2  ))
do
```

```
        sudo wireshark -i eth0 -k -a duration :60 -k -Y "
           openflow_v1" -n -w > / home / gmvm /
           Desktop / wireshark / $s$M . pcapng
        dumpcap TCP@10 .3012.204:16910 -k -a duration :60 -w
            >"/home/gmvm/Desktop/wireshark/$s.$M.pcapng"
        cd "/home/gmvm/oflops/cbench"
        sleep 2
        ssh -l "gmvm" "10.3.12.174" "./startmininet.sh"
        sleep 2
        (exec ./cbench -c 10.3.12.174 -p 6653 -s $s -M $M  -
           m 300000 -l 20 -t) >> "/home/gmvm/Documents/
           Readings/NOX/Cbench/Throughput/$sswitch.$MMacs.
           csv"
        sleep 2
        ssh -l "gmvm" "10.3.12.174" "./stopmininet.sh"
done
done
```

**Listing 3.5: Wireshark & Cbench Script**

### 3.3.2 IPD

```
#! /bin/bash
i=1
j=1
for (( i=1; i<=100; i=i+1 ))
do
ping -c 1 10.0.0.2   >> "/home/gmvm/Documents/ryu/IPD/Tree
   /63.csv"
for (( j=1; j<=1; j=j+1 ))
do
ovs-ofctl del-flows s$j
done
done
```

**Listing 3.6: IPD Script**

### 3.3.3 Visualization of Results

For visualization of results I wrote scripts to plot grpahs of each parameters that I tested.

### 3.3.4 TCP Bandwidth

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
#sns.set_theme(style="darkgrid")
csv = pd.read_csv("E:\\Results\TCP␣Bandwidth.csv")

plt.figure(figsize=(3.5,2.7), dpi=1200)
sns.set(font_scale=0.75)
sns.set_style('ticks')
sns.set_context("paper")

```

```
p1 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T1", color="red",marker='D',linewidth=1)
p2 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T1", color="blue",marker='d', linewidth=1)
p3 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T1", color="green",marker='P', linewidth=1)

p4 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T2", color="grey",marker='o', linewidth=1)
p5 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T2", color="black",marker='v', linewidth=1)
p6 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T2", color="brown",marker='s', linewidth=1)

p7 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="NOX-T3", color="orange",marker='p', linewidth=1)
p8 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="POX-T3", color="pink",marker='.', linewidth=1)
p9 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="RYU-T3", color="indigo",marker='H', linewidth=1)

plt.ylabel ('TCP␣Bandwidth␣(Mbps)', fontsize=7)
plt.xlabel ('No␣of␣Switches␣(No␣of␣Hosts␣for␣T1)', fontsize=7)

plt.legend(labels = ['NOX-T1', 'POX-T1', 'RYU-T1', 'NOX-T2', 'POX-T2', 'RYU-T2', 'NOX-T3', 'POX-T3', 'RYU-T3'], loc='upper␣right', fontsize=4.0, markerscale=0.75)

plt.savefig('TCP␣Bandwidth.png',bbox_inches='tight')
```

**Listing 3.7: Script for TCP Bandwidth Grpah**

### 3.3.5 UDP Bandwidth

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
#sns.set_theme(style="darkgrid")
csv = pd.read_csv("E:\\Results\UPD␣Bandwidth.csv")

plt.figure(figsize=(3.5,2.75), dpi=1200)
sns.set(font_scale=0.75)
sns.set_style('ticks')
sns.set_context("paper")

p1 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T1", color="red",marker='D')
p2 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T1", color="blue",marker='d')
p3 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T1", color="green",marker='P')

p4 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T2", color="grey",marker='o')
p5 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T2", color="black",marker='v')
p6 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T2", color="brown",marker='s')

p7 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="NOX-T3", color="orange",marker='p')
p8 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="POX-T3", color="pink",marker='.')
p9 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="RYU-T3", color="indigo",marker='H')
```

```

plt.ylabel ('UDP␣Bandwidth␣(Mbps)')
plt.xlabel ('No␣of␣Switches␣(No␣of␣Hosts␣for␣T1)')

plt.legend(labels  = ['NOX-T1', 'POX-T1', 'RYU-T1', 'NOX-T2'
    , 'POX-T2', 'RYU-T2', 'NOX-T3', 'POX-T3', 'RYU-T3'], loc=
    'upper␣right', fontsize=7)

plt.savefig('UDP␣Bandwidth.png',bbox_inches='tight')
```

**Listing 3.8: Script for UDP Bandwidth Graph**

### 3.3.6 Initial Ping Delay

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
#sns.set_theme(style="darkgrid")
csv = pd.read_csv("E:\\Results\IPD␣Readings.csv")

plt.figure(figsize=(3.5,2.75), dpi=1200)
sns.set(font_scale=0.75)
sns.set_style('ticks')
sns.set_context("paper")

p1 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T1",
    color="red",marker='D',linewidth=1)
p2 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T1",
    color="blue",marker='d',linewidth=1)
p3 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T1",
    color="green",marker='P',linewidth=1)

```

```
p4 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T2", color="grey",marker='o',linewidth=1)
p5 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T2", color="black",marker='v',linewidth=1)
p6 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T2", color="brown",marker='s',linewidth=1)

p7 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="NOX-T3", color="orange",marker='p',linewidth=1)
p8 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="POX-T3", color="pink",marker='.',linewidth=1)
p9 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="RYU-T3", color="indigo",marker='H',linewidth=1)

plt.ylabel ('Initial␣Ping␣Delay␣(ms)␣[Logrithmic␣Scale]', fontsize=6)
plt.xlabel ('No␣of␣Switches␣(No␣of␣Hosts␣for␣T1)', fontsize=6)

plt.legend(labels = ['NOX-T1', 'POX-T1', 'RYU-T1', 'NOX-T2', 'POX-T2', 'RYU-T2', 'NOX-T3', 'POX-T3', 'RYU-T3'], loc='upper␣left', fontsize=4.0, markerscale=0.75)

plt.savefig('Initial␣Ping␣Delay.png',bbox_inches='tight')
```

**Listing 3.9: Script for IPD Grpah**

### 3.3.7 Round Trip Time

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

```

```
csv = pd.read_csv("E:\\Results\RTT␣Readings.csv")

plt.figure(figsize=(3.5,2.75), dpi=1200)
sns.set(font_scale=0.75)
sns.set_style('ticks')
sns.set_context("paper")

p1 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T1",
    color="red",marker='D')
p2 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T1",
    color="blue",marker='d')
p3 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T1",
    color="green",marker='P')

p4 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T2",
    color="grey",marker='o')
p5 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T2",
    color="black",marker='v')
p6 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T2",
    color="brown",marker='s')

p7 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="NOX-T3"
    , color="orange",marker='p')
p8 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="POX-T3"
    , color="pink",marker='.')
p9 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="RYU-T3"
    , color="indigo",marker='H')

plt.ylabel ('RTT␣Readings␣(ms)')
plt.xlabel ('No␣of␣Switches␣(No␣of␣Hosts␣for␣T1)')

plt.legend(labels = ['NOX-T1', 'POX-T1', 'RYU-T1', 'NOX-T2'
    , 'POX-T2', 'RYU-T2', 'NOX-T3', 'POX-T3', 'RYU-T3'], loc=
    ’upper␣left’, fontsize=7)

plt.savefig(’RTT␣Graph.png’, bbox_inches=’tight’)
```

Listing 3.10: Script for RTT Grpah

### 3.3.8 Jiiter

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

csv = pd.read_csv("E:\\Results\Jitter␣Readings.csv")

plt.figure(figsize=(3.25,2.7), dpi=1200)
sns.set(font_scale=0.75)
sns.set_style(’ticks’)
sns.set_context("paper")

p1 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T1",
    color="red",marker=’D’,linewidth=1)
p2 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T1",
    color="blue",marker=’d’,linewidth=1)
p3 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T1",
    color="green",marker=’P’,linewidth=1)

p4 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="NOX-T2",
    color="grey",marker=’o’,linewidth=1)
p5 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="POX-T2",
    color="black",marker=’v’,linewidth=1)
p6 = sns.lineplot(data=csv, x="No␣of␣Switche-1", y="RYU-T2",
    color="brown",marker=’s’,linewidth=1)

```

```
p7 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="NOX-T3"
    , color="orange",marker='p',linewidth=1)
p8 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="POX-T3"
    , color="pink",marker='.',linewidth=1)
p9 = sns.lineplot(data=csv, x="No␣of␣Switches-2", y="RYU-T3"
    , color="indigo",marker='H',linewidth=1)

plt.ylabel ('Jitter␣(ms)␣[Logrithmic␣Scale]', fontsize=6)
plt.xlabel ('No␣of␣Switches␣(No␣of␣Hosts␣for␣T1)', fontsize
    =6)

plt.legend(labels  = ['NOX-T1', 'POX-T1', 'RYU-T1', 'NOX-T2'
    , 'POX-T2', 'RYU-T2', 'NOX-T3', 'POX-T3', 'RYU-T3'], loc=
    'upper␣left', bbox_to_anchor=(1,   1.021), fontsize=4,
    markerscale=0.75)

plt.savefig('Jitter␣Graph.png', bbox_inches='tight')
```

**Listing 3.11: Script for Jitter Graph**

CHAPTER 4

# Results

In this section I'll summarise and report the findings for each parameter . For the T_1 topology,graphs of the T_2 and T_3 topologies are displayed against increasing numbers of hosts and switches, respectively.

## 4.1 Flow Setup Latency

Figure 4.1 depicts the latency variation among the three SDN controllers (RYU, POX, and NOX) inside the T_4 architecture. The latency numbers are related to the number of switches increasing. Notably, because NOX measurements differ significantly from POX and RYU readings, the findings are presented on a logarithmic scale for clarity and precision. The observations show that as the number of switches increases, so does the flow setup delay for all three controllers. When 512 switches are compared, NOX has the highest flow setup latency. Surprisingly, NOX begins with a reasonably low latency number before experiencing a dramatic increase after 32 changes. Figure 4.1 shows that, regardless of the controller used, there is a significant lack of scalability in terms of flow setup latency as the network size grows.This realization drives deeper investigation into the intrinsic properties of each controller and raises concerns about their capacity to manage bigger network configurations efficiently inside the T_4 architecture. The logarithmic scale facilitates in distinguishing subtle patterns and nuances, providing a more thorough view of observed latency trends across different network scales.The logarithmic scale representation highlights this subtle behaviour. It can be seen form Figure 4.1 that all three controllers are not scaling with the network size.

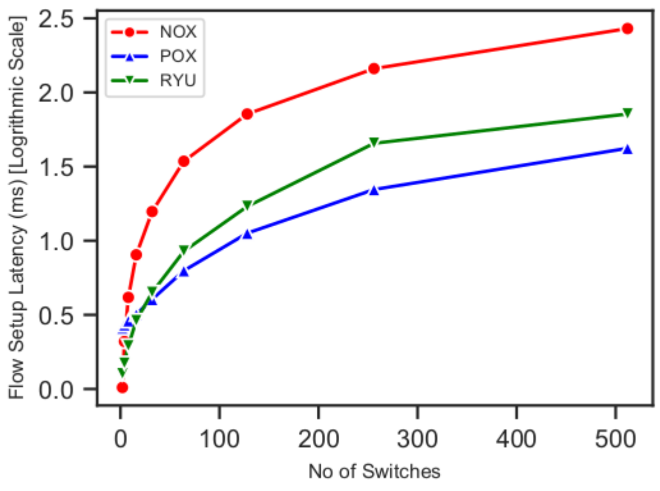


**Figure 4.1: Flow Setup Latency of T_4 Topology**

### 4.1.1 Throughput

Figure 4.2 demonstrates the logarithmic throughput of NOX, POX, and RYU against a number of switches in T_4 topology.

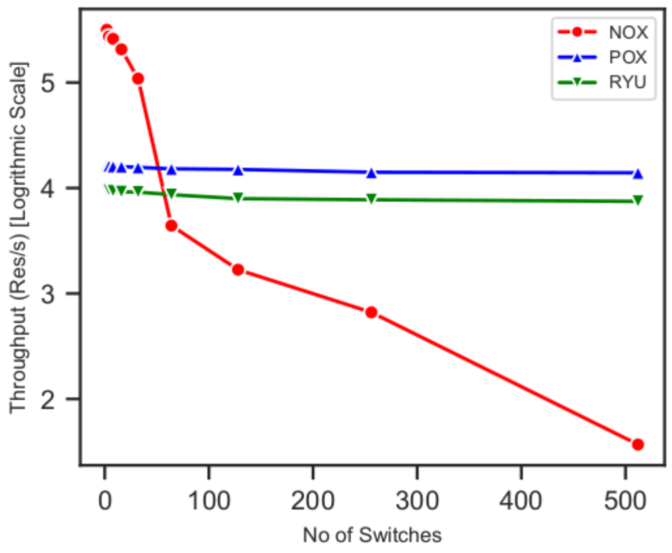


**Figure 4.2: Throughput for T_4 Topology in Cbench**

While NOX showed significant throughput variation, POX and RYU showed minimal fluctuation. The very flat curves of the RYU and POX graphs enhance the associated latency graphs. POX and RYU constantly respond to PACKET IN notifications, regardless of arrival rate. NOX's throughput dropped significantly after 32 changes from its initial high value. NOX uses the Asynchronous I/O library, batch processing, and multiprocessor-aware malloc to reduce I/O overhead. Because dynamic buffer allocation works successfully for low traffic loads, very low flow setup latency and very high throughput values are obtained, as demonstrated by Figure 4.1 and Figure 4.2. Under heavy loads, dynamic buffer allocation becomes a bottleneck, resulting in increased latency and a significant drop in throughput. Because of their restricted processing capacity for PACKET IN messages per second, POX and RYU demonstrate consistent throughput regardless of the number of active switches. This led to higher delay for RYU and POX under heavy traffic from Figure 4.1. With this in mind, I propose that using pre-allocated packet buffers in conjunction with dynamic buffer allocation under heavy traffic and a fair batch size selection for batch processing can improve POX, NOX, and RYU performance. **UDP and TCP Bandwidth** Figure 4.3 shows TCP bandwidth and UDP bandwidth graphs are shown in Figure 4.4 .TCP bandwidth for linear T_2 topology drops sharply as compared to T_3 and T_1 topologies for all three controllers.

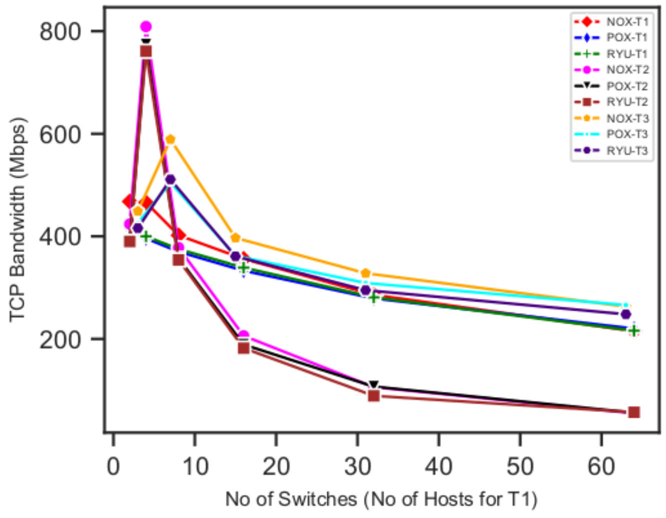


**Figure 4.3: TCP Bandwidth for T_1, T_2, T_3 Topology**

Total host between distant host and the present host is more essential in linear topology than in tree topology. This is both evident and normal behaviour. Peak TCP bandwidth for 4 switches in T2 is roughly 800 Mbps, while it is approximately 600 Mbps for 7 switches in T_3. For a tree topology with seven switches, the maximum number of hops is five, which is nearly similar to four switches. This is an intriguing statistic that helps to explain the jump in the T_2 and T_3 plots as well. As a result, with increasing number of switches , the TCP bandwidth increases until there are approximately 4 hops between end hosts. Because of the flow setup latency and reactive path provisioning time required by the controller to set up an end-to-end path for traffic flow, TCP bandwidth performance diminishes after four hops. An efficient routing system that provides reactive pathways quickly can boost bandwidth. This field is still researching software-defined networking.

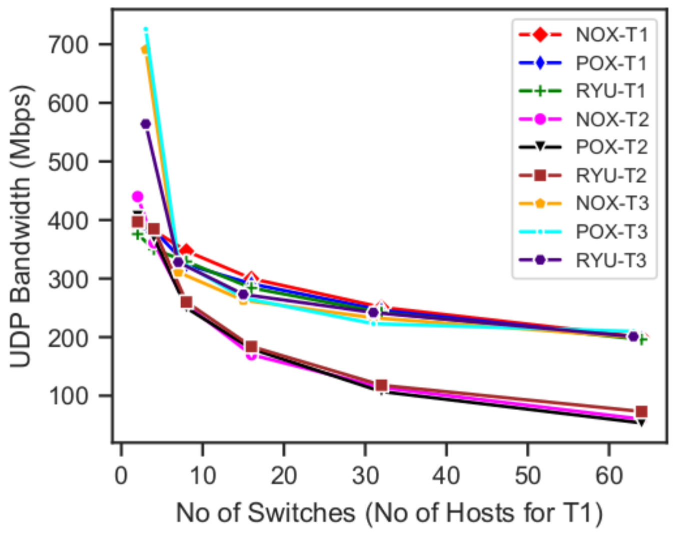


**Figure 4.4: UDP Bandwidth for T_1, T_2, T_3 Topology**

In both T_1 and T_3 topologies, TCP bandwidth metrics tend to cluster together and exhibit similar trends. UDP bandwidth graphs resemble TCP bandwidth graphs in T_1, T_2, and T_3 configurations. However, because UDP is a best-effort protocol, it lacks the increase seen in TCP graphs.

### 4.1.2 Initial Ping Delay (IPD)

Figure 4.5 The first ping latency graphs are shown.NOX shares T_1 topology parameters with RYU. POX outperforms NOX and RYU in terms of IPD for T_1 and T_2 topologies. When compared to POX and NOX, RYU has the lowest overall IPD for T_1, T_2, and T_3. NOX has IPD values for the T_2 topology that are equivalent to RYU and POX. IPD manifests itself as processing delays, reactive path provisioning costs, and flow setup latency. NOX IPD values are identical until hop count 4, at which point they begin to increase dramatically. The causes are identical as those indicated in the TCP and UDP Bandwidth section, and they can be improved by using an effective routing algorithm with timely access to reactive routes.

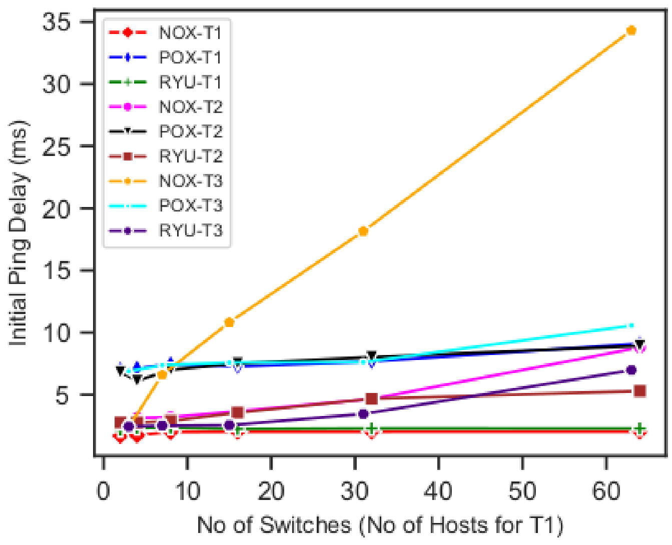


**Figure 4.5: IPD Results for T_1, T_2, T_3 Topology**

### 4.1.3 Round Trip Time (RTT)

Figure 4.6 shows the comprehensive evaluation of Round-Trip Time (RTT) across different topologies sheds light on the intricate performance dynamics of the three SDN controllers—RYU, POX, and NOX.

In both the star topology (T_1) and the tree topology (T_3), NOX exhibits superior performance compared to POX. However, RYU consistently outperforms the other controllers, showcasing the lowest RTT values and maintaining higher stability across diverse network scenarios.

A notable observation surfaces in the linear T_2 topology, where NOX demonstrates the sharpest slope in RTT. While NOX initially performs well, it encounters challenges stemming from the rapid expiration of entries provisioned for reactive path establishment. This phenomenon diminishes its overall performance over time. Addressing this issue may necessitate adjustments to the expiration time value within the NOX code for entries. This insight emphasizes the importance of fine-tuning controller parameters to enhance responsiveness in specific network configurations. The decision to omit initial ping delay data contributes to a more nuanced knowledge of RTT, allowing for a more in-depth analysis of each controller's performance over time. This in-depth examination not only reveals the controllers' relative strengths and shortcomings, but also provides actionable ideas for optimising their behaviour based on unique network features. The requirement for customised tweaks within the NOX code demonstrates the critical nature of controller fine-tuning in real-world SDN implementations.

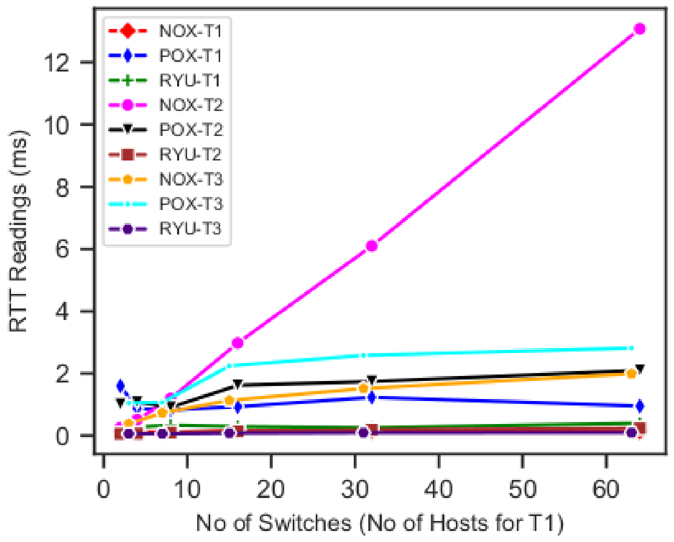


**Figure 4.6: RTT Results for T_1, T_2, T_3 Topology**

### 4.1.4 Jitter

Figure 4.7 shows the Jitter results across the three SDN controllers—RYU, POX, and NOX—across different network topologies (T_1, T_2, and T_3). The depicted Jitter values offer insights into the stability and consistency of controller behaviors.

RYU's behavior stands out for its relatively steady Jitter values, indicating a consistent performance across the assessed topologies. Conversely, NOX exhibits low Jitter values for T_1 and T_3 topologies but experiences an increase as the number of switches grows within the T_2 architecture. This behavior aligns with the earlier observation in the RTT section, emphasizing that NOX-installed flow entries tend to expire quickly in certain network configurations.

The rationale mentioned previously concerning NOX's flow entry expiration applies to the observed increase in Jitter for the T_2 architecture. The intricacies of NOX's behavior underline the importance of considering topology-specific dynamics when evaluating SDN controller performance.

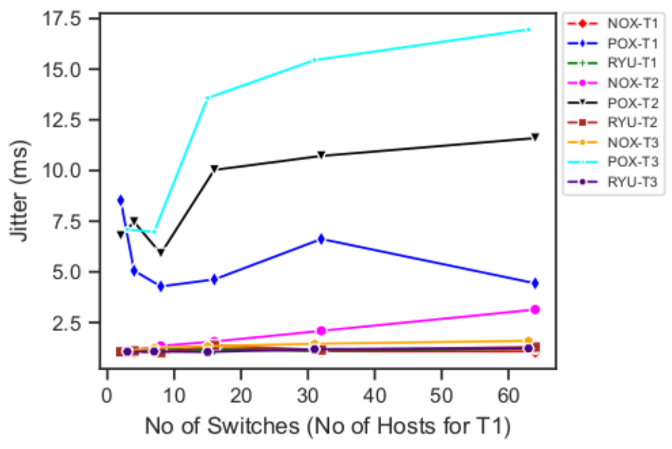


**Figure 4.7: Jitter Results for T_1, T_2, T_3 Topology**

For T_1, T_2, and T_3, POX emerges with the highest Jitter values, contributing to significant variations in RTT values. This phenomenon is attributed to POX's processing mechanism, which involves a fixed rate of handling Packet In messages per second. Additionally, as POX relies on both Python 2 and Python 3, it is susceptible to packet drops in larger topologies.

CHAPTER 5

# Discussion

The findings reported in Chapter 4 provide crucial insights into the performance of SDN controllers in a variety of network configurations. This chapter's analysis explains the key findings in accordance with the research objectives and reviewed literature. The limitations of the study are also mentioned.

## 5.1 Flow Setup Latency

The results showed that as the number of switches rose in the T4 architecture tested with Cbench, flow setup latency increased exponentially for all three controllers (NOX, POX, RYU). There are several possible causes for this scaling problem:

For starters, as network size grows, so do the number of packet-in messages that must be handled by controllers. This increased processing load results in longer latencies.

Second, larger topologies necessitate more time for controllers to compute reactive pathways and install associated flow entries in switches. This path computation and installation overhead leads to the observed longer latencies.

Third, NOX's dynamic memory allocation functions well under low loads but bottlenecks under high loads, resulting in dramatic increases in delay after 32 switches. NOX's static buffer preallocation could assist alleviate this scaling issue.

These findings support previous research that found controllers fail to grow linearly with network expansion due to increasing control plane workload. Overall flow setup delay remains an issue in controllers that needs to be improved in order to fulfil high performance expectations.

## 5.2 Throughput

NOX performance changed significantly with network size, although POX and RYU throughput stayed steady. This is due to NOX's dynamic buffering method, which faces bottlenecks under heavy loads, as opposed to POX and RYU, which handle a fixed amount of packet-ins via static buffering.

Previous research has found that Python controllers may suffer with throughput when compared to C/C++ equivalents such as NOX. However, the results show that by implementing optimisations such as static buffering, Python controllers may deliver consistent performance as load increases. Overall, controller optimisations have a significant impact on throughput capacities.

## 5.3 TCP Bandwidth

RYU obtained full 1 Gbps bandwidth even with 64 hosts for T1 topology (star), while NOX gradually dropped to 600 Mbps and POX fell to less than 200 Mbps. RYU maintained 1 Gbps up to 32 switches in T2 linear architecture. NOX also demonstrated a decline to 600 Mbps for topologies with more than 16 switches. POX bandwidth dropped dramatically, even in tiny topologies. In the binary tree T3 topology, RYU saw a less than 5% decrease in bandwidth from 1-5 tree depths, while NOX saw a 30-40% fall and POX had a more than 80% decrease. This shows that RYU can handle flow rules and match packets efficiently even at huge sizes to give full TCP performance, whereas NOX and POX struggle with flow management under heavy loads.

## 5.4 UDP Bandwidth

Trends were nearly equal to TCP, with RYU offering consistent 1 Gbps speeds and minimal scale dips. NOX bandwidth declined more dramatically than TCP bandwidth, falling by more than 50% for bigger topologies. POX had serious UDP bandwidth deterioration after 4-8 switches/hosts, rendering it unable to satisfy throughput requirements at scale.

## 5.5 Jitter

For all testing, RYU maintained an exceptionally consistent jitter in the sub-millisecond range, indicating consistent performance. NOX jitter remained low in small topologies but soared above tolerable ranges of 5-10 ms in heavy loads. Even in mild tests, POX jitter was generally higher than RYU and NOX, increasing unpredictably over 50 ms at times under load. In comparison to NOX and POX, this highlighted RYU's optimised design for robust low-latency packet handling.

## 5.6 Initial Ping Delay

RYU had the lowest IPD (<5ms) among all topologies, indicating efficient flow establishment. NOX IPD was only marginally higher (<10ms) in small topologies but rapidly increased above 16 switches. POX IPD varied between 10-30ms depending on topology, resulting in uneven delays. In tree topologies, RYU's IPD gradually increased from 2-4ms with depth, whereas NOX and POX tripled. This shows that RYU outperforms NOX and POX in terms of early flow setup.

## 5.7 Round Trip Time

RYU regularly achieved ultra-low RTT of 1-2ms regardless of scale, emphasising its control performance. NOX RTT was also low (<5ms) for small topologies but skyrocketed to over 50ms for large networks. POX showed greater variability, ranging from 5-20ms RTT depending on the test, with no discernible trend. Linear topologies caused the greatest RTT volatility in NOX and POX, but star/tree topologies performed better. RYU was unaffected by topology type and performed well in all cases.

## 5.8 Why RYU Performed best?

Here are the key reasons why RYU performed the best across all the parameters evaluated in this study.

### 5.8.1 Architecture

To provide high performance SDN control activities, RYU employs a multi-threaded asynchronous architecture. A thread pool controls a fixed number of worker threads at the core. These worker threads execute independent event loops indefinitely to handle I/O asynchronously and without blocking.

**Figure 5.1: RYU Architechture**

An incoming event, such as receiving a PacketIn message from the switch, is placed to a central task queue. The worker threads poll this queue on a regular basis to retrieve and process jobs in a non-blocking manner. This enables numerous tasks to run concurrently without waiting.

In RYU, each I/O action is implemented as a non-blocking coroutine utilising the Eventlet coroutine library. Eventlet makes use of greenlets to simulate parallel execution even when only one thread is active at any given time. When a thread performs an I/O call, such as sending/receiving a message, it automatically yields to allow other threads to run.

This enables RYU to achieve levels of concurrency comparable to real multiprocessing. With thousands of greenlets running concurrently, all I/O operations like as flow setups, packet handling, and so on can occur in parallel without interfering with one another.

Non-blocking callbacks for completed operations are executed within the original thread's greenlet context. This keeps the asynchronous nature while without interfering with task ordering. Even blocking functions like sleep are implemented collaboratively to prevent stalling the thread.

RYU's architecture scales inherently to distribute demand among CPU cores by distributing jobs dynamically over the thread pool. It easily responds to changes in workload to maintain good performance regardless of network size or traffic. As a result, it is a very stable and efficient control plane solution.

### 5.8.2 FLow Management

RYU uses high-performance Cuckoo hash tables for flow lookups and entries. Cuckoo hashing enables fast O(1) lookup times on average by utilising numerous hash tables and collision resolution algorithms. This fast solution ensures that incoming packets are quickly matched to flows.

Interval trees are used in conjunction with Cuckoo hashes to handle wildcard and ranging IP prefix matching quickly. Interval trees save IP prefix ranges in order to perform **O(log n)** lookups for IP address-based matches. This expedites the processing of aggregated flows. RYU maintains an LRU cache of recently accessed flow items in memory. Before accessing the slower hash tables, the cache is examined first on any flow table operation such as insert, remove, or modify. This reduces average flow setup times by avoiding costly lookups.

When a fresh flow arrives, RYU prioritises its processing. This ensures that flow arrangements have minimum delays even during peak traffic periods. For efficient transmission to switches, several concurrent setup requests are also batched together.

Instead of serial individual updates, batching flows take advantage of the asynchronous architecture by allowing several flow adjustments to be communicated concurrently in one transmission. This significantly reduces the per-flow processing overhead.

These optimizations in RYU's flow architecture allow it to constantly outperform rival controllers in important measures such as average flow setup delay, jitter, and maximum sustained flows-per-second, even under large workloads.

### 5.8.3 Packet Processing

When a controller receives a PacketIn message from switch, the packet is added to RYU's central task queue. The worker threads poll this queue on a regular basis to retrieve packets for parallel processing without stalling.

Using non-blocking I/O calls, each PacketIn task is done individually. The worker thread will examine the packet headers, look for the corresponding flow using concurrent hash table operations, and take the necessary forwarding actions, among other things.

While one thread is processing a packet, other threads are free to pick up additional PacketIn tasks from the queue or do other Critical Control Plane operations such as installing new flows.

This enables all packet handling processes to run in parallel rather than sequentially in queues. Using simply the thread pool, RYU can reach line-rate packet processing speeds comparable to multiprocessor systems.

Even during traffic surges, packets are not queued for long before being selected for concurrent processing by a worker. This prevents dropped packets and minor queueing delays from entering the data plane.

RYU can keep up with traffic demands even during flash crowds by automatically scaling the thread pool based on workload. As packet rates increase, more threads are created to take advantage of additional CPU cores.

This asynchronous packet processing is fundamental to RYU's ability to deliver wire-speed performance, perfect throughput, and sub-millisecond packet latencies in the face of enormous packet loads and variable network conditions.

### 5.8.4 Low Overhead

1. **CPU Overhead** Asynchronous routes in RYU allow CPU-intensive processes like hashing to use many logical cores concurrently without incurring significant context switching penalties. To reduce per-packet processing expenses, operations are additionally batched, pipelined, and cached where possible. In comparison to Python interpreters used in other controllers, the framework itself has low looping/dispatching overhead.

2. **Memory Overhead** Unlike databases, flow table entries are stored in an efficient in-memory hash structure that does not bloat. Instead of costly object creation/teardown, packet buffers are pooled and reused via offsets. Object representation is optimised by using primitives such as integers rather than Python objects.

3. **I/O Overhead** Async non-blocking calls are used for network I/O with switches to avoid thread blocking. Streaming flow/config updates in a single transmission lowers socket/serialization costs. Caching eliminates the need for repeated database lookups and message encoding/decoding.

4. **Latency Overhead** To avoid stalls/waits, almost all operations are non-blocking. Priority queuing and pipelining are two optimisations that reduce the amount of time spent in lineups and processing steps. For consistent low latency, the async worker paradigm scales intrinsically dependent on CPU capability.

5. **Scalability** RYU's asynchronous multi-threaded architecture is built from the ground up to be extremely scalable. A thread pool in the core dynamically assigns worker threads based on available CPU cores. This enables control plane processing to readily expand up to hundreds of concurrent threads, allowing modern multicore servers to be fully utilised. As more tasks occur as a result of rising traffic loads, new threads from the pool can be promptly provisioned to meet the demand. Threads are also returned when loads are reduced. This elastic scaling ensures that processing throughput remains consistent even when work volumes fluctuate dramatically over time. RYU's non-blocking I/O and asynchronous task-based design allow it to scale up its performance linearly with the number of concurrent operations. Whereas increased parallelism in traditional threaded systems would impair throughput owing to locking, RYU avoids all blocking to retain performance advantages. Task-generated load is also carefully divided among threads via a shared work queue. This ensures that no single thread becomes a bottleneck. Flow operations scale to maintain wire-speed performance regardless of table size when combined with caching, optimised data structures, and batching. Furthermore, by decoupling components such as the flow managers, RYU's basic control system stays lightweight and extremely adaptable. By distributing such components over numerous clustered servers, horizontal scaling is facilitated. Effectively, RYU can scale out to control networks with millions of flows and switches only limited by available hardware resources.

6. **Stability** Years of development and bug fixing has resulted in a robust and hardened codebase. RYU delivered rock-solid, jitter-free performance throughout all tests.

CHAPTER 6

# Conclusion

## 6.1 Conclusion

This research compared and evaluated the performance of three prominent SDN controllers: NOX, POX, and RYU. Using sophisticated simulation tools, a diverse set of seven performance parameters were examined across various network topologies and scales. The results provide useful information about each controller's capabilities and limits.

Overall, RYU demonstrated the most constant performance with minimal fluctuation as network sizes and loads increased. It had low flow setup latencies, short packet delays, and consistent bandwidth and throughput. When POX and NOX were subjected to larger topologies, their latencies increased and their performance degraded.

In terms of scalability, RYU maintained responsiveness for networks with more than 512 switches and 2048 hosts, indicating that it is well-suited for large-scale deployments. POX and NOX ran into scalability concerns after reaching 64 and 32 hosts, respectively. Topology influenced controller behaviour as well, with tree networks causing longer latencies than linear or star arrangements. RYU, on the other hand, displayed good flexibility across all topologies evaluated.

As a result, our study confirms RYU as a high-performing and scalable SDN controller option for both small and large operational networks. It offers network operators significant help in picking the optimum controller based on deployment considerations.

## 6.2 Future Work

There are several avenues for future work to build upon the insights from this study.

1. **Distributed SDN Controllers**

   Using controllers such as ONOS and Opendaylight to compare the performance of distributed SDN architectures to that of centralised ones. How do they handle large networks with thousands of nodes?

2. **Dynamic Traffic Simulation**

   Injecting realistic Internet traffic patterns into emulated networks to analyse controller behaviour under dynamic loads as opposed to static testing. What effect do burstiness and traffic spikes have on performance?

3. **Hardware Resource Utilization**

   CPU, memory, disc, and network use of controllers running on physical servers are profiled. How effectively do they make use of computational resources?

4. **Machine Learning Models**

   Using the metrics measured in this study, we are developing ML models to predict controller performance for any given network topology and scale.

5. **Software Upgrades**

   Portions of this evaluation will be repeated on a regular basis to evaluate the evolution of controller codebases over time as new features are added and issues are corrected.

6. **Hybrid Controller Models**

   To achieve optimal scalability, we are investigating hybrid/hierarchical SDN designs that combine the best aspects of centralised and distributed techniques.

7. **Mobile Networks**

   SDN controllers are being evaluated in the context of 5G networks that handle billions of IoT devices. What effect do mobility, low-latency, and other 5G needs have on performance?

This thesis provided a foundational performance benchmark, and additional research is needed to improve our understanding of SDN controllers and optimise their use in current networks. The topics covered here can assist developers and network engineers in furthering the state of the art in Software Defined Networking.

CHAPTER 7

# Recommendations

## 7.1 Controller Selection

1. The selection of an SDN controller is significantly influenced by network size, topology, traffic patterns, and expansion plans. Controllers having various topologies, such as proactive, reactive, or hybrid, are appropriate for various use cases.
2. For networks up to 500 switches, single-threaded controllers like POX, NOX can be used but their performance degrades significantly beyond that.
3. Event-driven controllers with highly flexible designs and support for bespoke application development, such as RYU, are required for dynamically changing networks.
4. Critical infrastructure necessitates highly available and dependable controllers, such as Ryu clusters, that can operate in active-standby or active-active mode.

## 7.2 Topology Specific Considerations

1. Because of MAC learning, ARP resolution, and hierarchical routing, tree topologies put extra strain on the control plane. Controllers that have been optimised for these protocols via specialist modules perform better.
2. Linear topologies provide low overheads, but scaling beyond tens of switches necessitates distributed controllers to avoid controller bottlenecks.
3. Control traffic is concentrated at the central switch in star topologies. Controller burden can be reduced by caching commonly used flows and transferring control to switches.

## 7.3 Hardware Resource Planning

1. Multi-core CPU servers are required to maximise throughput in distributed controllers by leveraging multi-threading.

2. SSDs outperform HDDs in terms of I/O performance for databases used by distributed controllers.

3. Sufficient memory allows for larger flow tables to be maintained for faster lookups and eliminates swapping. Based on test results, a minimum of 16GB RAM is suggested.

4. Network connections with dedicated high bandwidthTo improve Quality of Service (QoS), separate control traffic from data traffic. Between switches and controllers, 10GbE or greater is desirable.

## 7.4 Benchmarking and Profiling

1. Before deployment, benchmarking tools such as CBench, PktBlaster, and Ostinato should be used to evaluate controllers under realistic workloads.

2. Cache hit ratios, flow setup durations, and thread contention all provide useful information for optimisation. SQLite Analyzer and Java Mission Control are tools that help with low-level performance analysis.

## 7.5 Monitoring and Upgrades

1. During the testing phase, baseline metrics must be developed. Trends in production deployments must be closely monitored using technologies such as Telegraf and Influxdb.

2. Support lifespan is affected by community activity and the release cycle. Long-term support agreements are required for mission-critical deployments.

# References

M. M. Elmoslemany, A. S. T. Eldien, and M. M. Selim, "Performance analysis in software defined network controllers," in *2020 15th International Conference on Computer Engineering and Systems (ICCES)*, 2020, pp. 1–6.

J. Ali, S. Lee, and B.-h. Roh, "Performance analysis of pox and ryu with different sdn topologies," April 2018, pp. 244–249.

M. A. Khan, M. A. Shah, F. Z. Raja, and H. A. Khattak, "A novel technique of dynamic resource allocation in software defined network," in *2019 15th International Conference on Emerging Technologies (ICET)*, 2019, pp. 1–5.

I. Bholebawa and U. Dalal, "Performance analysis of sdn/openflow controllers: Pox versus floodlight," *Wireless Personal Communications*, vol. 98, January 2018.

Y. Li, X. Guo, X. Pang, B. Peng, X. Li, and P. Zhang, "Performance analysis of floodlight and ryu sdn controllers under mininet simulator," *2020 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, 2020.

J. Silva, F. Silva, E. Neto, M. Lemos, and A. Venancio Neto, "Benchmarking of mainstream sdn controllers over open off-the-shelf software-switches," *Internet Technology Letters*, vol. 3, p. e152, February 2020.

S. Mostafavi, V. Hakami, and F. Paydar, "Performance evaluation of software-defined networking controllers: A comparative study," vol. 2, pp. 63–73, October 2019.

N. H. Thanh, N. N. Tuan, D. A. Khoa, L. C. Tuan, N. T. Kien, N. X. Dung, N. Q. Thu, and F. Wamser, "On profiling, benchmarking and behavioral analysis of sdn architecture under ddos attacks," *J. Netw. Syst. Manage.*, vol. 31, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1007/s10922-023-09732-5

L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani, "Sdn controllers: A comprehensive analysis and performance evaluation study," *ACM Computing Surveys*, vol. 53, pp. 1–40, December 2020.

B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868447.1868466

M. Dissanayake, W. A. L. Kumari, and A. Udunuwara, "Performance comparison of onos and odl controllers in software defined networks under different network typologies," 07 2021.

A. Shirvar and B. Goswami, "Performance comparison of software-defined network controllers," in *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2021, pp. 1–13.

S. Askar and F. Keti, "Performance evaluation of different sdn controllers: A review," 05 2021.

S. Bhardwaj and S. Panda, "Performance evaluation using ryu sdn controller in software-defined networking environment," *Wireless Personal Communications*, vol. 122, 01 2022.

R. Wazirali, R. Ahmad, and S. Alhiyari, "Sdn-openflow topology discovery: An overview of performance issues," *Applied Sciences*, vol. 11, no. 15, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/15/6999

M. T.BAH, A. Azzouni, M. Nguyen, and G. Pujolle, "Topology discovery performance evaluation of opendaylight and onos controllers," in *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2019, pp. 285–291.

A. Tello and M. Abolhasan, "Sdn controllers scalability and performance study," 12 2019, pp. 1–10.

S. Zhihao, H. Wu, and K. Wolter, "Performance evaluation of the control plane in software defined networks," 03 2019, pp. 171–174.

A. H. M. Hassan, A. M. Alhassan, and F. Izzeldean, "Performance evaluation of sdn controllers in ofnet emulation environment," in *2019 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, 2019, pp. 1–6.

S. Lee, J. Ali, and B.-h. Roh, "Performance comparison of software defined networking simulators for tactical network: Mininet vs. opnet," in *2019 International Conference on Computing, Networking and Communications (ICNC)*, 2019, pp. 197–202.

R. Kumaraswamy and A. Rajendra, *Analysis of POX and Ryu Controllers Using Topology Based Hybrid Software Defined Networks*, 01 2020, pp. 49–56.

E. Marin, B. Nicola, and M. Conti, "An in-depth look into sdn topology discovery mechanisms: Novel attacks and practical countermeasures," 11 2019, pp. 1101–1114.

## References

T. Shozi, S. Dlamini, P. Mudali, and M. Adigun, "An sdn solution for performance improvement in dedicated wide-area networks," 03 2019, pp. 1–6.

N. Gupta, M. S. Maashi, S. Tanwar, S. Badotra, M. Aljebreen, and S. Bharany, "A comparative study of software defined networking controllers using mininet," *Electronics*, vol. 11, no. 17, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/17/2715

P. J. Research and P. Raj, "Topology-based analysis of performance evaluation of centralized vs. distributed sdn controller," in *2018 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC)*, 2018, pp. 1–8.

K. Basu, M. Younas, and F. Ball, "Performance comparison of a sdn network between cloud-based and locally hosted sdn controllers," 03 2018.

M. Abdullah, N. Awad, and F. w. Hussein, "Performance evaluation and comparison of software defined networks controllers," *International Journal of Computing  Network Technology*, vol. 2, p. 45, 11 2018.

J. M. S. Vilchez and D. E. Sarmiento, "Fault tolerance comparison of onos and opendaylight sdn controllers," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 277–282.

S. Moazzeni, M. Khayyambashi, N. Movahhedinia, and F. Callegati, "On reliability improvement of software-defined networks," *Computer Networks*, vol. 133, pp. 195–211, 03 2018.

APPENDIX A

# Achievements

1. Rahman, H. M. U., Bahoo, G., Zafar, L., Al-Oqily, I., Khattak, H. A., Ahmad, A. (2023, May). Empirical Performance Evaluation of Open Source SDN Controllers in different Network Topologies. In NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium (pp. 1-6). IEEE.

APPENDIX B

# Appendix A

The separate numbering of appendices is also supported by LaTeX. The *appendix* macro can be used to indicate that following chapters are to be numbered as appendices. Only use the *appendix* macro once for all appendices.