Improving The Performance Of Software Defined Networks Using Dynamic Flow Installation And Management Techniques

by

Philippos Isaia

Submitted in partial fulfilment
of the requirements for the award of

Doctor of Philosophy
of
Loughborough University

5th June 2018

Abstract

As computer networks evolve, they become more complex, introducing several challenges in the areas of performance and management. Such problems can lead to stagnation in network innovation. Software Defined Networks (SDN) framework could be one of the best candidates for improving and revolutionising networking by giving the full control to the network administrators to implement new management and performance optimisation techniques.

This thesis examines performance issues faced in SDN due to the introduction of the SDN Controller. These issues include the extra delay due to the round-trip time between the switch and the controller as well as the fact that some packets arrive at the destination out-of-order.

We propose a novel dynamic flow installation and management algorithm (OFPE) using the SDN protocol OpenFlow, which preserves the controller to a non-overloaded CPU state and allow it to dynamically add and adjust flow table rules to reduce packet delay and out-of-order packets. In addition, we propose OFPEX, an extension to OFPE algorithm that includes techniques for managing multi-switch environments as well as methods that make use of the packets interarrival time in categorising and serving packet flows. Such techniques allow topology awareness, helping the controller to install flow table rules in such a way to form optimal routes for high priority flows thus increasing network performance. For the performance evaluation of the proposed algorithms, both hardware testbed as well as emulation experiments have been conducted. The performance results indicate that OFPE algorithm achieves a significant enhancement in performance in the form of reduced delay by up to 92.56% (depending on the scenario), reduced packet loss by up to 55.32% and reduced out-of-order packets by up to 69.44%.

Furthermore, we propose a novel placement algorithm for distributed Mininet implementations which uses weights in order to distribute the experiment components to the appropriately distributed machines. The proposed algorithm uses static code analysis in order to examine the experimental code as well as it measures the capabilities of physical components in order to create a weights table

which is then used to distribute the experiment components properly. The performance results of the proposed algorithm evaluation indicated reductions in delay and packet loss of up to 65.51% and 86.35% respectively, as well as a decrease in the standard deviation of CPU usage by up to 88.63%. These results indicate that the proposed algorithm distributes the experiment components evenly across the available resources.

Finally, we propose a series of Benchmarking tests that can be used to rate all the available SDN experimental platforms. These tests allow the selection of the appropriate experimental platform according to the scenario needs as well as they indicate the resources needed by each platform.

Keywords: Software Defined Networking, OpenFlow, Dynamic Flow Installation, Mininet, POX

Acknowledgements

I would like to thank many individuals in the Computer Science Department of Loughborough University because without their kind support, help and encouragement the successful completion of this report would not have been possible.

I would like to express my deepest appreciation to my research supervisor Dr Lin Guan for her contribution in stimulating suggestions, constant encouragement, and useful feedback that helped me a lot to coordinate my project work.

Special thanks go to Dr Peter Bull for his guidance and valuable feedback throughout the execution of the experimental work and especially for his advice in report writing.

Last but not least, many thanks go to Dr Iain Phillips for accepting to be my annual reviewer throughout my PhD, as well as helping me to set up the virtual environment for experimentation.

Publications

Philippos Isaia, Lin Guan. "Performance Benchmarking of SDN Experimental Platforms", *2016 IEEE NetSoft Conference and Workshops (NetSoft 2016), Seoul, 2016*

Philippos Isaia, Lin Guan. "Distributed Mininet Placement Algorithm for Fat-Tree Topologies", *25th IEEE International Conference on Network Protocols 2017 (ICNP 2017), Workshop on Software Defined Networking and Network Function Virtualization Performance (PVE-SDN), 2017*

Contents

List of Figures

List of Tables

List of Abbreviations

APD	Average Ping Delay
API	Application Programming Interface
AQM	Active Queue Management
ARED	Adaptive Random Early Detection
AS	Autonomous System
AVQ	Adaptive Virtual Queue
BoS	Bottom of Stack
CBT	Class Based Thresholds
CPU	Central Processing Unit
DOT	Distributed OpenFlow Testbed
DDR	Double Data Rate
DSCP	Differentiated Services Code Point
DSRED	Double Slope Random Early Detection
DT	Drop Tail
EAVQ	Stable Enhanced Adaptive Virtual Queue
eBGP	external Border Gateway Protocol
ECN	Explicit Congestion Notification
FABA	Fair Adaptive Bandwidth Allocation
FIB	Forwarding Information Base
FRED	Flow Random Early Detection
GUI	Graphical User Interface
HRED	Hyperbola Random Early Detection
I-SID	Backbone Service Instance Identifier
I/O	Input / Output
iBGP	internal Border Gateway Protocol

ICMP	Internet Control Message Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IT	Information Technology
LRED	Loss Ratio Based Random Early Detection
LTS	Long Term Support
LUBA	Link Utilization Based Approach
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
MRED	Modified Random Early Detection
NRFR	No Response Failure Rate
NAT	Network Address Translation
ND	Neighbour Discovery
NetFPGA	Networks Field-Programmable Gate Array
NIB	Network Information Base
NIC	Network Interface Controller
NV	Network Virtualization
NFV	Network Functions Virtualization
ONF	Open Networking Foundation
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCP	Priority Code Point
PR	Predicate Routing
QoS	Quality of Service
QVARED	Queue Variation Adaptive Random Early Detection
RAM	Random Access Memory
RaQ	Rate-based and Queue-based
RAQM	Rate-based Active Queue Management
RARED	Refined Adaptive Random Early Detection
RCP	Routing Control Platform
REAQM	Rate-based Exponential Active Queue Management
RED	Random Early Detection

REM	Random Exponential Marking
RFC	Request For Comments
RIB	Routing Information Base
RPC	Proportional Rate-based Control
RR	Route Reflector
RTT	Round Trip Time
SATA	Serial Advanced Technology Attachment
SAVQ	Stabilized Adaptive Virtual Queue
SCTP	Stream Control Transmission Protocol
SDN	Software-Defined Networking
SelfRED	Self Configuring Random Early Detection
SFB	Stochastic Fair Blue
SHRED	Short Lived Flow Friendly Random Early Detection
SNAC	Simple Network Access Control
SRED	Stabilized Random Early Detection
StoRED	Stochastic Random Early Detection
SVB	Stabilized Virtual Buffer
TCP	Transmission Control Protocol
TM	Traffic Matrix
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine
WAN	Wide Area Network

Chapter 1

Introduction

Conventional enterprise network architectures typically consist of three tiers (core, aggregation and edge) of Ethernet switches arranged in tree style topologies [6]. Such approaches can provide an excellent performance and efficiency level in a client-server computing environment. Nowadays, dynamic computing and storage needs, as well as more advanced data centre topologies, are forcing out the domination of client-server computing. According to a research by the Enterprise Strategy Group (ESG) [7], 63% of enterprises are moving towards more advanced data centres to catch up with the emerging challenges and needs of today's networking. These challenges can be summarised into four areas, *(a)* changing of traffic patterns, *(b)* rise of cloud services, *(c)* IT consumerization and *(d)* bandwidth exponential growth.

A forecast research taken by Cisco [1, 2, 8] clearly indicates the importance and the extent of the aforementioned challenges. The east-west annual traffic (i.e. traffic within the data centre) was 1.8 zettabytes (ZB) in 2011 and is expected to reach 15.3ZB by 2020, thus increasing nine times in a period of nine years. On the other hand, the annual north-south traffic (i.e. traffic crossing the Internet) is about to reach 2.3ZB by 2020 from 0.37ZB it was in 2011. This not only indicates the bandwidth exponential growth but also the difference between east-west and north-south traffic which will become almost seven times greater by 2020. Furthermore, Cisco states that global cloud traffic will reach 14.1ZB by 2020, compared to 0.8ZB in 2011. These forecast clearly indicates the importance of the challenges of today's networking mentioned above. It is clear that traffic

patterns are changing, with more traffic moving within the data centre compared to traffic outside the data centre. The exponential growth in traffic as well as the differences between traffic types and directions indicate the rise of cloud services. These changes, together with stagnation, are forcing networks into a less robust and harder to handle state with limitations such as *complexity*, *scalability* and *vendor dependency*.

Networking vendors have been working on a number of innovations in order to overcome emerging problems but there are still several problems that they cannot solve or they are reluctant to solve such as the unification of various methodologies and platforms. A research group at Stanford University while working on a project to segment their production network in a way that both production and research traffic can be transferred within the same network without one affecting the other, came with the idea of OpenFlow [9]. Several big organisations began investing in OpenFlow as one of the protocols that can help solving the emerging networking problems. OpenFlow is a Software Defined Networking (SDN) [4] protocol, essentially allowing the implementation of the SDN model. Physical or virtual networking equipment consists of a control and a data plane. What the SDN model proposes is the decoupling of these two planes, and merging the control plane of several networking devices into a central control entity. This allows network centralization as well as replacement of hard-wired instructions to an open software. OpenFlow is the communication protocol between the centralized control entity and the data plane.

The SDN model has the potential to address several of the challenges created by the evolution of networks. SDN can virtualize the networks using the centralization of the control plane, the open API's as well as the ability to program the network using software. As a result, it has the ability to facilitate the emerging requirements as well as control and extend the complex infrastructure using virtual network segments. SDN's biggest benefits are the *secure network segmentation*, *traffic engineering* and *network provisioning abilities*, all resulting from the effective centralisation as well as the open software handling the network implementation.

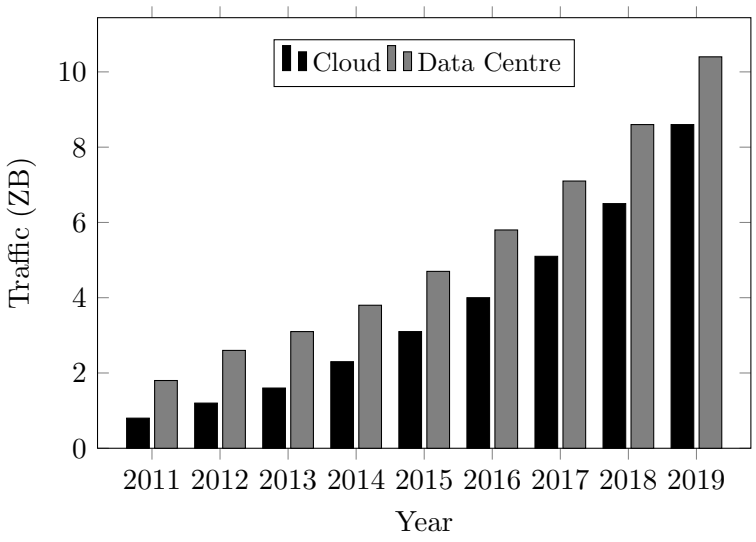


Figure 1.1: Cisco Data Centre and Cloud Traffic Forecast (*Sources: [1] and [2]*)

1.1 Motivation

Currently, SDN is an emerging networking model with its own limitations. These limitations are the direct impact of the key changes in the architecture of SDN. In SDN, network devices (mainly switches) control planes are managed by controllers through what is known as the controller-to-switch link. Controllers are servers that run a process tree responsible for managing a network device. One of the major drawbacks of such an approach is the fact that a controller runs on top of a conventional Operating System (OS), meaning that it will have to share processing resources with all the OS processes. A controller will create a number of processes, which will wait for the OS scheduler to assign them some processing time. If the OS scheduler does not assign the appropriate processing time, or assign it but not in the correct priority then the network will face performance decrease. A conventional OS will be unable to give higher priority to processes that come from high priority flows, resulting in a decrease of Quality of Service (QoS) [10, 11].

In addition, if the controller gets overloaded, the network will experience a decrease in performance in the form of delay and packet loss increase. An overloaded controller can be very easily become unresponsive causing a disastrous situation. The SDN architecture is highly depended on the controller, meaning that without a controller it will be unable to have any dynamic control over traffic flows. The switch may be pre-programmed to work with existing flows or in a stand-alone

mode, but all the flows without pre-existing rules or wildcards will be rejected or handled incorrectly.

Furthermore, the way the switch handles flows can cause some serious limitations. In an SDN implementation, using OpenFlow, the switch handles the packets it sends to the controller using the First In First Out (FIFO) method. As a result, when there is a queue of several packets from various flows waiting to be send to the controller, there is no flow prioritisation as to which packet from which flow reaches the controller first. Finally, the controller-to-switch link also affects the overall performance of an SDN implementation and under the appropriate conditions it can become the bottleneck of the network. These limitations can cause serious QoS drawbacks and in some extreme conditions they can cause a disaster in the network.

Due to the nature of the SDN model and the OpenFlow protocol, altering any parts of an SDN/OpenFlow implementation will result in losing the benefits that it brings as a model to the networking area. Instead of changing the whole model, what will benefit such an implementation is a mechanism that will keep the individual components in a healthy state combined with an algorithm that will be able to handle flows and provide better QoS in the form of less delay, packet losses and out-of-order packets.

1.2 Aim and Objectives

The aim of this research is to improve the overall SDN performance through the design and implementation of performance enhancement algorithms. The proposed OFPE algorithm will be implemented on the controller, and will provide better performance, controller stability and prevent the controller from overloading. Furthermore, using modules for topology awareness, it will increase the performance in multi-switch environments. This will be achieved using flow installations for dynamically created routes. Moreover, the proposed algorithm will use techniques to prevent situations such as a full flow table and controller-to-switch link bottleneck.

The objectives of this research are summarised as follows:

1. To identify the management techniques used by OpenFlow in heavy traffic

networks and produce possible scenarios in which these techniques lack in performance.

2. To create a methodology that can improve the current performance of OpenFlow during heavy traffic periods.

3. To implement an algorithm that can improve the performance of OpenFlow networks during periods of heavy traffic.

4. To identify the individual events that can decrease the performance of the algorithm and define ways to prevent them.

1.3 Original Contributions

The original contributions of this thesis are listed below:

A. Propose a novel OpenFlow Performance Enhancement Algorithm (OFPE) implemented in the OpenFlow controller which uses Dynamic Flow Installation and Management. The proposed algorithm achievements are:

   (a) Prevents controller-to-switch link failure due to overload.

   (b) Prevents controller from failure due to overload.

   (c) Keeps packet delay and loss at an acceptable level.

   (d) Reduces the amount of out-of-order packets arriving at the destination.

B. Extend the proposed OFPE Algorithm by using packets Inter-Arrival time. This allows the algorithm to be more independent due to the fact that several parameters that needed to be manually added to the OFPE algorithm are now calculated by the algorithm itself. The proposed OFPE extended algorithm achievements are:

   (a) Calculate the appropriate timeout times used in the flow table.

   (b) Better prediction of traffic load during relevant time periods.

   (c) Prevents failure due to overload of both the controller and the controller-to-switch link.

(d) Reduces the amount of both delay and packet loss as well as it reduces the number of out-of-order packets.

C. Propose a new placement algorithm for distributed Mininet implementations which uses a weights table in order to place the experiment components based on the available resources. The proposed placement algorithm achievements are:

(a) Static code analysis of Mininet scenario code.

(b) Measures the physical link capacities and forms bins based on the available resources.

(c) Distributes the components (hosts and links) evenly across the available workers (reduces CPU usage standard deviation).

(d) Reduces the amount of both packet loss and delay which is caused by the limited resources.

D. Propose a series of Benchmarking tests that can be used to rate all the available SDN Experimental Platforms. The proposed benchmarking tests achievements are:

(a) Allows for selection of the appropriate experimental platform according to the scenario needs.

(b) Gives the resources needed by each experimental platform (in the form of CPU and RAM).

(c) Shows the multi-core capabilities and the efficient distribution of load of each experimental platform.

Note: Up to this date of writing this thesis and to the best of the author's knowledge, the proposed algorithms and the novel work presented in this thesis is not a published knowledge.

1.4 Thesis Outline

The rest of this thesis is organised as follows:

Chapter 2 provides an overview of SDN and OpenFlow. It gives details for the SDN model as well as the the OpenFlow protocol. Several advances in the area of SDN are discussed as well as a comprehensive analysis on the individual parts of the SDN model.

Chapter 3 provides details about the proposed OpenFlow Performance Enhancement Algorithm Using Dynamic Flow Installation And Management.

Chapter 4 Provides an extension to the proposed OpenFlow Performance Enhancement Algorithm, with the use of packets interarrival time.

Chapter 5 Provides details about the proposed Placement Algorithm for Distributed Mininet implementation with optimisation in Fat-Tree Topologies.

Chapter 6 Uses the proposed placement algorithm in order to test the proposed performance enhancement algorithm in a distributed Mininet implementation.

Chapter 7 Provides a series of Benchmarking tests that can be performed on SDN Experimental Platforms in order to rate them. In addition, the proposed tests are performed on Mininet Emulator and full analysis of the results is provided.

Chapter 8 Deals with the performance analysis and evaluation of OpenFlow-enabled hardware and software which can be used for the creation of experimental environments.

Chapter 9 Provides the conclusions as well as some future recommended work.

Chapter 2

An Overview Of Software Defined Networking

2.1 Introduction

Traditional IP networks have served traffic for decades but as demand grows and traffic patterns are changing, these networks are becoming more complex and harder to manage [12]. Network engineers have to manage a constantly changing state of networking traffic and adapt to changes fast and accurately. Automatic adaptation is almost non-existent and together with the fact that control and data planes are bundled inside the networking devices, manual low-level reconfiguration of networking devices in constantly needed. These are some of the most important reasons causing lack of networking infrastructure innovation and evolution.

SDN is an emerging networking model that has the potential in solving these problems by transferring the control plane to a separate machine called the controller. The SDN model, transforms the networking equipment (switches & routers) into forwarding devices and achieves centralisation by placing the control plane of several networking equipment into one unified controller. This chapter presents a comprehensive literature review of the SDN model as well as several innovations and use cases. In addition, it goes through the OpenFlow protocol with comparison of several OpenFlow versions as well as several controllers. The sections of this chapter can be summarised as follows:

- Section 2.2 (Programmable Networks): This section presents all the cen-

tralised network control systems existed prior to SDN. Through the analysis, which follows a chronological order, the evolution of centralised network control systems can be seen.

- Section 2.3 (OpenFlow): This section presents a literature review of the first SDN protocol, OpenFlow. Several OpenFlow characteristics as well as comparison of its three most stable versions are presented.

- Section 2.4 (Software Defined Networking): This section introduces the concept of SDN.

- Section 2.5 (SDN Controllers): This section provides a brief overview of OpenFlow controllers. It introduces some behaviours the controllers share and also describes some of the most known controllers.

- Section 2.6 (Benchmarking Simulation and Emulation Environments): This section provides some benchmarkings performed on a number of simulation and emulation environments used for SDN experimentation.

- Section 2.7 (OpenFlow Related Projects): This section presents numerous OpenFlow related projects as well as outlines their benefits and uses. Through the OpenFlow related projects, one can see the effect SDN that has on networks architectures as well as the benefits it can bring to networking.

- Section 2.8 (Summary and Discussions): This section gives a summary of the overview of SDN, indicating some of our conclusions and important aspects of the SDN model.

2.2 Programmable Networks

The idea of having an easily programmable network emerged as early as mid-1990s. The Active Networks research group [13] explored several alternative ideas in the area of networking, and one of their major visions was to create a programming interface that can give access to network nodes resources such as queues and processing. This allowed programmers to construct custom functionality and

policies directly on the networking equipment. At the time, researchers thought that such direct programmability can solve problems and evolve networks. More specifically, it can lower the barrier to innovation, provide network virtualisation as well as provide a unified architecture for middlebox orchestration [14]. With today's terms, they have envisioned a Network API with functionality similar to SDN. Unfortunately, active networking did not make an impact due to the fact that it did not provide a practical performance and security level. Furthermore, the network API targeted end-users who had to create Java code and transfer it together with data packets in order to use the API.

Due to Internet's flourish and the rapid growing of backbone networks in the early 2000's, ISPs were having network management and reliability issues. In order to overcome their problems, they relied on two innovations. The first was to create an open interface between the control and the data planes. This was achieved in the Forwarding and Control Element Separation (ForCES) [15] project as well as with the use of the Linux Netlink [16]. Secondly, they have created a logically centralized network control method such as the Routing Control Platform (RCP) [17] discussed in section 2.2.1.

2.2.1 Routing Control Platform and 4D Architecture

RCP, using the existing Border Gateway Protocol (BGP) [18], proposed a centralised approach to solve several of the networking management problems faced at the time. RCP uses a central server which communicates with all the routers that it is connected to in order to collect external BGP (eBGP) route updates and to compute BGP routes on behalf of all the routers in the Autonomous System (AS). This approach eliminates the need for full mesh connections, but only requires each router to have one connection with the central server, thus providing better scalability. Overall, RCP shows the advantages of centralisation in the area of internal BGP (iBGP) [18] routes computation, and that it is practical to have a centralised system build with reasonable scalability performance. On the other hand RCP has some limitations: **(a)**, the scalability of the central server and **(b)**, replicated servers and routers will achieve consistent decisions during steady state, but not the same happens with transient state.

In 4D Architecture [19–21], researchers argue that the main reason for the network being fragile and difficult to manage is the complexity of the control and management planes of today's network. It is because the control logics are coupled with packet forwarding functions distributed among elements in the network. In order to solve such problems, 4D proposed four planes, Decision, Dissemination, Discovery and Data. 4D completely separates network's decision logic from distributed protocols that handle basic packet forwarding. Network-wide objectives are specified in the decision plane and then translated by specific algorithms into actual direct control configurations for routers and switches, forming the data plane. Data plane is responsible for basic data packet processing functions such as packet forwarding, packet filtering, packet queuing or address translating. The dissemination plane is a robust and efficient communication mechanism between decision and data plane. The discovery plane is responsible for discovering the physical components in the network and creates logical identifiers to represent them as well as collects measurement data to construct the network-wide view for the decision plane to achieve its objectives. Overall 4D was a pioneer work which provided the idea that a centralised decision element controlling the whole network is more flexible. This is due to the fact that new functionality can be centrally programmed instead of having to create a new distributed algorithm. The only drawback was the fact that the network will end up with one point of failure. Adequate resilience could be achieved by applying standard replication techniques to the central decision making element. These replication techniques are completely decoupled from the network control algorithms, so they do not impede application innovation. The goal of 4D was to control forwarding and thus their network view only included the network infrastructure.

2.2.2 SANE, Ethane and Tesseract

SANE [22] inspired by 4D, argues that in the enterprise network environment, security is critical, centralised control is normal and uniform consistent policies are important. Network security usually involves actions on both routing and access control. According to SANE this is problematic because these coupled actions need to be coordinated. It proposes a clean-slate approach by separating control plane

from data plane. Moreover, it also centralises the routing and security policies and uses a separate channel to carry control plane traffic between switches and the central controller by spanning the tree rooted at the central controller. The difference of SANE from 4D is that it does not allow communication between end hosts; therefore the security policies are achieved by controlling whether or not the capabilities should be issued, and not by packet filters or firewalls like 4D does. SANE comes with three limitations:

1. Because all data plane traffic is routed by source route issued from the central controller, end hosts need to be modified to at least have a proxy to translate IP packets to packets using source route. This brings overhead in processing each packet, increasing latency and also prevents plug-and-play ability.

2. The encryption/decryption computation for the secure source route requires a large amount of computing power, increasing the queuing delay.

3. Although SANE central controller can handle tens of thousands of nodes, in order to achieve that, there must not be frequent requests generated in the network. Due to the nature of todays networks, the number of requests is much more than SANE can handle.

Ethane [23] followed SANE in the same direction of thinking. The biggest difference from SANE is that it takes a less ambitious approach. The end hosts do not need to be changed because source route is no longer used, thus the proxy to translate IP addresses to source routes is no longer needed. It also enables incremental deployment since it is possible to couple Ethane flow-based switches with Ethernet switches. Ethane also uses a central controller to enforce security policies and compute routes for flows in the network. It also provides a policy composition language called Pol-Eth, inspired from predicate routing (PR) [24] for programming the security policies based on identity bindings. Even though Ethane is almost the same as previous proposals, its real contribution was the fact that the system was deployed in real life at Stanford's Computer Science Department [25]. They build different type of Ethane switches, like wireless switches, hardware-accelerated wired switches as well as pure software wired switches. With a sample of around 22,000 they concluded that one central controller is enough to handle

all the requests. The drawbacks of Ethane are: **(a)**, the central controller is a monolithic control plane; therefore it can only support existing functionalities and modify them. In case that the users want to add other features into the controller plane, or even replace existing features with other implementations, it will not be an easy job and they may face serious problems. **(b)**, the central controller of Ethane, as in the case of SANE, cannot scale up very well.

Tesseract [26], designs and implements all the planes proposed by 4D architecture. In contrast to SANE and Ethane, Tesseract works towards more classical and more general ways of routing, that is, non-flow-based routing. In Tesseract, the central controller pre-computes forwarding paths for all allowed traffic and configures routers/switches whose responsibilities are forwarding packets, thus it can work with both IP and Ethernet networks. Tesseract proved that it is practical to separate decision logics from classical packet-based routing network and to centralise such decision logics with reasonable scalability and convergence performance upon network failures. Furthermore, Tesseract contributed in the design and implementation of secure dissemination service for the dissemination plane. This is important because it is separated from the data plane, so circular dependency between correct data plane behaviour and working control channel does not exist as in the case that control channel relies on data plane. In relation to the decision plane, Tesseract includes different control components such as incremental shortest path routing, traffic engineering, spanning tree algorithm and filter placement algorithm. However, Tesseract's decision plane is a monolithic system, with all the components being more or less coupled with each other.

2.2.3 NOX and Maestro

All the previously shown works have a monolithic central control plane in which all the functionalities are more or less "hard-coded". It is difficult for the users to replace or rewrite specific control components to reach special control goals. Therefore a modularised and flexibly programmable centralised control plane framework will make it much easier for users to realise complicated and flexible management goals. NOX [27] which is a follow up work of SANE and Ethane, billed as a "network operating system", concentrates on providing such a modularised and

flexible framework for users to write control components. Due to the fact that the control plane is responsible for establishing every flow in the network, if it does not have enough capacity in handling all the requests, it will become the bottleneck of the network. The initial version of NOX lacked of such throughput scalability because it can only utilise one CPU core. Although cooperative-threading is used to reduce the overhead introduced by waiting for I/O operations, it is not really multi-threaded to leverage multi-core processing. Furthermore, NOX processes each request individually, thus there is huge amount of overhead introduced by such separate processing. However, a multi-threaded version of NOX, NOX-MT [28] was released showing that it can handle well these scalability problems.

Maestro [29,30] is another project billed as "network operating system" which was developed in parallel to NOX. In general, network operating systems have two basic purposes. First, to provide an application with a higher level of abstraction so they do not need to deal with low-level details and second, to control the interactions between applications. NOX focuses on the first purpose whereas Maestro focuses on the second. It is a flexible programming framework for composing centralised network control functions for different types of networks. It can be applied in a classical packet-based routing network, in a flow-based routing network like OpenFlow [9], or even in a network to coordinate centralised controls with distributed routing protocols. Maestro provides explicit direct control over interactions among control components, and over network state synchronisation. It tries to solve the scalability problem of the centralisation but focuses on a single machine solution by exploring parallelism provided by recent multi-core technology. Its goal was to build the best performance single machine OpenFlow controller. Finally, Maestro coordinates centralised and distributed network controls to solve the responsiveness and robustness problems of pure centralised solutions.

2.2.4 HyperFlow, Onix and DIFANE

Apart from maximising the performance of each physical controller machine, several works have aimed at enabling a cluster of controller machines to work as a single logical controller to further improve scalability. HyperFlow [31] extends NOX

into a distributed control plane. By synchronising network-wide state among distributed controller machines in the background through a distributed file system, HyperFlow ensures that the processing of a particular flow request is localisable to an individual controller machine, thus minimising the control plane response time to data plane requests, and at the same time improving the whole system's throughput. However due to the fact that the control plane is again distributed, and HyperFlow does not provide strong guarantee against network state inconsistency, it still has the problems that distributed controls have.

Onix [32] provides a general framework for building distributed coordinating network control plane, especially for the case of OpenFlow controllers. Onix provides a Network Information Base (NIB) roughly analogous to the RIB used by IP routers, which gives, users, access to several state synchronisation frameworks with different consistency and availability requirements.

Through DIFANE [33] a different approach is presented in order to improve flow-based networks' control plane performance. Instead of only verifying flows and computing paths for them upon request, DIFANE proactively computes wildcard matching rules for flows based on high level policies. Such rules are distributed among authority switches in the network, in order to improve both scalability and robustness, and at the same time reduce the length of the path that needs to be taken by the first packet of a flow. That way, switches are not only responsible for data plane functionality, but are responsible for control plane functionalities as well. On the other hand the central controller is only responsible for partitioning and distributing rule partition among these authority switches, and does not need to be involved in matching packets against these rules as in OpenFlow. DIFANE authors showed that it can achieve very good scalability in throughput of handling flow requests, compared to centralised OpenFlow controller NOX. However, NOX's security model is strong, such that all flows are explicitly controlled and managed by the central controller. In the case of DIFANE, since it implements rule pre-computation and distribution, it can increase the chance that attackers can direct their traffic through in the network. The way DIFANE is used, it cannot dynamically control the security policies flexibly as OpenFlow does, therefore comparing NOX with DIFANE is a bit unfair.

2.3 OpenFlow

Researchers face a huge problem in experimenting with new network protocols in a sufficiently realistic environment. The reason behind this is because networks have become a critical infrastructure for enterprises, homes and schools. Nobody is willing to allow researchers to experiment with production traffic or any other real life situation, due to the fact that any mistake may cause reductions in QoS, security and privacy issues or even in the worst case, a network failure. The only way researchers can test their ideas is by simulating or emulating environments in the lab. However, this comes at a cost, no simulated or emulated environment is close enough to real life situations. Therefore, even if a new idea succeeds in the lab, it may end up to be a disaster if it is implemented in the outside world. This forces the enterprises not to give a lot of chances to new products, therefore networks have become more static and no real innovation is going on.

In order to enable large scale research an experimenting, projects such as PlanetLab [34] (a geographically distributed computer network for research purposes) and Emulab [35] have been created. Both projects were successful, and a lot of government funding has been assigned to them for networking research. Driven by the success of the several research ideas tested in PlanetLab and Emulab, a research group at Stanford University created the Clean Slate Program. The mission of this program as stated was to eventually "reinvent the Internet" by overcoming architectural limitations, incorporate new technologies, enable new class of applications and services and allow the Internet to be a platform for innovations. Due to the fact that such ideas needed a large scale experimentation platform, they have decided to create OpenFlow. At first, the idea behind OpenFlow was to enable them to use the Stanford University campus network for experimentation without affecting production traffic.

OpenFlow was the first implementation that brought a balance between full programmability of networks and a real-world development. Even though at first it relied on existing hardware that was not designed with OpenFlow in mind, it was very fast deployed and gave the existing university network more functions and flexibility. Essentially, OpenFlow was fulfilling the criteria for a very good

experimental platform:

1. Support of high-performance networks
2. Low-cost implementation
3. Be able to isolate experimental traffic from production traffic
4. Capable of supporting a broad range of research
5. Easily implemented in other laboratories for confirmation
6. Easily implemented on production hardware
7. Be consistent with vendor's need for closed platforms

There are other solutions that can achieve some of the criteria. One solution is the use of a PC equipped with a number of network interfaces. From one point of view, this is a very good solution since there are many operating systems that allow the implementation of packet routing protocols, therefore researchers can build their own protocols and experiment. Unfortunately, there are two limitations with this approach. The first limitation is the shortage of ports due to the fact that a PC cannot support the number of ports a typical switch does, and second limitation is the number of packets a PC can handle. Typical switches can handle more than 100Gbits/s whereas for a typical PC is hard to even exceed 1Gbit/s. Another solution is the use of Networks Field-Programmable Gate Array (NetFPGA) [36–39], which is a reasonably low-cost programmable Peripheral Component Interconnect (PCI) card for processing packets. NetFPGA is capable of handling more data than a PC, it is fully customisable, but it is limited to just four ports, which are insufficient for real life experimentation. In some cases researchers used more than one card in order to achieve more ports, but this increases the complexity as well as the cost.

The most promising solution and the one that meets almost all the criteria stated earlier is OpenFlow. OpenFlow exploits the fact that most Ethernet switches and routers contain flow-tables that run at line-rate to implement firewalls, Network Address Translation (NAT), QoS and to collect statistics, in order to provide an open protocol allowing users (users can be humans or machines)

to program flow-tables. Having this ability, researchers are able to control their own traffic by choosing the routes their packets follow and the processing they receive. This enables them to try new routing protocols, security models, addressing schemes and even alternatives to IP, on the same network as the production traffic without affecting the way production traffic is processed and routed.

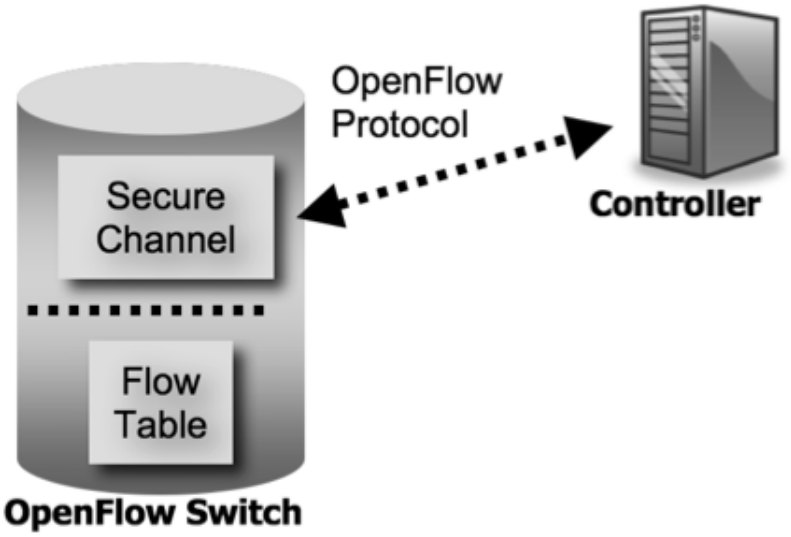


Figure 2.1: Main components of an OpenFlow switch (Source [3])

The minimum requirements for an OpenFlow switch (as shown in Figure 2.1) are the following:

1. A Flow Table, with an action associated with each flow entry, in order for the switch to process the flow (discussed in section 2.3.2)

2. A Secure Channel that connects the switch to a remote control process, allowing commands and packets to be sent between a controller and the switch (discussed in section 2.3.3)

3. Support of the OpenFlow Protocol in order to connect to a controller

OpenFlow can be implemented in traditional hardware as well as hardware supporting only OpenFlow. This creates two types of OpenFlow switches, *OpenFlow-Only* and *OpenFlow-Hybrid* (also known as *OpenFlow-Enabled*). The difference between them is that OpenFlow-Only switches support only OpenFlow operations. In such a switch all packets are processed by the OpenFlow pipeline and cannot

be processed otherwise. The OpenFlow-Hybrid switches support both OpenFlow operations as well as normal Ethernet switching operations. Such switches have to provide a classification mechanism outside of OpenFlow that routes traffic to either OpenFlow pipeline or normal pipeline. Such a mechanism could be the usage of the port number or the VLAN tag in order to define which flows go to OpenFlow pipeline and which go to normal pipeline. According to the specification of the OpenFlow protocol, researchers can define Flow Table entries externally without the need to program the switch. This makes OpenFlow the first standard communication interface defined between the control and data layers of an SDN architecture. OpenFlow became a key enabler of SDN and currently is the only standardised SDN protocol that allows direct manipulation of the data plane of network devices. Although it was initially applied to Ethernet-based networks, OpenFlow extended its usage to a much broader area as discussed in Section 2.7.

OpenFlow standardisation is an ongoing process managed by the Open Networking Foundation (ONF). The first OpenFlow Switch Specification (version 0.2.0) was released in 2008 and until 2017 several newer versions have been released. Some of the versions have an experimental approach, and some of them are under the stable category. All of the versions are listed in Table 2.1. Some of the versions included in the table, were released just to correct code mistakes of previous versions. These rapid developments in the OpenFlow protocol brought many advantages as well as some disadvantages. Some of the advantages include the fact that ONF seems to respond to researchers feedback by quickly fixing problems as well as supporting new features, thus OpenFlow becomes even more powerful. However, vendors cannot cope with that speed and it takes them a lot of time to implement the latest OpenFlow protocol on their machines. Even OpenFlow Controllers, which most of them have a great number of active developers working on them, are not implemented at those speeds. This results in both compatibility as well as research issues.

2.3.1 OpenFlow Specification

The first complete OpenFlow Switch Specification (version 1.0.0) was released in December, 2009 [3]. It requires an OpenFlow switch, consisting of a flow table,

OpenFlow Version	Release Date
0.2.0	Mar. 28, 2008
0.8.0	May 5, 2008
0.8.2	Oct. 17, 2008
0.8.9	Dec. 2, 2008
0.9.0	Jul. 20, 2009
1.0.0	Dec. 31, 2009
1.1.0	Feb. 28, 2011
1.2.0	Dec. 5, 2011
1.3.0	Apr. 13, 2012
1.3.1	Sep. 6, 2012
1.3.2	Apr. 25, 2013
1.3.3	Sep. 27, 2013
1.3.4	Mar. 27, 2014
1.3.5	Mar. 26, 2015
1.4.0	Oct. 14, 2013
1.4.1	Mar. 26, 2015
1.5.0	Dec. 19, 2014
1.5.1	Mar. 26, 2015
1.6	Sep. 2016

Table 2.1: OpenFlow Standards Release Dates

which performs packet lookup and forwarding, and a secure channel that connects the switch to an external controller. Using OpenFlow protocol the controller is responsible to manage the flow table of the switch. The flow table consists of a number of flow entries, activity counters and a set of zero or more actions. All packets that are processed by the switch are compared to the flow table entries. In the case of a matching entry, the switch proceeds with the actions specified. If there is no match, then the switch will forward the packet to the controller over the secure channel as shown in Figure 2.2. At that point the controller becomes responsible for the future of the packet. The controller is also responsible to add

or remove flow entries.

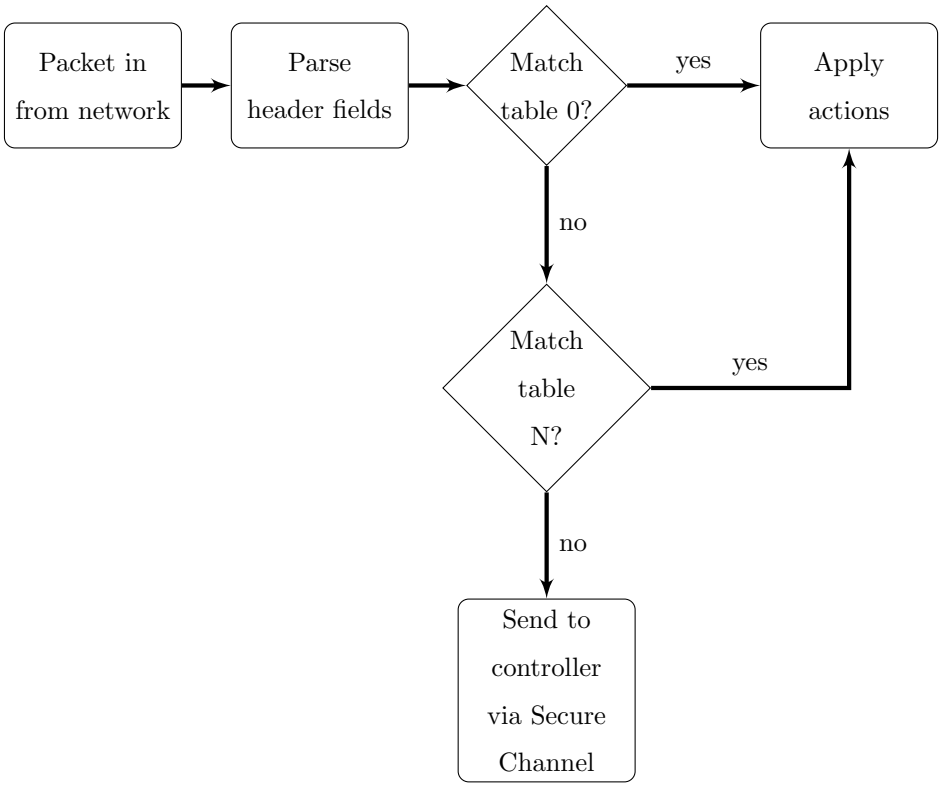


Figure 2.2: Packet flow in an OpenFlow switch

2.3.2 Flow Table

The flow table consists of three sections, the *Header Fields*, *Counters* and *Actions*. *Header Fields* section, began as a 12-tuple in OpenFlow Version 1.0.0 and became a 39-tuple in OpenFlow Version 1.3.2 [40], and contains the fields listed in Table 2.2. These fields are used in order for the switch to match all incoming packets. The *Counters* section (listed in Table 2.3) keeps count of several numbers that can be used to calculate several statistics that are useful in flow control and management. These counters are updated by the switch upon packet matching. Finally, *Actions* section contains the actions that the switch will perform on the matching packets. Each flow can have zero or more actions, where the case of zero actions results in a packet drop. Also the switch is able to reject a flow entry if it cannot process the action list in the order specified. Some of the actions are required and some are optional. The required actions appear in OpenFlow-Only switches whereas optional actions may appear in OpenFlow-Hybrid switches. All the actions are

listed in Table 2.4

2.3.3 Secure Channel

As mentioned in OpenFlow minimum requirements in Section 2.3, the secure channel is the interface that connects the OpenFlow switch to a controller. Using this interface, the controller configures and manages the switch, sends packets out to the switch as well as it receives events from the switch. The secure channel supports three message types, *controller-to-switch*, *asynchronous* and *symmetric*, each of them having multiple sub-types as shown in Table 2.5.

Controller-to-switch messages are created and sent by the controller in order to directly manage or inspect the state of the switch. On the other hand, *asynchronous* messages are created and sent by the switch with the purpose of updating the controller on network events and changes to the switch state. Finally *symmetric* messages are created and sent by either the switch or the controller and are typical messages like requests, replies or connection startup.

2.3.4 OpenFlow Versions Comparison

This section presents a comparison between different versions of OpenFlow, including the changes OpenFlow underwent through version development. Only versions 1.0.0 [3], 1.2.0 [41], 1.3.0 [42] and 1.4.0 [43] are compared due to the fact that they form the most complete OpenFlow versions. Even though OpenFlow first release was version 0.2.0 as shown in Table 2.1, all the versions prior to 1.0.0 were either not complete or simply a set of ideas for creating a more solid version.

OpenFlow version 1.0.0, released in December 2009, came to introduce a variety of features and to form the basis of the OpenFlow protocol. It clarified at a high level of detail how OpenFlow protocol works and left many open doors for radical improvements in future work. This can be considered as a satisfactory approach since it presented the developers and system engineers with enough functionality to work, and also allowed research to remain active, thus helping to expand OpenFlow capabilities. In fact, the majority of OpenFlow version 1.0.0 functions are inherited by later version releases as shown in Tables 2.2, 2.3, 2.4 and 2.5. One of its major drawbacks was the fact that it supported only IPv4. Researchers can argue that

	OpenFlow Version					
Field	**1.0.0**	**1.1.0**	**1.2.0**	**1.3.0**	**1.4.0**	**1.5.0**
Ingress Port	✓	✓	✓	✓	✓	✓
Physical Ingress Port			✓	✓	✓	✓
Table Metadata		✓	✓	✓	✓	✓
Ethernet Source Address	✓	✓	✓	✓	✓	✓
Ethernet Destination Address	✓	✓	✓	✓	✓	✓
Ethernet Type	✓	✓	✓	✓	✓	✓
VLAN ID	✓	✓	✓	✓	✓	✓
VLAN Priority	✓	✓	✓	✓		✓
VLAN PCP					✓	
IP DSCP			✓	✓	✓	✓
IP ECN			✓	✓	✓	✓
MPLS Label		✓	✓	✓	✓	✓
MPLS Traffic Class		✓	✓	✓	✓	✓
MPLS BoS bit				✓	✓	✓
Provider Backbone Bridges I-SID				✓	✓	✓
Logical Port Metadata				✓	✓	✓
IPv6 Extension Header				✓	✓	✓
IPv4 Source Address	✓	✓	✓	✓	✓	✓
IPv6 Source Address			✓	✓	✓	✓
IPv4 Destination Address	✓	✓	✓	✓	✓	✓
IPv6 Destination Address			✓	✓	✓	✓
IPv6 Flow Label			✓	✓	✓	✓
ICMPv6 Type			✓	✓	✓	✓
ICMPv6 Code			✓	✓	✓	✓
ND Target Address			✓	✓	✓	✓
ND Source link-layer			✓	✓	✓	✓
ND Target link-layer			✓	✓	✓	✓
IP Protocol	✓	✓	✓	✓	✓	✓
IPv4 ToS bits	✓	✓				
TCP Source Port			✓	✓	✓	✓
TCP Destination Port			✓	✓	✓	✓
UDP Source Port			✓	✓	✓	✓
UDP Destination Port			✓	✓	✓	✓
SCTP Source Port			✓	✓	✓	✓
SCTP Destination Port			✓	✓	✓	✓
ICMP Type			✓	✓	✓	✓
ICMP Code			✓	✓	✓	✓
ARP Opcode			✓	✓	✓	✓
ARP Source IPv4 Address			✓	✓	✓	✓
ARP Target IPv4 Address			✓	✓	✓	✓
ARP Source Hardware Address			✓	✓	✓	✓
ARP Target Hardware Address			✓	✓	✓	✓
Transport Source Port / ICMP Type	✓	✓				
Transport Destination Port / ICMP Code	✓	✓				
Provider Back Bone UCA						✓
TCP Flags						✓
Action Metadata Output Port						✓
Packet Type Value						✓

Table 2.2: List of Header Fields

	OpenFlow Version					
Counter	**1.0.0**	**1.1.0**	**1.2.0**	**1.3.0**	**1.4.0**	**1.5.0**
Per Table						
Reference Count (active entries)	✓	✓	✓	✓	✓	✓
Packet Lookups	✓	✓	✓	✓	✓	✓
Packet Matches	✓	✓	✓	✓	✓	✓
Per Flow						
Received Packets	✓	✓	✓	✓	✓	✓
Received Bytes	✓	✓	✓	✓	✓	✓
Duration (seconds)	✓	✓	✓	✓	✓	✓
Duration (nanoseconds)	✓	✓	✓	✓	✓	✓
Per Port						
Received Packets	✓	✓	✓	✓	✓	✓
Transmitted Packets	✓	✓	✓	✓	✓	✓
Received Bytes	✓	✓	✓	✓	✓	✓
Transmitted Bytes	✓	✓	✓	✓	✓	✓
Received Drops	✓	✓	✓	✓	✓	✓
Transmit Drops	✓	✓	✓	✓	✓	✓
Received Errors	✓	✓	✓	✓	✓	✓
Transmit Errors	✓	✓	✓	✓	✓	✓
Receive Frame Alignment Errors	✓	✓	✓	✓	✓	✓
Receive Overrun Errors	✓	✓	✓	✓	✓	✓
Receive CRC Errors	✓	✓	✓	✓	✓	✓
Collisions	✓	✓	✓	✓	✓	✓
Duration (seconds)				✓	✓	✓
Duration (nanoseconds)				✓	✓	✓
Per Queue						
Transmit Packets	✓	✓	✓	✓	✓	✓
Transmit Bytes	✓	✓	✓	✓	✓	✓
Transmit Overrun Errors	✓	✓	✓	✓	✓	✓
Duration (seconds)				✓	✓	✓
Duration (nanoseconds)				✓	✓	✓
Per Group						
Reference Count (flow entries)		✓	✓	✓	✓	✓
Packet Count		✓	✓	✓	✓	✓
Byte Count		✓	✓	✓	✓	✓
Duration (seconds)				✓	✓	✓
Duration (nanoseconds)				✓	✓	✓
Per Group Bucket						
Packet Count		✓	✓	✓	✓	✓
Byte Count		✓	✓	✓	✓	✓
Per Meter						
Flow Count				✓	✓	✓
Input Packet Count				✓	✓	✓
Input Byte Count				✓	✓	✓
Duration (seconds)				✓	✓	✓
Duration (nanoseconds)				✓	✓	✓
Per Meter Band						
In Band Packet Count				✓	✓	✓
In Band Byte Count				✓	✓	✓

Table 2.3: List of Counters

		OpenFlow Version					
Action	**Process**	**1.0.0**	**1.1.0**	**1.2.0**	**1.3.0**	**1.4.0**	**1.5.0**
Forward	ALL	✓	✓	✓	✓	✓	
	CONTROLLER	✓	✓	✓	✓	✓	✓
	LOCAL	✓	✓	✓	✓	✓	✓
	TABLE	✓	✓	✓	✓	✓	✓
	IN_PORT	✓	✓	✓	✓	✓	✓
	NORMAL	✓	✓	✓	✓	✓	✓
	FLOOD	✓	✓	✓	✓	✓	✓
	ANY			✓	✓	✓	✓
	Enqueue	✓					
	Drop	✓	✓	✓	✓	✓	✓
	Set-Queue		✓	✓	✓	✓	✓
	Group		✓	✓	✓	✓	✓
	Output		✓	✓	✓	✓	✓
	QoS		✓	✓		✓	✓
Push/Pop	Push VLAN header		✓	✓	✓	✓	✓
	Pop VLAN header		✓	✓	✓	✓	✓
	Push MPLS header		✓	✓	✓	✓	✓
	Pop MPLS header		✓	✓	✓	✓	✓
	Push PBB header				✓	✓	✓
	Pop PBB header				✓	✓	✓
Modify-Field	VLAN ID	✓	✓	✓	✓	✓	✓
	VLAN priority	✓	✓	✓	✓	✓	✓
	Strip VLAN header	✓				✓	✓
	Ethernet source MAC address	✓	✓		✓	✓	✓
	Ethernet destination MAC address	✓	✓		✓	✓	✓
	IPv4 source address	✓	✓				
	IPv4 destination address	✓	✓				
	IP source address					✓	✓
	IP destination address					✓	✓
	TCP/UDP source port					✓	✓
	TCP/UDP destination port					✓	✓
	IPv4 ToS bits	✓	✓				
	Transport source port	✓	✓				
	Transport destination port	✓	✓				
	MPLS label		✓	✓	✓		
	MPLS traffic class		✓	✓	✓		
	MPLS TTL		✓	✓	✓	✓	✓
	IPv4 ECN bits		✓				
	IPv4 TTL		✓	✓			
	TTL outwards		✓	✓		✓	✓
	TTL inwards		✓	✓		✓	✓
	IP TTL			✓	✓	✓	✓
	PBB I-SID				✓	✓	✓
	PBB I-PCP				✓	✓	✓
	PBB C-DA				✓	✓	✓
	PBB C-SA				✓	✓	✓

Table 2.4: List of Actions

		OpenFlow Version					
Type	**Message**	**1.0.0**	**1.1.0**	**1.2.0**	**1.3.0**	**1.4.0**	**1.5.0**
Controller-to-Switch	Features	✓	✓	✓	✓	✓	✓
	Configuration	✓	✓	✓	✓	✓	✓
	Modify-State	✓	✓	✓	✓	✓	✓
	Read-State	✓	✓	✓	✓	✓	✓
	Send-Packet	✓					
	Barrier	✓	✓	✓	✓	✓	✓
	Role-Request				✓	✓	✓
	Asynchronous-Configuration				✓	✓	✓
	Packet-out		✓	✓	✓	✓	✓
Asynchronous	Packet-in	✓	✓	✓	✓	✓	✓
	Flow-Removed	✓	✓	✓	✓	✓	✓
	Port-status	✓	✓	✓	✓	✓	✓
	Error	✓	✓	✓	✓	✓	✓
Symmetric	Hello	✓	✓	✓	✓	✓	✓
	Echo	✓	✓	✓	✓	✓	✓
	Vendor	✓					
	Experimenter		✓	✓	✓	✓	✓

Table 2.5: List of Messages

it was not a drawback since at that time, OpenFlow was not advertised as being one of the technological advantages of the future. Back in 2009, OpenFlow was considered a new protocol for helping researchers to experiment on large scale (university campus) networks. On the other hand, it was well known that IPv4 had a limited time left, therefore OpenFlow would have a limited lifetime if it supported only IPv4.

The limited number of header fields available in version 1.0.0 can be considered as another limitation. With such limited variety of header fields, OpenFlow could not be used in enterprise networks and perform to an accepted level. Similarly, counters and actions were also limited, which it was a major drawback for OpenFlow as an enterprise solution candidate. Unlike those three groups of fields, the variety of the messages between the switch and the controller were very satisfactory and enough to allow a broad range of communication capabilities.

The fact that version 1.0.0 supported multiple queues per output port, that can provide minimum bandwidth guarantees can be considered as an advantage. This function was called “slicing” due to the fact that it was able to provide a slice of the available bandwidth to each queue. The final two limitations of version

1.0.0 are the absence of multiple tables and groups.

OpenFlow version 1.2.0 inherited all the capabilities of version 1.0.0 and extended them in order to overcome some of its major drawbacks. One of the major improvements was the OpenFlow Extensible Match (OXM) which allowed the user to include new fields that can be used for packet matching. As a result, users were no longer dependent on the predefined OpenFlow matching fields. The second major improvement was the implementation of IPv6 support and the third, which was actually released in version 1.1.0 and improved in version 1.2.0, was the support of multiple tables and groups.

Other minor improvements of version 1.2.0 include:

1. Introduction of "metadata" field in the "packet-in" message, which helps the controller to figure out what happened to the packet inside the switch before reaching the controller. Through this feature, the controller is able to know which flow entries were matched or not matched against the packet's header.

2. Addition of a function that enables users to create custom error messages.

3. Implementation of controller role change mechanism. Such a mechanism allows the switch to be connected to several controllers in parallel, thus in case of a controller failure the switch can change to another controller. Controllers on their own can set their role (i.e. equal, master or slave) in order to help the switch choose the best controller in case of failure.

OpenFlow version 1.3.0 was released in June 2012 with a large number of additions and corrections to previous versions bugs. Major improvements include:

1. Ability to match IPv6 extension headers such as hop-by-hop, router, fragmentation, authentication, encrypted security payload and destination options.

2. Support for per-flow meters that can be attached to flow entries and can measure and control the rate of packets.

3. Per connection event filtering function, which allows controller to filter undesirable events coming from the switch.

4. Auxiliary connections for the controller-to-switch link. Controller-to-switch link supported only TCP connection in previous versions. In version 1.3.0 this connection can be set as TCP,UDP or DTLS.

5. Flexible non-matching flows (table-miss) handling. In previous versions, users were forced to use one of the three predefined behaviours for flows that did not match the flow table entries. In version 1.3.0 those behaviours were replaced by a separate table-miss flow entry. Through this approach, users can specify their own behaviours.

Version 1.3.0 included some minor corrections such as matching of Multiprotocol Label Switching Bottom of Stack bit (MPLS BoS), Provider Backbone Bridging (PBB) tagging, more flexible tag ordering, efficient classification of packet-in messages using a new field called "Cookies", on demand flow counters and the addition of "Duration" field for most of the statistics.

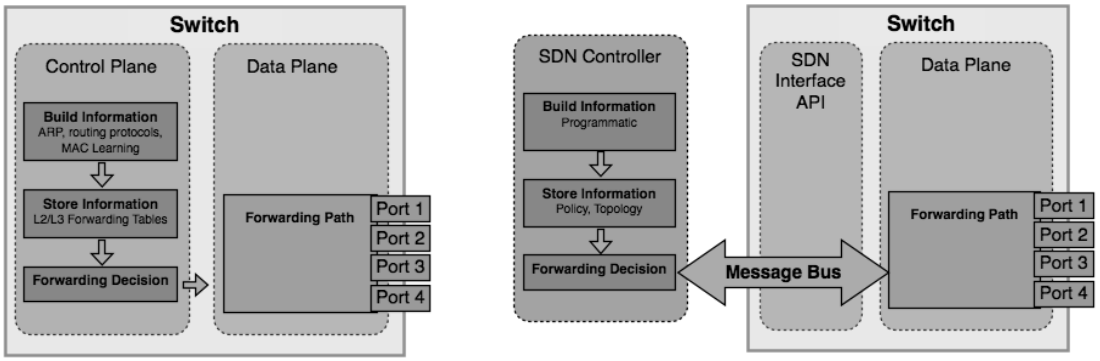
ONF Extensibility Working Group which is responsible for developing extensions for OpenFlow, released a report on April 2013 [44] stating that it is not planning to further extend OpenFlow version 1.0.X. Furthermore, it stated that OpenFlow version 1.3.X will become a long term support version and version 1.4.0 will have incremental improvements on previous versions.

OpenFlow version 1.4.0 was finally released in October 2013, with 4 months delay since it was expected in June. The main advancement was the improvement of extensibility first introduced in OpenFlow version 1.2. Except from extensibility, some new more descriptive "reason" values were added such as table miss (i.e. no matching flow in the flow table) and invalid TTL for packets with an invalid Time to Live (TTL). In addition, version 1.4.0 added support for Optical ports, flow removal due to meter deletion as well as some enhancements in flow monitoring.

2.4 Software Defined Networking

SDN is the latest centralised network control architecture that is promising to redefine networking as we know it, altering the current architecture of networks by splitting apart data plane and control plane as shown in Figure 2.3.

As shown in Section 2.2 the idea of Programmable Networks existed for years before the arrival of OpenFlow. Therefore, the methodology and the principles used to form the basis of SDN have matured after years of research. The only problem that was faced by SDN was that none of the actual implementations of SDN showed any real potential for real-world applications. That is why OpenFlow is the most important SDN implementation. It was the first implementation that achieved the SDN goals in such a simplistic but elegant way, allowing the SDN architecture to be deployed in real-world enterprise networks. With the arrival of OpenFlow, the SDN community began working again into finalising the SDN architecture in order to achieve the appropriate goals.



(a) Traditional Switch **(b)** SDN Switch

Figure 2.3: Changes to hardware due to SDN

2.4.1 SDN Architecture

Data plane, also known as the forwarding plane is responsible to deal with packets arriving on an inbound interface. Most of the times, the data plane will look in a table or a number of tables that have specific rules on how to deal with the packet. Routing Information Base (RIB) [45] is one of the tables that the data plane will look at before taking an action on the packet. Such a table contains a list of routes to particular networks destinations as well as metrics associated with those routes [45]. In other words it is kind of a network map. The reason they are helpful is because if the router cannot send a packet directly to the destination, it can use an indirect way of sending the packet. This can be done by sending the packet to a node that can reach the destination.

Another table that can be found in the data plane is called Forwarding Information Base (FIB) [46]. It is roughly the same as RIB, but has a better next hop management leading to less Central Processing Unit (CPU) usage thus avoiding router meltdown. Apart from the fact that it can send the packet to the destination, it can also discard the packet. According to Internet Protocol (IP) specification [47], upon that decision it will reply back to the sender of the packet with an Internet Control Message Protocol (ICMP) stating that the destination is unreachable. In some cases this is not true due to the fact that by replying back, a potential attacker becomes aware that the system is protected. Therefore, the router drops the packet silently.

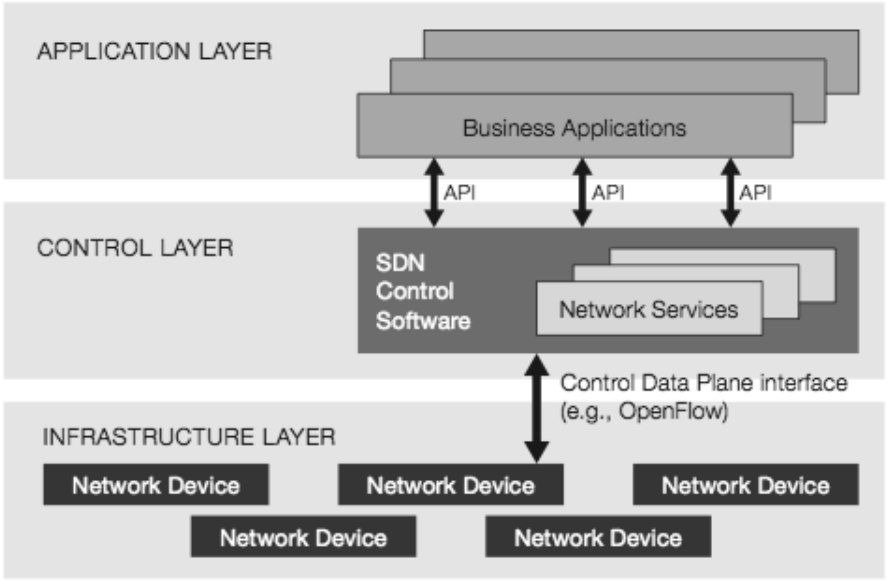


Figure 2.4: Software-Defined Network Architecture (*Source: [4]*)

Control plane on the other hand, is responsible for finding the information that builds up the routing tables as well as the network map [15]. The most important role that the control plane plays, apart from the aforementioned, is that it can decide the best route so it can install it as the most preferred route in the routing tables. It gets its information by monitoring the hardware status, from dynamic routing protocols or from a manually preconfigured route that a network administrator has installed. Sometimes in comparing the information, it may find that there are more than one equally good routes thus, it may end up sharing the traffic across them in order to minimize the load.

This decoupling of the two planes achieved in the SDN architecture, brings many advantages in the area of networks. From enterprises point of view, one of the greatest advantages is that they are not dependent on existing protocols as well as vendors anymore. Almost all control and data plane functions are developed and maintained by vendors, and enterprises cannot alter or improve them. Traditionally, if an enterprise wants to develop a new function, it has to wait for the vendors to accept it and develop it into a new released firmware or even a new device. This usually takes a lot of time, ranging from months to even years. In some cases the vendor may find it a waste of time and not even implement it, thus resulting in an increase in cost because enterprises will need to purchase new equipment capable of serving their needs. By splitting apart data and control plane, and by moving the control plane to a separate controller, SDN gives the ability to network administrators and operators to programmatically create and apply their own configurations and methodologies to all of their equipments without having to configure each device separately. Furthermore, it allows them to modify their networks in a matter of hours or days rather than depend on a vendor that may take weeks or even months to create a new solution for their machines. Last but not least, network equipment does not need to have its own private controller, thus one controller may control any number of equipment. Furthermore, several controllers may be controlled by a more powerful controller creating a tree structure, which gives the ability to make global changes in a matter of seconds, avoiding any security bridges or failures in the Quality of Service (QoS).

2.4.2 SDN Implementations

Except from OpenFlow (Section 2.3) which is the most widely accepted and deployed open SDN implementation, there are several other implementations that come with some important capabilities. Protocol-Oblivious Forwarding (POF) [48] is one of the main OpenFlow competitors as it does not only implements SDN but it also enhances it. The biggest advantage of POF is that it proposes a generic flow instruction set (FIS) which essentially creates a protocol-oblivious forwarding plane. This is due to the fact that the forwarding devices act as white boxes, therefore the packet parsing is done by the controller. This solves several Open-

Flow issues, including version compatibility. In OpenFlow, in order for a packet to be examined, several header fields are used. With each new OpenFlow version, new header fields are added, thus creating backwards compatibility issues. As a result, the OpenFlow switches have to support the OpenFlow version used by the controller. On the other hand, POF switches are protocol-agnostic since all the processing and parsing is performed by the controller.

The Open vSwitch Database Management Protocol (OVSDB) [49] can be considered as an OpenFlow extension since it uses a lot of OpenFlow's functionality, but its purposes is to give more management flexibility to SDN. It allows users to create vSwitch instances, control individual elements, configure tunnels, set QoS policies as well as collect statistics and manage queues. Similarly to OVSDB, both OpenState [50] and Revised OpenFlow Library (ROFL) [51] are extensions of the OpenFlow protocol. OpenState introduces programming abstractions into the forwarding plane in order to extend OpenFlow's match and action abstractions. This allows the programmers to develop several tasks and procedures inside the networking equipment instead of the controller. ROFL extends OpenFlow by introducing an abstraction layer that alleviates OpenFlow versions. This results in backwards compatibility and also it gives the developers a much clearer API to work with.

OpFlex [52] is one of the newest SDN proposals which has the goal of distributing parts of its functionality back to the networking devices in order to improve both their performance and the network scalability. Other than that, it shares the same logical centralisation as OpenFlow, with mostly the same functionality.

2.5 SDN Controllers

In the SDN architecture, the controller is the device responsible for maintaining and distributing all of the network rules as well as instructions. It determines how the switch should handle the packets by adding, modifying as well as deleting entries from the switch's flow table. Apart from sending rules via the secure channel, controllers can also request network and traffic related statistics from the switch.

The controller usually runs on a network-attached server, and can manage all the network's devices, a group of them or a single device. Managing all of the network's devices results to a centralised configuration. This gives the advantage to the controller to know the network state in every single device of the network, allowing it to take precautions in case of failures. On the other hand this may cause problems due to the fact that the network will end up with a single point of failure. Furthermore, the processing power needed for such a controller will be enormous; therefore, such an approach may add extra delay to the network.

On the other hand a controller can manage a group or a single switch, eliminating the single point of failure as well as the enormous processing power needed. Such an approach will not give the centralisation of the network given by the previous approach. A more powerful approach would be a combination of the two in a tree structure. Such an approach would give reasonable centralisation as well as minimising the possibility of one point of failure as well as the processing power required.

Most importantly, the controller should act as a high-level programming API, allowing the network administrators to develop their own policies and management schemes. This can solve problems faced by device-specific low level management instruction commands found in most networking operating system.

2.5.1 Controller Behaviours

Controllers can manage the network in different ways. Some of these ways are listed below.

- *Flow Routing:* The controller adds a flow entry for every flow. This forces the network device to search for an *exact* match before redirecting the flow. This is reasonable to use in experimental networks or in campus networks used by researchers.

- *Aggregated:* The controller adds a flow entry for a group of flows. For example, instead of an exactly packet match, the network device can look only at one packet's feature. That kind of flow entry is called a *wildcard*. Such an approach is preferred for use in network backbones since all the

packets arriving there would have been proven not malicious by previous controllers/devices handling the network.

- *Reactive:* The controller adds a flow entry once it is triggered by a flow. In this way, the delay experienced by flows will be increased (depending on the number of flows arriving) since the network device would have to wait for the controller to decide and install a flow entry before forwarding the flow.

- *Proactive:* As soon as the controller is connected to the network device, it pre-installs several entries in the flow table. Using that approach will minimize the overall delay experienced and the traffic will continue undisrupted.

2.5.2 Controller Examples

Apart from NOX and Maestro (discussed in Section 2.2.3), who began their journey as "Network Operating Systems" and they are now known as two of the best OpenFlow controllers, there are several other controllers which through extensive research as well as production usage, have been proven to be very competitive.

2.5.2.1 Trema

Trema [53], originally designed and developed by NEC research lab, is an open source OpenFlow controller platform mainly for research. Has a multi-process modular architecture that provides enough stability to be extended to a distributed controller. It has integrated testing and debugging environment and is able to provide support for manage, monitor and diagnosis on the entire system. After the year 2011, apart from code written in C, Trema fully supports code written in Ruby as well. The main Trema project goal was to allow researchers to develop their own controllers on top of it.

2.5.2.2 Beacon

Beacon [54], is a Java cross-platform open source OpenFlow controller that supports both event-based and threaded operations. Furthermore, is a very stable multithreaded controller that comes with an extensible UI framework. Due to its multithreaded nature, researchers as well as network administrators can start,

stop, refresh, install or even delete code bundles during runtime, without interrupting any other non-dependent running bundles.

2.5.2.3 SNAC

Simple Network Access Control (SNAC) [55], designed by Stanford Clean Slate Program, Nicira and GENI/NSF, is an open source OpenFlow controller that comes with a web-based policy manager Graphical User Interface (GUI). SNAC is not a standalone controller, it is a module of NOX and therefore requires NOX controller to work. Some of its features include increased visibility, captive portal and flexible policy manager.

2.5.2.4 OpenDaylight

OpenDaylight [56] is a Linux Foundation collaborative project that has been highly supported by Cisco, Big Switch, and several other networking companies. Like Floodlight, OpenDaylight is written in Java and is a popular, well-supported SDN controller. It also includes exposure with a REST API and a web based GUI. The second release of OpenDaylight (Helium) includes support for SDN, Network Virtualization (NV) and Network Functions Virtualization (NFV) and is intended to be scaled to very large sizes. Like Floodlight it also has a number of pluggable modules (interfaces, protocols, and applications) that can be used to alter it to the needs of an organisation. OpenDaylight is a little different from other controllers because it allows for other non-OpenFlow southbound protocols.

More OpenFlow controllers are listed in Table 2.6

2.6 Benchmarking Simulation and Emulation Environments

In the area of benchmarking simulation and emulation environments for SDN, there is not much done other than the work presented by the developers of each environment. Therefore, in this section most of the benchmarks presented come directly from those environments proposal papers, and cannot be used in a useful

Controller	Prog. Language
POX	Python
NOX-Classic	C++, Python
NOX [27]	C++
Trema [53]	C, Ruby
Beacon [54]	Java
Floodlight [57]	Java
Maestro [29]	Java
Ryu [58]	Python
NodeFLow [59]	JavaScript
Helios [60]	C
BigSwitch [61]	Java
SNAC [55]	C++, Python
IRIS [62]	Java
OpenDaylight [56]	Java
DISCO [63]	Java
HP VAN SDN [64]	Java
HyperFlow [31]	C++
Kandoo [65]	C, C++, Python
Meridian [66]	Java
MuL [67]	C
OpenContrail [68]	Python, C++, Java
ProgrammableFlow [69]	C
SMaRtLight [70]	Java
yanc [71]	C++, Python
Fleet [72]	Python
NVP Controller [73]	C++
PANE [74]	Java
Rosemary [75]	Python, Java
MobileFlow [76]	Custom Commands
UnifiedController [77]	Custom Commands

Table 2.6: List of OpenFlow Controllers

comparison due to the fact that they do not share the same parameters, topologies and methodologies.

In [78], several basic benchmarks are presented for EstiNet OpenFlow Simulator and Emulator. It presents the main memory consumption using incremental

number of OpenFlow switches, the Average Ping Delay (APD) together with its Standard Deviation as well as the No Response Failure Rate (NRFR). Furthermore, it suggests that EstiNet solves several problems that Mininet has, which result from the fact that Mininet is highly depended on the Operating System Scheduler. This is partially true due to the fact that EstiNet solves the problem if and only if the simulation mode is used. In the emulation mode it still has the same problems as Mininet has.

In [79], an OpenFlow Extension for OMNeT++ [80] using the INET Framework [81] is presented. Unfortunately it is not as comprehensive as [78] (EstiNet Benchmarks), even though it gives the mean Round-Trip Time (RTT) of different spanning trees, which is one of the problems that solves. The reason that it cannot be considered as comprehensive is because other than one area, it cannot be used as a comparison to other existing platforms.

In [82], benchmarks for Mininet such as end-to-end bandwidth, setup time, stop time and memory usage are presented. The benchmarks presented cannot be considered as comprehensive due to the fact that each result comes from a totally different topology. For example, it gives results from Linear topology with 100 switches and it compares them to Tree, Fat Tree and Mesh topologies. It would have been more comprehensive to compare each topology against the same topology and just change some parameters such as the number of switches or nodes than compare each topology with a totally different topology. Furthermore, all of the results come from a virtual machine running on an Apple MacBook Pro, and there is no indication in the paper about any effects the laptop's OS has on the results. The most comprehensive Mininet benchmark comes from a 2012 technical report [83], which takes four typical topologies and tests them with different number of switches, giving out metrics like throughput and fairness.

2.7 OpenFlow Related Projects

OpenFlow has already found many uses in academia research projects as well as production networks. All these OpenFlow uses are helping OpenFlow as a protocol to become more mature due to the extensive testing as well as the active

team that maintains and extends the protocol. Below we present several OpenFlow related projects which either contribute to OpenFlow enhancement or use OpenFlow features to enhance existing networking.

2.7.1 Data Centre Related

PortLand [84], is a set of Ethernet compatible routing, forwarding and address resolution protocols, which by using OpenFlow, creates a scalable fault-tolerant data centre network fabric. It consists of a logically centralised fabric manager that maintains soft state about network configuration information such as topology. It is responsible for assisting with ARP resolution, fault tolerance and multicast. The local switches that are connected to the rest of the devices, communicate with the fabric manager through OpenFlow protocol. Using OpenFlow, fabric manager resolves ARP requests and manages forwarding tables for multicast sessions. It also monitors connectivity with each switch and reacts to the live information by updating its fault matrix. Switches also send "keepalives" to their immediate neighbours every 10ms in order to detect any link failure and update the fabric manager.

Ripcord [85], is a modular platform for rapidly prototyping scale-out data centre networks. It enables researchers to build and evaluate any network features and topologies, using only commercially available hardware and open-source software. Its whole architecture is based on OpenFlow programmable switches and by using NOX it passes the messages between modules and also modifies and views switch state. All of the details such as flow entries and statistics are transferred using the OpenFlow protocol.

NOX to Data centre [86], is a research which demonstrated the effectiveness that NOX, in combination with OpenFlow, can provide to the data centre. It states several advantages that NOX alone can provide and also demonstrates the extra advantages a data centre can get once it uses both NOX and OpenFlow. Even though NOX can use any similar to OpenFlow protocols for manipulating switch forwarding entries, OpenFlow provides the capability to install a second flow entry of lower priority, allowing NOX to have previously calculated and installed backup paths in case of failure in order to minimize latency. Furthermore,

both of them combined provide the basis for an integrated monitoring architecture through the per flow and per port statistics maintained.

CloudNaaS [87], acts as a service platform that uses OpenFlow in order to provide extended networking functionality to production networks running IaaS clouds. Some of the supported functionality includes isolation, middlebox functions, QoS as well as the use of existing address spaces which minimizes reconfigurations. FlowComb [88], acts similarly to CloudNaaS but in the Big Data processing applications domain. With the use of application domain knowledge, FlowComb can detect network transfers between application components and proactively or reactively change the network path in order to support those transfers.

2.7.2 Flow Management Related

FlowVisor [89, 90], slices a physical network into abstracted units of bandwidth, topology, traffic and network device CPUs. It operates as a transparent proxy controller between the physical switches of an OpenFlow network and other OpenFlow controllers and enables multiple controllers to operate the same physical infrastructure, much like a server hypervisor (Virtual Machine Monitor VMM) allows multiple operating systems to use the same x86-based hardware. Other standard OpenFlow controllers then operate their own individual network slices through the FlowVisor proxy. This arrangement allows multiple OpenFlow controllers to run virtual networks on the same physical infrastructure. Although SDN research community considers FlowVisor an experimental technology, Stanford University which is a leading SDN research institution, has run FlowVisor in its production network since 2009 [91]. FlowVisor lacks some of the basic network management interfaces that would make it enterprise-grade. It currently has no command line interface or Web-based administration console. Instead, users make changes to the technology with configuration file updates.

Hedera [92], is a scalable, dynamic flow scheduling system that adaptively schedules a multi-stage switching fabric to efficiently utilise aggregate network resources. Hedera has a three step control loop. First, it detects large flows at the edge switches, afterwards, it estimates the natural demand of large flows and uses placement algorithms to compute good paths for them and finally, these paths are

installed on the switches. In order for Hedera to get all the information needed, all the switches used have OpenFlow implemented. Getting flows information, allows Hedera scheduler to redirect a flow entry that grows beyond a specified threshold in a newly chosen path.

Ident++ [93], is a simple protocol to request additional information from end-hosts and networks on the path of a flow, thus making administrators less of a bottleneck when policy needs to be modified and allows network administration to follow organisation lines. Ident++ allows users and end-hosts to participate in network security enforcement by providing information that the administrator might not have or rules to be enforced on their behalf.

OpenQoS [94], uses OpenFlow centralised capabilities in order to create a dynamic QoS routing mechanism that provided end-to-end QoS in order to minimize packet losses and latency. Similar to OpenQoS, PolicyCop [95] is a QoS policy framework. It is autonomic and offers per flow control and dynamic flow aggregation as well as dynamic configuration of traffic classes with the use of a RESTful based API. Its easy API guarantees ease of deployment as well as reduced operational overhead.

OpenTM [96], is a traffic matrix estimation system for OpenFlow networks which uses built-in features provided in OpenFlow switches to directly and accurately measure the traffic matrix with a low overhead. OpenTM is a C++ application designed for NOX OpenFlow controller. In addition, it uses the routing information gained from the OpenFlow controller to intelligently choose the switches from which to obtain flow statistics, thus reducing the load on switching elements. Due to the critical information provided only through OpenFlow, OpenTM is an OpenFlow-only application and cannot be used with any other network. Testbed experimenting showed that OpenTM derives an accurate TM (Traffic Matrix) within 10 switch querying intervals, which is extremely faster than any other existing TM estimation techniques. That result is based solely on information achieved through OpenFlow.

2.7.3 Wireless Related

OpenRoads [5,97–100], also known as OpenFlow Wireless, is a platform for innovation and realistic deployment of services. Mobility services are heavily researched but the verification is hard. Wireless channels are difficult to simulate and realistic user traffic is crucial to solid validation of ideas. OpenRoads brings OpenFlow to wireless networking in order to allow research to take place in production network. OpenRoads has a multi-layer architecture (shown in Figure 2.5) consisting of a physical layer, network virtualisation/slicing layer and controller layer.

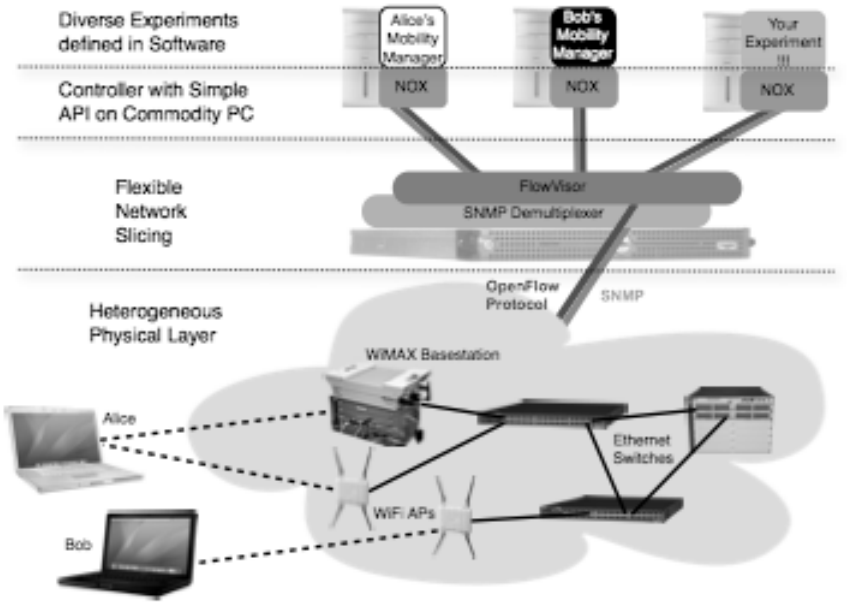


Figure 2.5: OpenRoads Architecture (*Source: [5]*)

AeroFlux [101, 102], proposes a wireless SDN architecture, scalable enough that supports carrier level WiFi deployments. Due to its low-latency programmatic control of transmission settings, it is suitable for better quality of experience in shared wireless medium. Similar to AeroFlux, Mobileflow [76] introduces a software defined mobile network architecture in order to increase the innovation potential in mobile networks. This is due to the use of OpenFlow to create an open architecture that provides an API which developers can use to implement new functionality as well as control the traffic.

Odin [103], presented three new additions in the area of mobile networking that not only help innovation in the current state of wireless networks but it also

makes them future proof. Odin introduced Light Virtual Access Points (LVAP) which is a programming abstraction used to address the complexity in the IEEE 802.11 protocol. This helps in a faster and more efficient design of software defined WiFi networks that can be implemented on top of the current access points without any modification to the underlying IEEE 802.11 protocol. Using Odin, researchers proved its superiority over the current mobile networks state in areas such as load-balancing, jammer detection, mobility management, automatic channel-selection as well as energy management.

Open Radio Access Network (OpenRAN) [104,105] main goal is to create open, controllable, flexible and evolvable radio access network (RAN). This is achieved with a three layered architecture consisting of a wireless spectrum resource pool, a cloud computing resource pool and an SDN controller. Similar to OpenRAN, Software Radio Access Network (SoftRAN) [106] proposes a software defined control plane for RAN that abstract base stations in an area as a virtual big-base station. This allows the deployment of management functionality that increases load balancing, deals efficiently with interference and maximises throughput.

2.7.4 Security Related

Resonance [107], is a system for securing enterprise network, where the network elements themselves enforce dynamic access control policies based on both flow-level information and real-time alerts. Resonance allows switches to dynamically re-map clients based on several inputs like alarms from distributed network monitoring systems. Alert systems control traffic by sending messages to the controller, which in turn controls switch behaviour via the standard, OpenFlow-based switch interface. This keeps on-path forwarding decisions simple, while still allowing complex policies to be implemented through a standard control interface. It uses flow tables that have rules for matching traffic flows to actions, which is the place where OpenFlow becomes important. Switches can use these tables for any given principal, where the table that the switch uses at any given time depends on the security class and the current state of that principal.

OpenSafe [108], is a system that enables the arbitrary direction of traffic for security monitoring applications at line rates. OpenSafe comes with "A Lan-

guage for Arbitrary Route Management for Security" (ALARMS) which is a flow specification language that greatly simplifies management of network monitoring appliances. OpenFlow is a major part of ALARMS, but ALARMS was using OpenFlow version 0.8.9 and almost all of the problems faced and proposed solutions have been implemented in later versions of OpenFlow.

The Flow-based Network Access Control (FlowNAC) [109] proposes a mechanism that allows user rights for accessing a network depending on the service requested, over the IEEE 802.1X standard. The major improvement brought by FlowNAC is the fact that these access rights can be performed by several services at once instead of the current solutions which allow just one. Furthermore, with the use of SDN, it can decouple Port Access Controller (PAC) from the Authentication and Authorization (AA), with the one taking place at the data plane and the other directly at the controller.

FortNOX [110] is an extension to the NOX controller used with OpenFlow and with security policy enforcement kernel. The major strength of FortNOX is the live rule conflict detection engine which is performed by a conflict analysis algorithm. This gives an extra level of security to the OpenFlow controller as well as the network operators. In addition, FortNOX supports role-based authentication for OpenFlow application through the use of digital signatures. FRESCO [111] is an OpenFlow application development framework for security modules. Due to its click-inspired nature, it is very easy to rapid design, implement, share and perform collaborative work on security detection and mitigation modules.

2.8 Summary and Discussions

Summarising everything, we have seen that networks centralised approach began with iBGP which connects all routers to an AS. Then we move to RCP approach which uses a central server to communicate which can communicate with all the network routers. The most revolutionary idea came with 4D which sliced the network into four planes, namely, decision, dissemination, discovery and data plane. Following 4D, SANE and Ethane based their ideas on 4D's approach of splicing the network into planes. The first approach that actually implemented 4D's idea

was Tesseract. NOX and Maestro used SANE and Ethane approach of a network operating system and provided a more advanced network OS. HyperFlow came to extend NOX into a distributed control plane. DIFANE on the other hand provided a different approach to improve flow-based networks' control plane performance, through the proactive computation of wildcard matching rules based on high level policies.

After a lot of research OpenFlow emerged as the first SDN protocol. The main advantage of OpenFlow is that it is maintained by ONF which includes large networking enterprises such as Level(3), Google, Juniper, Verizon, Oracle, Microsoft, Cisco, HP and IBM. ONF takes OpenFlow seriously which is something that one can spot at the rate of implementation of OpenFlow versions. On the other hand this fast evolving nature of OpenFlow becomes one of its disadvantages since software and hardware vendors cannot cope at such implementation speeds. OpenFlow has found many uses such as PortLand which creates a scalable fault-tolerant data centre network fabric. Ripcord provides a platform for rapidly prototyping scale-out data centre networks. Using OpenFlow, FlowVisor slices a physical network into abstracted units of bandwidth, topology, traffic and network device CPUs. Hedera on the other hand used OpenFlow to provide a scalable, dynamic flow scheduling system that adaptively schedules a multi-stage switching fabric to efficiently utilise aggregate network resources.

Furthermore, OpenFlow found uses in wireless networks with OpenRoads, which is a platform for innovation and realistic deployment of services. Resonance and OpenSafe on the other hand used OpenFlow for network security purposes. Finally OpenTM is a traffic matrix estimation system for OpenFlow networks.

It is obvious that network research is moving towards SDN approach. Sooner or later enterprises will become confident about SDN and they will start implementing it in their own networks and data centres. Google went a step further and is the first enterprise company to adapt SDN in their current networks. According to Google's senior vice president of technical infrastructure Urs Hölzle [112], the two large backbones of Google, namely Internet-facing backbone and Data centre backbone, have adapt SDN architecture. Dr. Hölzle stated that, due to this move Google benefited from up to 50 times better performance in several areas. And to

realise how big these two backbones are, according to ATLAS 2010 annual traffic report, "If Google were an ISP, as of September 2010 it would rank as the second largest carrier on the planet". Unfortunately, Google does not provide any details on how it is using it and what the performance is on several aspects. Therefore enterprises that may not have the resources Google has, are not confident in moving to SDN just to see if the performance is better. Enterprises want to know that the performance will be better before investing their resources on new equipment and new structures.

Chapter 3

OpenFlow Performance Enhancement Algorithm Using Dynamic Flow Installation And Management (OFPE)

3.1 Introduction

In the OpenFlow protocol (described in Section 2.3), the control plane of a networking device is moved to a separate hardware called the controller. One of the disadvantages of such an approach is the fact that the network performance is dependent on the individual performance of the switch, the controller and the link that connects them.

More specifically, the controller as well as the link that connects it with the switch are affecting the performance of the network every time a packet-in event occurs. Packet-in is a message sent by the switch to the controller containing a captured packet. This happens if the switch is programmed to do so or if the packet arriving at the switch does not match any flow table rule. In both cases, the switch will forward the packet to the controller which will take a decision. The controller can either drop the packet or reply back to the switch with a rule to be installed in the flow table. This rule indicates to the switch how to deal with the rest of the packets that have identical headers to the initial packet, until the rule

expires.

This procedure will add an extra round-trip time (i.e. the time it takes for a packet or packet header to travel from the switch to the controller and return back) for each packet that takes the diverted route. Only one packet from each flow takes the diverted route if we bear in mind the following three assumptions:

1. The packets arrive at a rate equal or lower than two parameters combined. a) the rate the controller handles each packet and b) the time it takes for each packet to travel to the controller and back to the switch

2. The hard timeout of each flow table rule installed is equal to or greater than the duration of the flow (hard timeout is the time it takes for the rule to expire)

3. The idle timeout of each flow table rule installed is equal to or greater than the rate at which the packets of the flow arrive (idle timeout is the inactivity time required for a rule to expire)

In any other case, more than one packets take the diverted route, resulting in a decrease in performance that under some circumstances can cause serious problems with the two most important being:

1. The increase in delay due to the extra round trip time between the switch and the controller

2. The increase in the number of "out-of-order" packets

This chapter analyses these two problems and the effects they have on the performance of OpenFlow networks through a series of experiments. It then proposes, a novel flow installation and management algorithm OFPE, which improves the performance of OpenFlow networks overcoming the limitations discussed in this chapter.

The remainder of this chapter can be summarised as follows:

- Section 3.2 (Current State): This section presents the current state in OpenFlow networks and their performance limitations.

- Section 3.3 (Proposed OpenFlow Performance Enhancement Algorithm): This section presents the proposed OpenFlow performance enhancement algorithm together with an explanation of every operation used in the algorithm.

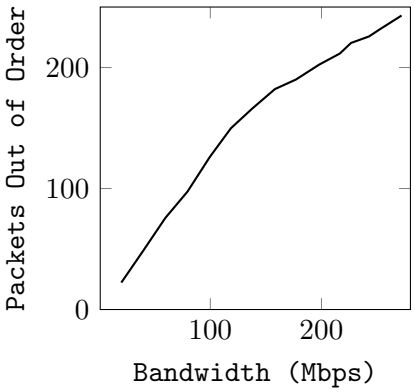
- Section 3.4 (Experiments and Analysis): This section presents a number of experiments performed in order to examine the proposed algorithm performance and compare it with scenarios that make no use of the proposed algorithm.

- Section 3.5 (Summary and Discussions): This section gives a summary of the proposed algorithm, indicating its strengths and the most significant performance improvements achieved.

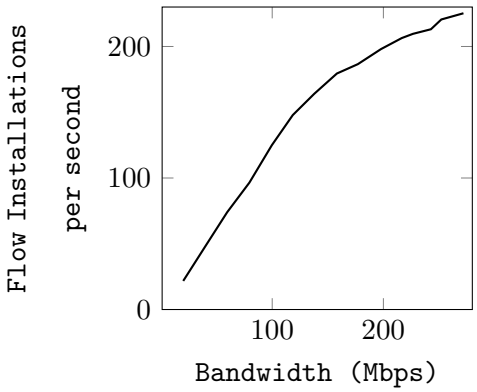
3.2 Current State

With the current data transfer speeds [1] the assumption stated in Section 3.1 (i.e. packet arrival rate is lower or equal to the rate that the controller handles each packet) is very unlikely to happen in enterprise networks. As a result, more than one packet will have to travel to the controller for decision making thus increasing the delay due to the extra round trip time. This added delay is not constant; it depends on the controller's performance as well as the number of packets in the queue waiting to be examined by the controller. Thus as the interarrival time of the packets decreases, delay increases as shown in Figure 3.1c.

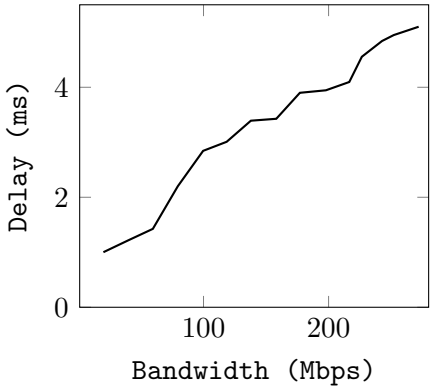
Furthermore, another problem that appears due to multiple packets traveling to the controller, has to do with the order the packets are reaching their destination. It is a problem that takes place mainly in UDP traffic, it happens in TCP as well, but it needs some appropriate conditions in order to affect the TCP traffic. Even though TCP dominates Internet traffic, some new applications tend to favour UDP [113], which is why UDP traffic is also important. The reason behind such a problem is the fact that, the OpenFlow switch has no queues for the packets that are waiting for the controller to take a decision.



(a) Number of Packets Out of Order as Bandwidth Increases



(b) Flow Installations per Second as Bandwidth Increases

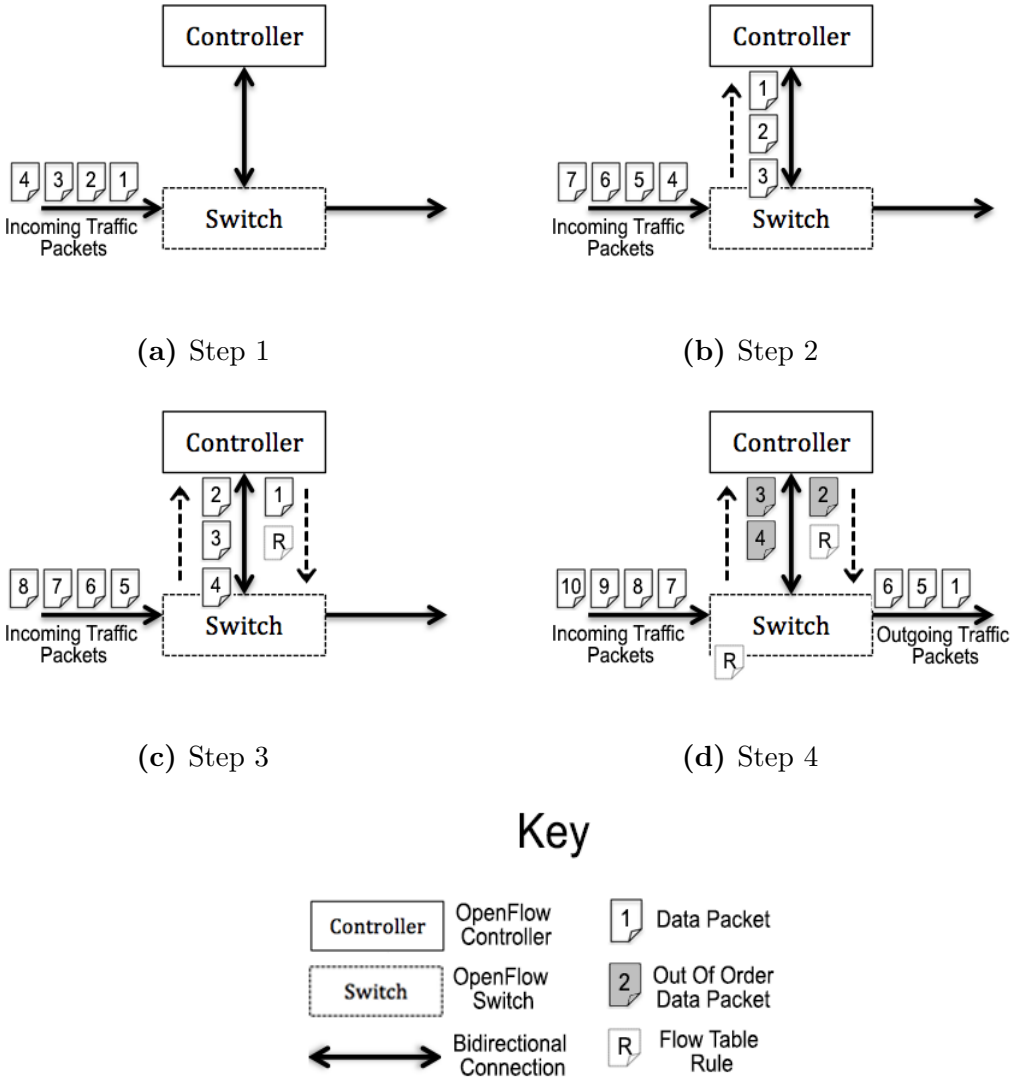


(c) Delay as Bandwidth Increases

Figure 3.1: Packets Out of Order Relation to Packets Rate of Arrival

Instead of a queue, the switch has a buffer where it can save the packets and forward their headers as well as part of the payload to the controller until it has a rule that fulfils the packet's parameters (Figures 3.2a and 3.2b). If the buffer is full, then the whole packet is forwarded to the controller. The controller will then examine the first packet's headers and return it back to the switch together with its decision (Figure 3.2c). Assuming that the decision is a flow table rule installation, then it proceeds with the installation. Every subsequent packet arriving at the switch will get forwarded to the destination. However, by that time, there are still packets waiting at the controller and buffer whereas newly arrived packets of the same flow get forwarded to the destination. As a result, several packets reach the destination out-of-order (Figure 3.2d). The problem of the packets arriving at the destination out-of-order was addressed at Open Networking Summit in 2012

during the presentation of Frenetic [114], a high-level programming language for OpenFlow networks. To the best of our knowledge, no further work has been done since then, for solving this problem.



(a) Step 1 **(b)** Step 2

(c) Step 3 **(d)** Step 4

Figure 3.2: Out Of Order Packets Explanation

In addition to the out-of-order packets problem, another problem will take place which is the installation of duplicate flow table rules. The controller has no knowledge of the rules installed in the flow table, therefore for every packet received, it will create a new rule and send it to the switch. Thus in the case shown in Figure 3.2, the controller will create several duplicate rules. Such a situation happens if some packets are waiting for a decision by the controller and

the flow table rule for those packets has a timeout time greater than the time needed by the controller to process each of the packet waiting. A solution to this is instructing the controller to check for the flow table rules installed on the switch every time it has to install a new flow table rule. However, this can also decrease the performance of the network as stated by Curtis et al. [115], because gathering information from the switch at such small time intervals will create too much control plane load leading to an increase in delay.

An experiment was contacted to confirm the relation of packet arrival rate with (a) the number of packets out-of-order, (b) the number of duplicate flow installations and (c) the delay. The topology as shown in Figure 3.3 consisted of two virtual machines (summarised in Table 3.1), one of them running Mininet [82] emulator and the other running POX [116] as the controller. Mininet was used to create a virtual network with two Virtual hosts and an OpenFlow switch (Open vSwitch [117]).

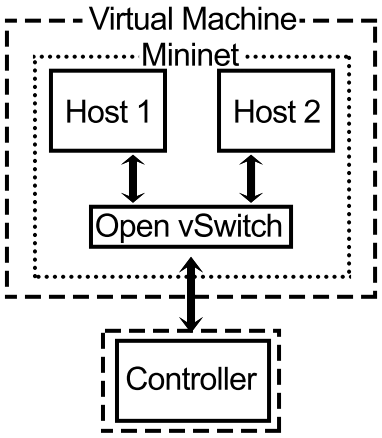


Figure 3.3: Packet Arrival Rate Importance Experiment Topology

A separate virtual machine was utilised for the controller to prevent Mininet CPU consumption from affecting the controller's performance. Also, in a real world environment, the controller and the switch are two different and independent entities that have to communicate between them using a dedicated link. In the experiment, UDP traffic was travelling from *Host 1* towards *Host 2* at predetermined rates for a duration of 300 seconds. The timeout out time of each of the flow table rules was set to 1-second hard timeout. For each of the predetermined rates, the experiment was repeated 30 times to decrease experimental error.

OS	Ubuntu 14.04.1 LTS
Kernel version	3.13.0-32
Architecture	x86-64
Cores	2
CPU (GHz)	2.40
Cache (MB)	4
RAM (MB)	2048
Virtualisation	VT-x
Hypervisor	QEMU KVM

Table 3.1: Virtual Machines Specifications

The results shown in Figure 3.1 and summarised in Table 3.2, indicate that:

1. The number of out-of-order packets arriving at the destination is proportional to the rate at which the packets arrive at the switch, thus increasing the rate, increases the number of packets out-of-order.

2. The number of flow installations per second is proportional to the rate at which the packets arrive at the switch. Theoretically, the number of flow installations should always be 1 per second since the hard timeout used in the experiment is 1 second, but as explained in Figure 3.2 this is not the case.

3. The delay is proportional to the rate at which the packets arrive at the switch.

The effects listed above, become more severe as the number of switches and flows increases. Each packet-in event will take place “n” times, where “n” is the number of OpenFlow switches that the packet has to travel through until it reaches its destination. Furthermore, the more the number of switches, the higher the overload caused to the controller, due to the increased number of packets waiting for decision. This will increase the time it takes for the controller to take a decision, therefore it will increase the overall delay. The main reason that both delay and out-of-order packets increase as the controllers CPU load increases is

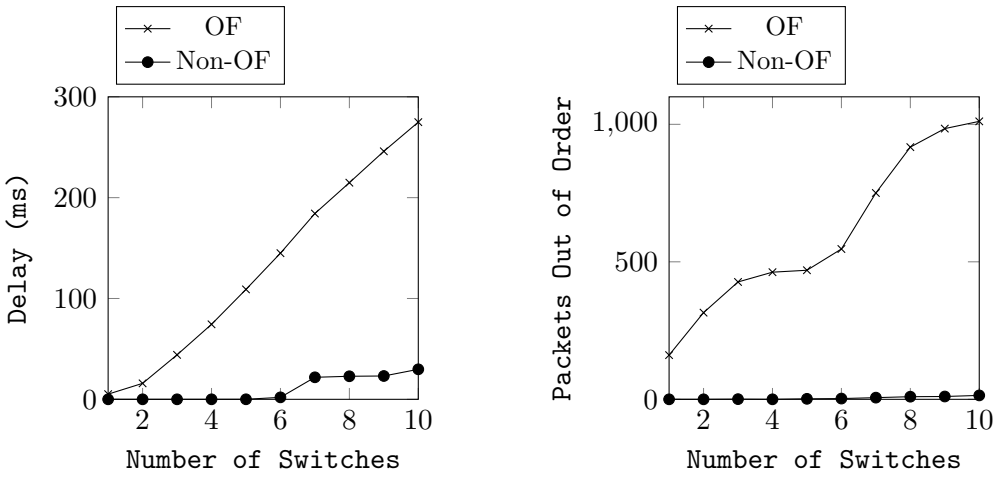
Bandwidth (Mbps)	**Packets Out of Order (s^{-1})**	**Flow Installations (s^{-1})**	**Delay (ms)**	**Packets Sent (s^{-1})**
19.97	22.26	21.73	1.00	1698.18
39.80	48.78	48.01	1.22	3388.20
59.50	75.63	73.99	1.43	5080.10
79.46	97.44	96.46	2.20	6770.30
99.70	126.27	124.97	2.84	8492.40
118.52	149.80	147.86	3.01	10110.30
137.88	166.21	164.07	3.39	11780.30
158.19	182.22	179.45	3.43	13519.30
177.06	190.09	186.58	3.90	15151.80
197.86	202.37	198.02	3.95	16950.60
216.54	211.58	206.40	4.10	18548.50
226.59	220.32	209.61	4.56	20350.20
242.75	225.69	213.10	4.84	20918.40
252.11	231.38	220.54	4.95	23340.50
272.02	242.97	225.26	5.10	23743.10

Table 3.2: Packets Out of Order Relation to Packets Arrival Rate Experiment Summary

that currently the controller has an operating system which is not specifically designed for OpenFlow. As a result, the scheduler of that operating system will split the available CPU time evenly to all the processes that have to be executed by the controller. The process priority (like "niceness" value in Linux) cannot solve the problem due to the fact that in a centralised environment each controller will control more than one switch. This means that there will be several packets and subsequently processes that will be in the queue waiting for some processing time. Giving higher priority to one of those processes will subsequently affect the rest of the processes.

A performance comparison between OpenFlow and Non-OpenFlow switches, conducted using Mininet, indicates this reasoning as shown in Figure 3.4.

Finally, OpenFlow switches have a limited number of entries in their flow table. The number of flow table entries allowed in a physical OpenFlow switch varies from vendor to vendor and also depends on the matching fields used by OpenFlow. For the typical 12-tuple table matching fields used in OpenFlow v1.0 [3] (OpenFlow v1.4.0 [43] has more than 40 matching fields), the number of flow table entries



(a) Delay **(b)** Packets Out of Order

Figure 3.4: OpenFlow Vs Non-OpenFlow Switches

are in the range of just below a thousand [118] to about five thousand. If the switch is required to match only the Layer-2 fields, then the number of flow table entries is in the range of 32,000 to 160,000. Typically, vendors will only state the Layer-2 flow table entries number without mentioning anything about the 12-tuple number. Table 3.3 shows the flow table size for some OpenFlow Switches. On one hand this limitation can be considered as an advantage because having a big number of flow table rules, means that the response time of the switch will be greater since it will have to scan through a big list of rules. On the other hand, it can be considered as a disadvantage because it can easily get full and a flow table that is full is worse than having no flow table at all because the controller will be forced to perform further actions as listed below:

- Gather flow statistics directly from the switch to become aware of the rules that have to be removed.
- Inform the switch of those rules that have to be removed.

These actions will result in additional delay and traffic in the link between the controller and the switch.

Switch	**Number of Flow Table Entries**	**Throughput** (Gbps)
Brocade MLX 24-Port 10 GbE	Up to 32,000 (Layer-2) [119]	200
Brocade MLX 20-Port 10 GbE	Up to 56,000 (Layer-2) [120]	400
NEC PF5240	Up to 64,000 (Layer-2) for Real Switch (RSI) and up to 160,000 (Layer-2) for Virtual Switch (VSI) [121]	176
NEC PF5820	750 (12-tuple) and 80,000 (Layer-2) [118]	1280
Arista 7050	Up to 128,000 (Layer-2) [122]	1280
Arista 7150	Up to 64,000 (Layer-2) [123]	480
Arista 7300	Up to 32,000 (Layer-2, per module) [124]	2560
Arista 7500	Up to 128,000 (Layer-2) [125]	15000

Table 3.3: OpenFlow Physical Switches Flow Table Size

3.3 Proposed OpenFlow Performance Enhancement Algorithm

The purpose of the proposed algorithm is to efficiently increase the performance of SDN by decreasing delay, packet loss as well as the number of packets out-of-order. In order to achieve this in the best possible way, the controller will have to perform a number of operations. These operations are divided into two categories: a) *Common Operations* and b) *Topology Specific Operations*. The common operations are the ones that are always used, whereas the topology specific operations are the ones that may be used depending on the topology. In addition, it allows networks administrators to list special flows either before the algorithm initialisation or on an ad-hoc basis. These special flows are handled as higher priority ones and they get higher performance than the rest of the flows when the network is under stress. Listed in Section 3.3.1 are the operations performed by the controller whereas in Section 3.3.2 the benefits of those operations are described.

3.3.1 Algorithm Operations

1. Common Operations

   - **CPU Monitor**: Constant monitoring of controller's CPU usage at a frequency equal to the smallest timeout time set for a flow table rule. If the available CPU percentage is less than a predefined value (i.e. the controller is overloaded), it increases the hard timeout time of the high priority flows as shown in Figure 3.5.
   - **Flow Modification**: Removal or adjustment of flow table rules of high priority flows. If a high priority flow gets removed from the high priority list (held by the controller), the controller performs a flow table rule removal or adjustment as shown in Figure 3.6.
   - **Flow Table Statistics**: Gather statistics from the switch at predefined time intervals as shown in Figure 3.7. The controller dynamically adjusts the predefined time intervals. If the number of available flow table rule spaces decreases, then the controller gathers statistics more often to prevent flow table from getting full.

2. Topology Specific Operations

   - **Network Topology Awareness**: Network administrator adds as a parameter a list of all the hosts that are connected to the switches, as well as the switch to switch connections. After the topology is known, the controller examines those information and analyses the topology of the network as shown in Figure 3.8.
   - **Route Formation**: Upon receiving an "unknown" packet, the controller will install flow table entries for all the switches that the packet will travel through to reach its destination. Except for the initial switch, the flow table entries that will be installed will have their timeout time given as idle timeout. For the first switch, the timeout time will be given as hard timeout. The route formation is performed as shown in Figure 3.9.

```
Require: min_timeout: Smallest timeout time set for a rule
Require: cpu_limit: CPU percentage of overload
1: procedure CPU_MONITOR(flows)
2: loop:
3:     cpuusage = READ CPU_USAGE
4:     database += cpuusage
5:     if cpuusage > cpu_limit then
6:         foreach flows as flow
7:             FLOW_MOD (flow,idle,hard)
8:     Wait(min_timeout)
```

Figure 3.5: CPU Monitor Algorithm

```
1: procedure FLOW_MOD(flow_details,idle,hard)
2:     msg = of.ofp_flow_mod()
3:     msg.match = flow_details.fields
4:     msg.hard_timeout = hard
5:     msg.idle_timeout = idle
6:     send_to_switch(msg)
```

Figure 3.6: Flow Modification Algorithm

```
Require: time_interval: Predefined time in seconds
1: procedure FLOW_TABLE_STATS
2: loop:
3:     while (respond=false) do
4:         send_to_switch(Req_Stats)
5:         Wait(timeout_time)
6:     database += respond
7:     Wait(time_interval)
```

Figure 3.7: Flow Table Statistics Algorithm

3.3.2 Operations Benefits

In both single and multi-switch topologies, the controller constantly monitors its CPU usage at a rate equal to the shortest timeout time set for a flow table rule. In the case that the CPU usage rises to a point where it indicates overload, then the controller increases the hard timeout times of the new incoming high priority flows. As a result, the following two goals are achieved:

```
Require: node_links: Parameter list of all the hosts and their con-
    nections
 1: procedure NET_TOPO_AWARE
 2:     foreach node_links as link
 3:         source_a = link.Starting_Address
 4:         foreach node_connections as link1
 5:             source_b = link1.Starting_Address
 6:             if (source_a == source_b ) then
 7:                 destination = link1.Ending_Address
 8:                 Array(end_nodes).append(destination)
 9:         Array(Connections)=[source_a,end_nodes]
10:     database += Connections
```

Figure 3.8: Network Topology Awareness Algorithm

```
Require: src_addr: Source address
Require: dst_addr: Destination address
Require: link: Links between nodes
 1: procedure ROUTE_FORMATION(link, src_addr,dst_addr)
 2:     queue = [(src_addr, [src_addr])]
 3:     while queue do
 4:         (link.node, path) = queue.pop(0)
 5:         for next in link - set(path) do
 6:             if next == dst_addr then
 7:                 yield path + [next]
 8:             else
 9:                 queue.append((next, path + [next]))
```

Figure 3.9: Route Formation Algorithm

- It ensures that the high priority flows are still served and they will not face increases in delay as well as out-of-order packets if the controller response time increases (i.e. CPU is overloaded).

- It ensures a decrease in the controller's CPU usage thus allowing the controller to reach a more efficient state. This is achieved by serving less flow table rule installations because increasing the timeout time means that flows take longer to expire and revisit the controller.

Gathering flow statistics as well as removing high priority flows that cease to be part of the high priority list, are both measures to prevent flow tables from getting full. Statistics gathering is performed in a way to prevent overloading the link between the switch and the controller. The controller gathers statistics in predefined (long) time intervals. If the number of flow table entries closely approach

the maximum flow table entries allowed then the controller gathers statistics more often until the flow table entries are decreased. This results in a proactive controller, that knows an approximation of the number of flow table entries at any given interval, without having to get live statistics. The network benefits in two ways.

- No extra delay until the controller analyses the flow statistics received from the switch.
- It does not cause significant traffic increase in the link between the controller and the switch if it is not necessary. It only causes a significant traffic increase if the readings indicate that it has to act to preserve network's quality of service.

Finally, in a multi-switch environment, the controller is aware of the network topology with the use of the Network Topology Awareness operation. OpenFlow Discovery Protocol (OFDP) might have been used for topology discovery, but is not the best solution because it has some serious security and efficiency limitations [126]. With the method used, when it comes to complexity, using the big O notation, this operation will take $O(n \times c)$ time to complete. This is because the algorithm will have to visit once each member of the inputted list of each of the switches in the topology. If the number of members of the list is indicated by n and the number of switches by m, then the time needed is $O(n \times c)$. After the topology discovery, the algorithm proceeds with the Route Formation operation which is using a breadth-first search algorithm that searches through connections to find the best path to the destination. If we say that n is the number of nodes and c is the number of connections of each node, then each node n will be enqueued or dequeued at most once therefore using big O notation it takes O(n) time. Also for each of those nodes, scanning through their connections takes O(c). As a result, the whole algorithm takes $O(n \times c)$. Using these two operation together, allows the controller to install flows in a more efficient way. When a packet arrives at a switch and subsequently forwarded to the controller, the controller installs flow table rules for all the switches the packet (and subsequently the flow) has to travel to reach its destination. For the first switch it uses a hard timeout time, and for

the subsequent switches, it uses an idle timeout equal to the time used as a hard timeout in the first switch. This methodology not only reduces delay (since the packet travels to the controller just once until it reaches its destination), but it also decreases the number of out-of-order packets drastically.

3.4 Experiments and Analysis

The experimental phase consisted of nine different scenarios; all of them have been performed with and without the use of the proposed algorithm for comparison. The initial four scenarios were used to tests and optimise the algorithm's operations in an individual basis whereas the rest of the scenarios tested the algorithm as a whole in more realistic topologies. Performance metrics such as overall bandwidth, average delay, the number of packets out-of-order, the number of packets lost and the total number of packets sent were recorded. All of the experiments were repeated 30 times to and the average reading was calculated in order to decrease the experimental error.

3.4.1 Experimental Equipment

Due to limited equipment resources, a testbed was only used for scenarios 1 and 2 whereas for the rest of the scenarios a server with multiple virtual machines was used to emulate an experimenting network. The testbed environment consisted of HP Procurve 3500-24 switch [127], and three physical machines (system information in Table 3.4, two of them used as clients (a sender and a receiver) and one of them as a POX controller.

In the emulation environment, for scenarios 3 and 4 two independent virtual machines were used (details are summarised in Table 3.1), one running POX controller and the other running Mininet. In each experiment, the controller's CPU usage was recorded, the same way the algorithm records the CPU usage, using Linux command "mpstat". For traffic generation, Iperf [128] was used, generating UDP traffic.

Component	Details
Processor	4 Cores at 2.4GHz
Microprocessor Cache	4MB Level 2 Cache
RAM	8GB DDR3 1066MHz
Hard Drive	32GB SATA 5400rpm
Operating System	Ubuntu 14.04.1 LTS

Table 3.4: Server Experimenting Machine

3.4.2 Scenario 1 - Incremental Increase of CPU Load

In the control experiment in which no algorithm was used, Host **H1** was sending a 75Mbps UDP stream to host **H2**, and the hard timeout of each flow table entry was set at 1 second. Every 50 seconds the CPU usage/load was increased by 20% until it reached 100%. The experiment was then repeated with the use of the proposed algorithm. Once the CPU usage exceeded 75%, then the algorithm started working by increasing the hard timeout as well as perform all the other operations. The reasoning behind the 75% CPU usage is the fact that in all of the single switch experiments, above 75% in POX controller CPU the performance of the network becomes very unstable. This is, of course, a scenario specific value and it can be changed according to the scenario needs.

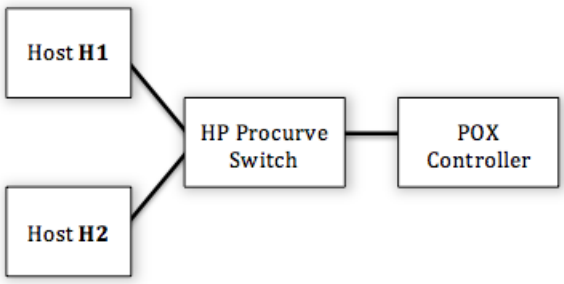
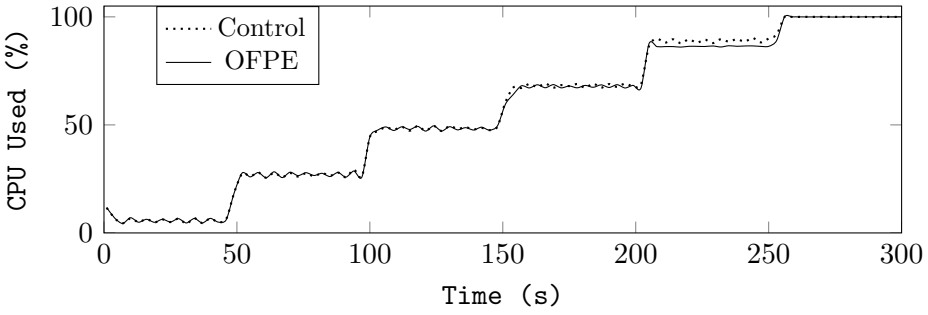
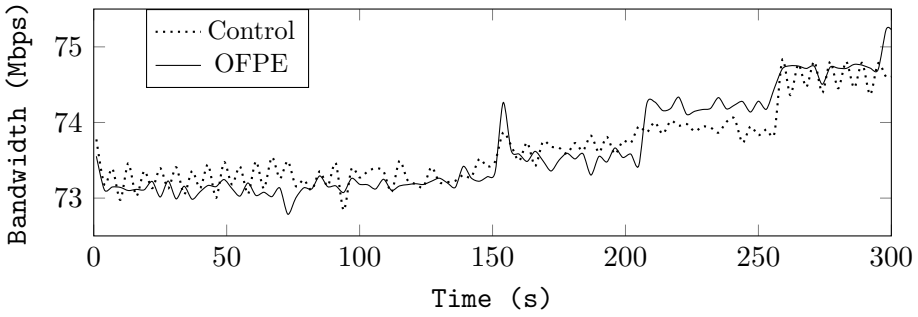


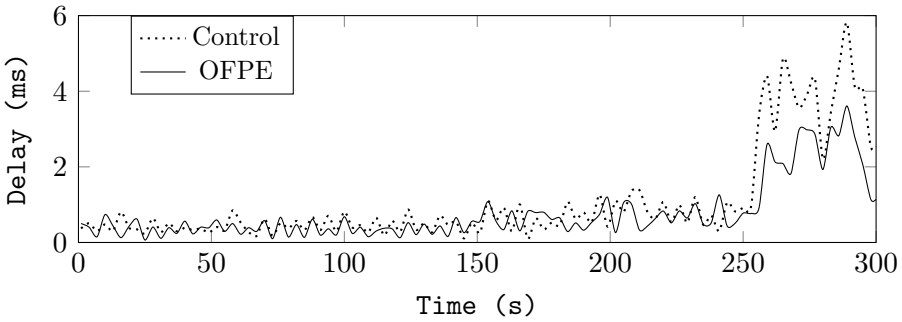
Figure 3.10: Scenarios 1 and 2 Topology



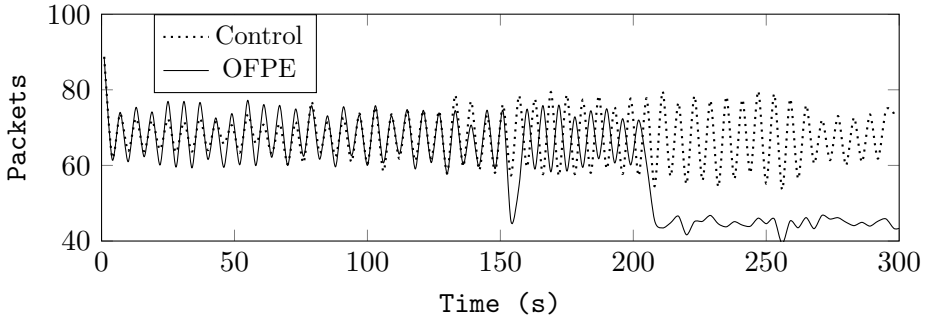
(a) CPU Usage



(b) Bandwidth



(c) Delay



(d) Packets Out-of-Order

Figure 3.11: Scenario 1 - Results Graphs

	Control	OFPE	Change (%)
Bandwidth (Mbps)	73.65	73.65	0
Average Delay (ms)	1.07	0.77	28.04
Packets Out Of Order (%)	1.01	0.90	10.89
Packet Loss (%)	2.10	2.09	0.48
Average Number of Packets Sent s^{-1} (10^3)	6.71	6.71	0
Total Number of Packets Sent (10^6)	2.01	2.01	0

Table 3.5: Scenarios 1 Readings Summary

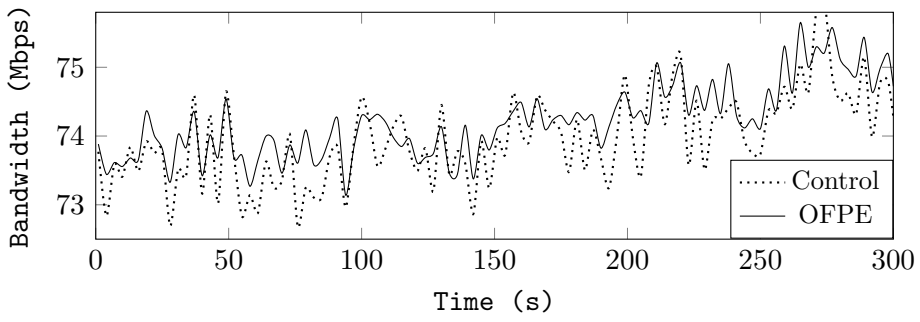
3.4.3 Scenario 2 - Incremental Increase of CPU Load with 4 Streams

This scenario shared most of the characteristics of scenario 1. The only difference was the fact that four streams of UDP traffic was used in order to increase everything by a factor of four, causing more overload to the controller due to the increased amount of traffic it had to handle. This also increased the amount of traffic on the link between the switch and the controller.

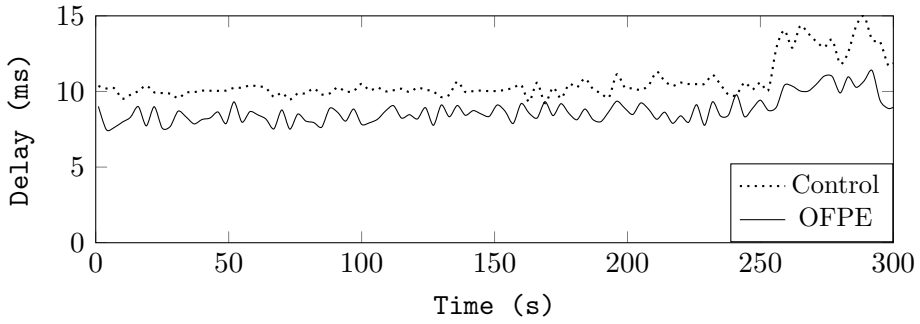
	Control	OFPE	Change (%)
Bandwidth (Mbps)	73.58	73.61	0.04
Average Delay (ms)	10.91	8.93	18.15
Packets Out Of Order (%)	1.42	1.37	3.52
Packet Loss (%)	2.32	2.25	3.02
Average Number of Packets Sent s^{-1} (10^3)	26.72	26.72	0
Total Number of Packets Sent (10^6)	8.02	8.02	0

Table 3.6: Scenario 2 Readings Summary

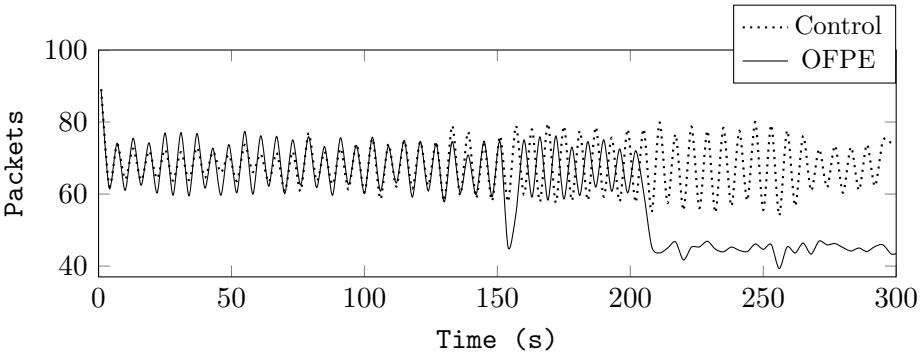
Using scenarios 1 and 2 the importance of altering the timeout time during CPU overload was tested. As indicated in Section 3.2, the readings during overload periods (after the 200th second) became very unstable, as shown in scenario 1 results in Figures 3.11b, 3.11c, and 3.11d. This is because the controller takes a longer time to reach its decision since the CPU is overloaded and the operating system scheduler has more processes waiting for processing time. Therefore, the controller's CPU scheduler splits the processing time evenly, and thus the controller will take less portion of processing time than in a non-overloaded en-



(a) Bandwidth



(b) Delay



(c) Packets Out-of-Order

Figure 3.12: Scenario 2 - Results Graphs

vironment. The same happens in scenario 2, even though the number of streams was increased to four. After the algorithm begun at the 200th second, packets out-of-order began to decrease rapidly as shown in Figure 3.12c. The same happens for delay as well as bandwidth as shown in Figures 3.12b and 3.12a.

Tables 3.5 and 3.6 show that there is an improvement in performance when the proposed algorithm is used. The improvement in performance could be much higher but due to the fact that the algorithm begun working after the 200th, the increase is relatively small. Even though the proposed algorithm acted for only one-third of the experiment's duration; in scenario 1 it has managed to decrease

delay by 0.30ms which is 28.04% change. Also, it has decreased the number of packets out-of-order by 10.89% and decreased the total number of packets lost by 0.48%. In scenario 2 delay was decreased by 18.15% and both out-of-order and lost packets by 3.52% and 3.02% respectively.

3.4.4 Scenario 3 - Multi-Switch Environment

In the control experiment of scenario 3, five switches were connected sequentially (in series one next to the other), and host **H1** was sending a 75Mbps UDP stream to host **H2**. The hard timeout time was set at 1 second, and every 50 seconds the CPU usage/load was increased by 20% until it reached 100%. In the experiment where the proposed algorithm was used, during initialisation the topology awareness as well as the route formation operations are in action. Once the controller received a packet for decision making from the first switch it automatically installed a flow table entry of 1-second hard timeout for the first switch and 1-second idle timeout for all the subsequent switches that the packet was going to travel through to reach its destination.

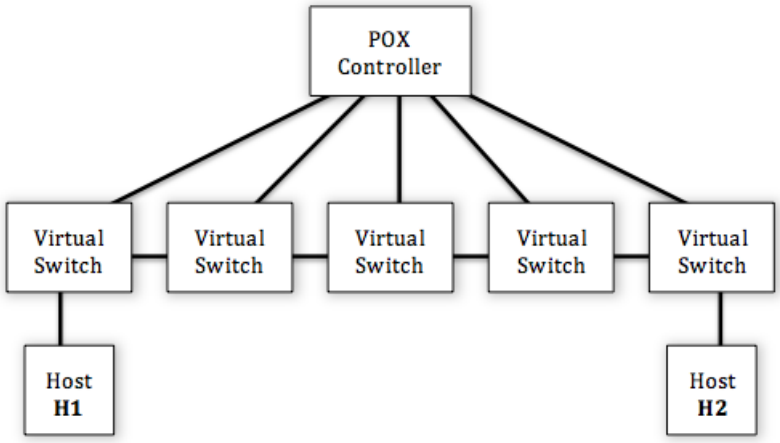
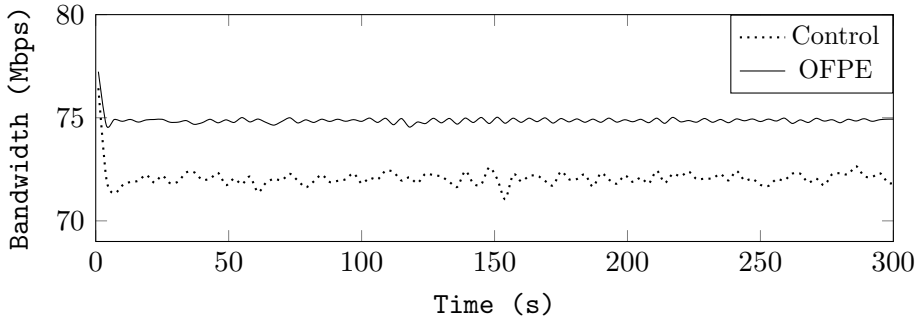


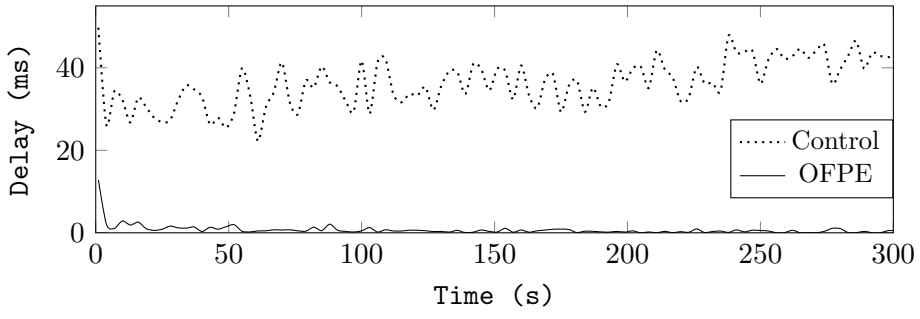
Figure 3.13: Scenarios 3 and 4 Topology

3.4.5 Scenario 4 - Multi-Switch Environment with 4 Streams

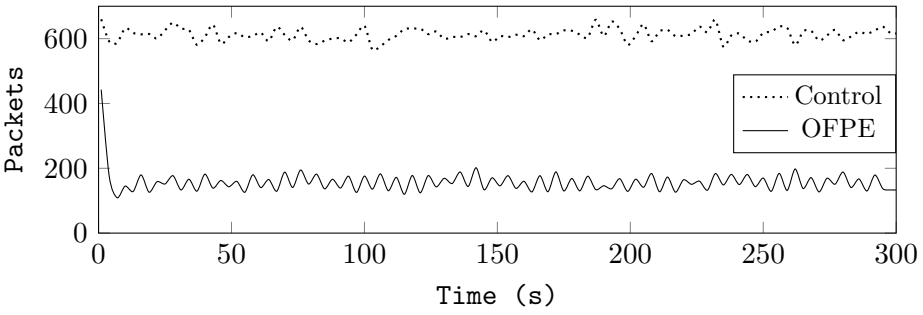
This scenario shared most of the characteristics of scenario 3, with the only difference being that four UDP streams were used. This caused the same stress on the controller and the link between the switch and the controller as scenario 2 but



(a) Bandwidth



(b) Delay



(c) Packets Out-of-Order

Figure 3.14: Scenario 3 - Results Graphs

	Control	OFPE	Change (%)
Bandwidth (Mbps)	72.08	74.88	3.88
Average Delay (ms)	36.23	0.72	98.01
Packets Out Of Order (%)	9.21	2.36	74.38
Packet Loss (%)	6.74	2.11	68.69
Average Number of Packets Sent s^{-1} (10^3)	6.66	6.69	0.45
Total Number of Packets Sent (10^6)	2.00	2.01	0.50

Table 3.7: Scenario 3 Readings Summary

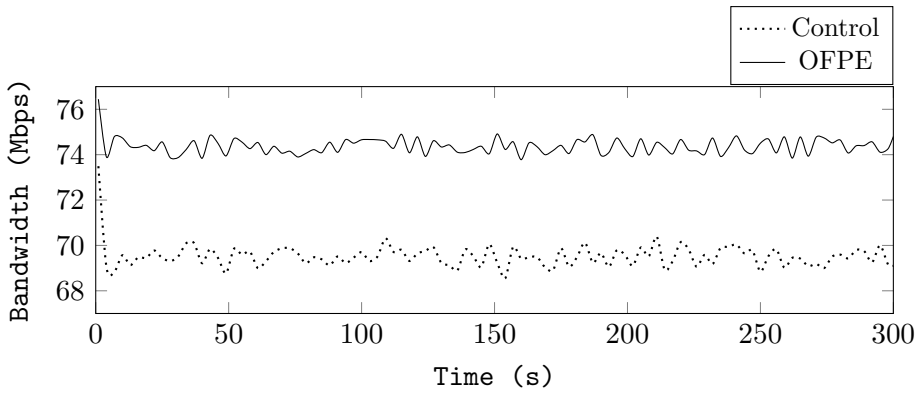
this time the effect was five time greater due to the fact that five switches were present in the topology.

	Control	OFPE	Change (%)
Bandwidth (Mbps)	71.10	73.60	3.52
Average Delay (ms)	125.63	9.35	92.56
Packets Out Of Order (%)	11.91	3.64	69.44
Packet Loss (%)	9.31	4.16	55.32
Average Number of Packets Sent s^{-1} (10^3)	26.51	26.70	0.72
Total Number of Packets Sent (10^6)	8.00	8.02	0.25

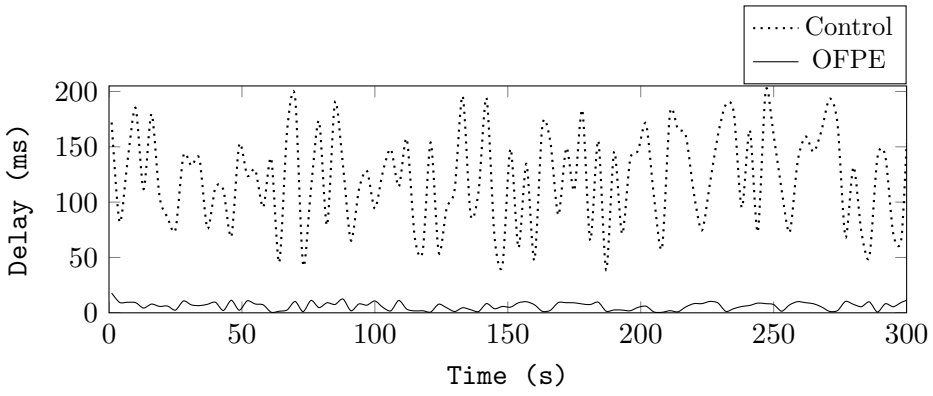
Table 3.8: Scenario 4 Readings Summary

The most significant performance improvement came in scenarios 3 and 4. In those scenarios, the performance enhancement algorithm used the common operations as well as the topology specific operations, namely the topology awareness and route formation operations. As shown in Figures 3.14a, 3.14b, 3.14c and 3.15a, 3.15b, 3.15c the improvement in performance, obtained is significant. One can even compare this performance with a non-OpenFlow environment (shown in Figures 3.4a and 3.4b) since the results are relatively close.

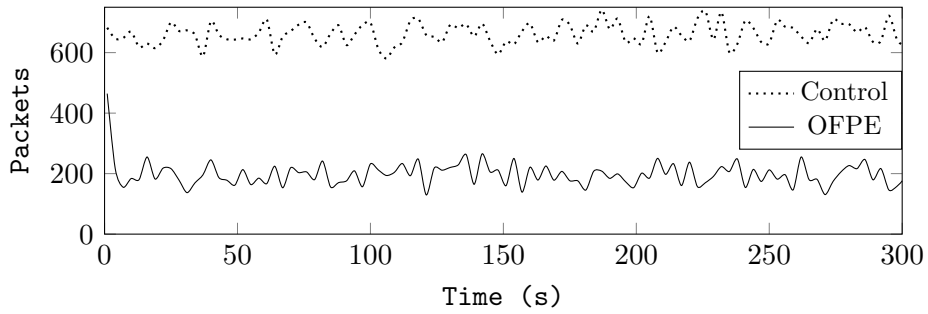
Looking at the scenario 3 as summarised in Table 3.7 using the proposed algorithm a bandwidth increase of 2.8Mbps or 3.88% was achieved, 98% decrease in delay, as well as a reduction in the number of out-of-order and lost packets by 74.38% and 68.69% respectively. The same effects were observed in scenario 4 as summarised in Table 3.8. Delay was decreased by 92.56%, out-of-order packets by 69.44%, and packet loss by 55.32%.



(a) Bandwidth



(b) Delay



(c) Packets Out-of-Order

Figure 3.15: Scenario 4 - Results Graphs

3.5 Summary and Discussions

This chapter proposed a new OpenFlow Performance Enhancement Algorithm which uses dynamic flow installation and management techniques. Its main goals are to decrease the delay caused by the extra round trip time between the controller and the switch introduced in OpenFlow networks as well as the number of packets arriving out-of-order to the destination. In addition, the algorithm has to make

sure that there is no risk of losing some of the OpenFlow benefits such as the per flow control throughout the network.

As shown in Section 3.4 the proposed OFPE Algorithm increased the performance of OpenFlow networks (especially in the multi-switch topologies) without losing the per flow control throughout the network. The improvement in performance came in the form of bandwidth increase and decrease in delay, packet loss and in the number of packets arriving out-of-order at the destination. In some cases it has managed to increase bandwidth by 3.52% and decrease delay by 92.56%, packet loss by 55.32% as well as out-of-order packets by 69.44%.

Currently, some of the algorithm parameters are static. This means that the network administrator has to provide the controller with several details, or fine tune them in order for the algorithm to reach in optimal performance according to the scenario needs. This poses some serious drawbacks such as the fact that if the topology changes (i.e. a node is added or removed), then the network administrator will have to update the controller's information. A dynamic approach will solve such problems since the controller will be able to actively determine the topology and its characteristics and update the rules accordingly.

Chapter 4

OpenFlow Performance Enhancement Algorithm Based on Packet Interarrival Time (OFPEX)

4.1 Introduction

In Chapter 3, we discussed the fact that *(a)* timeout time, *(b)* the way flows are handled and *(c)* the controller performance; are affecting the overall performance of SDN. It was confirmed that even an one second increase (the minimum change allowed by OpenFlow) in the timeout time of a flow table rule can decrease both the delay as well as the number of packets arriving at the destination out of order. One of the limitations of the approach shown in Chapter 3 is that the algorithm parameters have to be manually provided by a network administration mechanism which can be a traffic management software or a network engineer. One of those parameters is the timeout time, which is statically specified and might not be enough in some scenarios for the algorithm to provide the optimal performance increase.

A very important piece of information that can be used by the proposed algorithm to overcome this limitation is the interarrival time between the packets of a flow. Using the interarrival time, the algorithm can dynamically assign the

appropriate timeout times as well as refine all the parameters for better QoS.

The sections of this chapter can be summarised as follows:

- Section 4.2 (Packet Interarrival Time in OpenFlow): Provides a thorough analysis on packet interarrival time and how it can be used in an OpenFlow network. In addition, some useful use cases of interarrival time in networking are provided.
- Section 4.3 (Packet Interarrival Time Based Enhancement Algorithm (OFPEX)): Provides all the details about the proposed algorithm that takes advantage of the packets interarrival time in order to achieve better performance in OpenFlow networks.
- Section 4.4 (Experiments and Analysis): Provides details about all the experiments performed in order to test the validity of the proposed algorithm as well as a thorough analysis of the results gained.
- Section 4.5 (Summary and Discussions): Provides the conclusion, summarising every important aspect of the chapter.

4.2 Packet Interarrival Time in OpenFlow

The interarrival time of packets in networking is the time between two successful packet arrivals at a destination. There are two ways to calculate the interarrival time, *(a)* either from the start of each packet, which essentially includes the transmission time or *(b)* start with the last bit of the preceding packet and end with the first bit of the next packet. In most of the cases, the second way is preferred due to the fact that it is unaffected by the transmission time. From the interarrival time statistics one can conclude on many traffic characteristics. For example a close mean, mode and median with low standard deviation indicates that the interarrival time is consistent and there is very low delay present. On the other hand if there is a diverge between mean, mode and median and the standard deviation is large then there is a lot of delay present.

When it comes to the performance of OpenFlow networks, an efficient way of managing flows leads to better QoS and therefore better end-to-end performance.

Therefore, except from packets interarrival time, a very important aspect is to know how flows are acting within a network. Some very good works have been published for data centres. In data centres, the interarrival time of flows depends on the data centre type. According to [129], in University data centres, 80% of the flows have interarrival times between 4ms and 40ms whereas in private data centres 80% of the flows have an interarrival time under 1ms. Furthermore, [130] suggests that 80% of the flows last for less than 10 seconds whereas less than 0.1% of the flows last more than 200s. In addition, 50% of the bytes (data) are in flows that last less than 25 seconds. Finally, the paper indicates that sometimes there are periodic short-term burst of flows but on average one can expect around 10^5 flows per second.

Inspecting flows and packets is a very difficult task in OpenFlow due to the fact that the information exchange between the switch and the controller is limited. As stated in Chapter 2, this information includes *(a)* Event-based Messages, *(b)* Packet-in Messages and *(c)* Flow Statistics with *(b)* and *(c)* being the most useful ones. Flow Statistics is aggregated flow information over a period of time provided upon controller request by the switch. This poses a limitation due to extra load on both the switch and the controller for every such request [115]. Furthermore, due to average flow durations [129, 130], most of the flows will have already expired by the time the statistics request is made and analysed by the controller. In addition, [131] concludes that 60% of the links in a data centre are actively used with significantly higher utilisation in the core links compared to edge links. In [132], a new information channel is proposed for per-flow sampling in order to overcome the flow statistics problem that is faced by OpenFlow.

In order to examine the importance of packet interarrival time in the performance of OpenFlow networks an experiment was conducted. In the experiment, the packet interarrival time started at 5ms and it was slowly decreased to 0.04ms. At the same time, the number of flow table rule installations as well as the number of packets arriving at the destination out-of-order was measured. Furthermore, both delay and packet loss have also been measured. The experiment was repeated 30 times and the average readings were calculated in order to minimise experimental error.

From the experiment results, the relation between the interarrival time and the performance of OpenFlow networks can be observed. First of all, the number of new flow table rule installations as well as the number of packets out of order is exponentially related to the interarrival time. As the interarrival time decreases, both of them as well as packets lost exponentially increase as shown in Figures 4.1 and 4.2. Exactly the same happens to delay as shown in Figure 4.3. This confirms that the interarrival time of packets affects several of the performance areas that the algorithm is trying to improve, but it also gives a very good indication that it can be used in favour of the proposed algorithm. By knowing or estimating the interarrival time of packets within a flow, the controller can act proactively in order to increase the overall network performance.

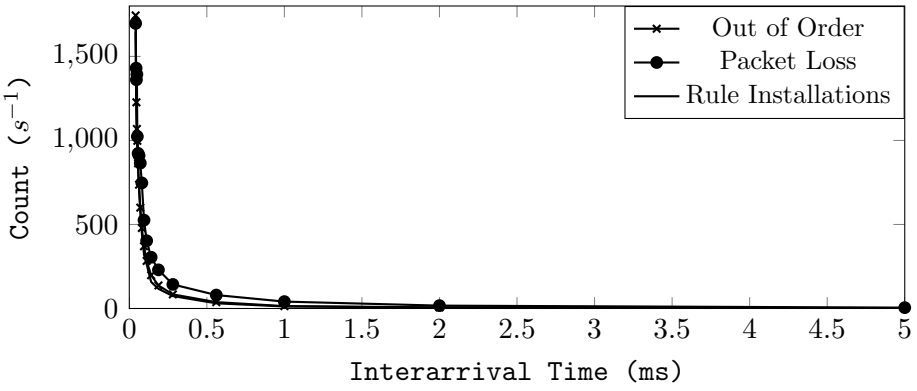


Figure 4.1: Number of Out-of-Order Packets, Rule Installations and Lost Packets Against Packet Interarrival Time

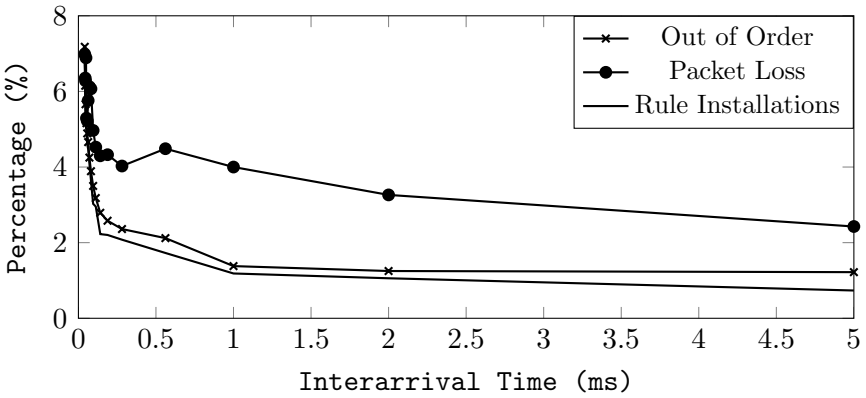


Figure 4.2: Percentage of Out-of-Order Packets, Rule Installations and Packet Loss Against Packet Interarrival Time

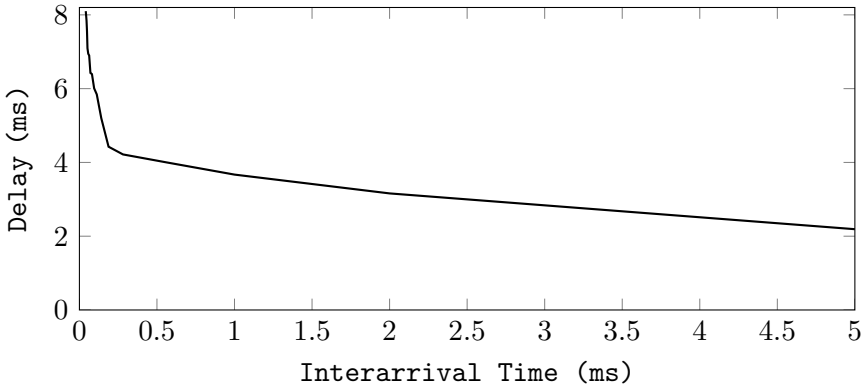


Figure 4.3: Delay Against Packet Interarrival Time

4.3 Packet Interarrival Time Based Enhancement Algorithm (OFPEX)

In order for the algorithm to successfully enhance the performance in OpenFlow networks it has to fulfil the following criteria:

1. Do not cause switch and controller overload
2. Collect and analyse valid flow statistics
3. Provide better QoS for priority flows in the form of decreased delay and packet losses as well as reduction of out of order packets

4.3.1 Statistics Gathering

In order to avoid switch and controller overload, the controller continuously monitors its own CPU usage and saves the data in order to be able to make statistical predictions for high load hours. The data is then used when the proposed algorithm collects statistics from the switch. It begins collecting statistics from the switch at predefined short time intervals starting at 1 second. Upon detection of increase in CPU utilisation (predefined threshold), the controller increases the time interval by 1 second and re-inspects the condition of both the switch and the controller. It then compares it with the previous reading and if it has an increasing tendency, it continues to increase the time interval. The maximum interval time is set to 25 seconds due to the fact that 50% of the data is in flows that last less than

25 seconds [130]. As soon as the controller observes a decrease in CPU utilisation it starts decreasing the time intervals by 1 second until it reaches the initial 1 second. An experiment was conducted in order to detect when the CPU utilisation causes decrease in network quality of service in the form of increased delay. From the experiment (Figure 4.4), after the 70% CPU utilisation, the readings showed a small increase in standard deviation. After 90% of CPU utilisation delay shows a significant increase. Therefore it was decided to set the predefined threshold at 90% of CPU utilisation.

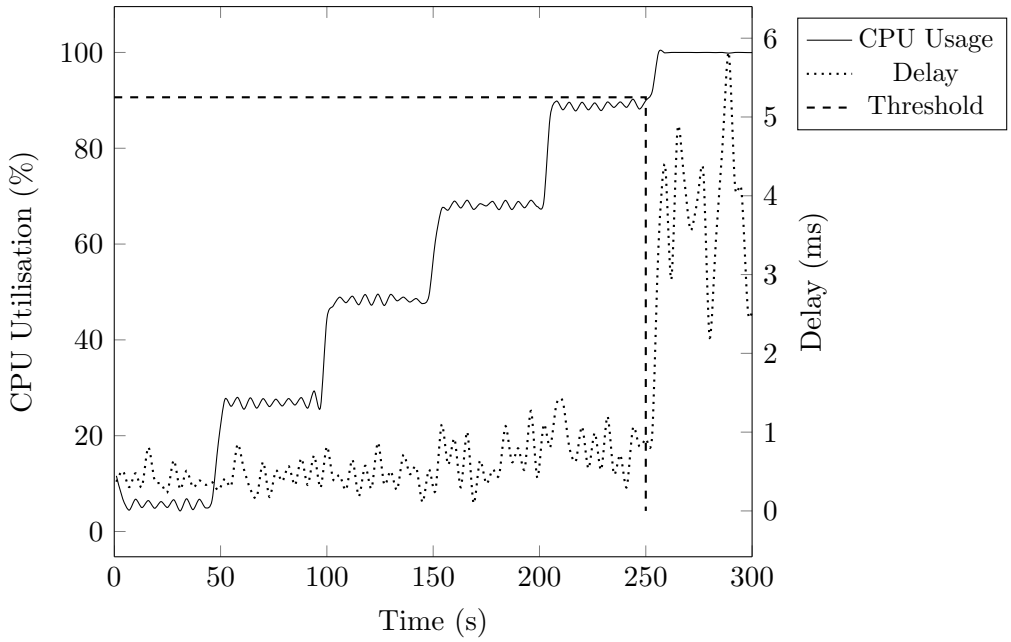


Figure 4.4: CPU Utilisation Threshold Experiment

Collecting flow statistics in OpenFlow networks is not an easy task. Packets arrive at the controller after a Packet-in Table-miss event and usually only the first 128 bytes of the packet are forwarded to the controller. These headers do not always contains useful information in order to help with flow analysis. If for example there is a TCP flow, then the first packet will be TCP SYN (three-way handshake), which contains no useful data in the payload to help with flow categorisation. Furthermore, in the case of UDP, having the switch send the full packet with its payload to the controller will impose significant load on the link between the switch and the controller as well as the controller. This is due to the fact that not only one packet from each flow reaches the controller as discussed in Chapter 3. This essentially gives an advantage in finding the packet interarrival

time without causing any extra load in the controller to switch link.

On the other hand, the statistics gathered by the switch namely Per Flow Entry Counters, do not offer a lot of information and in addition they are removed as soon as the flow expires. The most useful ones are the Per Port Counters which are not removed after a flow table entry expires. Most of the counter fields by default are optional in order to allow for less information collection and less network overload.

In the proposed algorithm, both the switch statistics as well as the Packet-in event packet headers are used for the statistical analysis. The information added to the database that the controller uses in order to manage the network includes:

1. Estimated flow duration

    - The duration of a flow is estimated using the initial packet that arrives from the Packet-in event and the timeout time x set for the flow table rule entry.
    - If after the expiration of the flow table rule no packets from the same flow arrive, then the flow is considered to have lasted less than or equal to the timeout time set ($\textit{flow duration} < x$). If a Packet-in event occurs then the new timeout time x is added to the initial.

2. Source and destination addresses and ports as well as the traffic type.

3. Timestamp

    - Using estimated flow packet interarrival time and timestamp, the algorithm can predict near future events such as switch or port overloading. Except from near future predictions, through time and data analysis from collected data, the controller can predict traffic patterns in extended periods of time. This allows the controller to act proactively and preserve a working condition for the switch, itself and the network by installing the appropriate rules.

4. Packet interarrival time

- If more than one packet of the same flow arrive at the controller (Chapter 3.2) then two subsequent packets are used to calculate the packet interarrival time.
- If only one packet arrives at the controller, then the packet interarrival time is estimated using the estimated flow duration and the flow statistics ("packet_count" Number of received packets) taken from the switch. This results in a rough estimate of the flow packet interarrival time.

The controller continuously updates the database by removing old flows and keeping newer flows in a "most common" order. This maintenance helps the controller in *(a)* faster search results *(b)* keeping the database small and manageable. All the calculated average data is kept in an independent database area and is updated with recent calculated data. In order to avoid chronological data affecting the controller judge during unpredicted events, the controller compares chronological average values consistency. If the old values are not consistent with the new values then the database entries for those values are updated in order to reflect the new changes. In addition, it monitors the flow table and the flow statistics in order to find rules that have exceeded the average flow duration. Upon finding such rules, the controller performs a flow table rule removal action.

In a multi-switch environment, the controller inspects the routes followed by the flows and compares them with the duration of the flow in order to find the best possible routes to redirect or guide the traffic through. At the same time using each switch condition it is able to find the cost for each route. Adding cost to routes, helps the controller decide the best possible route using two criteria, *(a)* Route Cost and *(b)* Flow Duration. Low duration flows are redirected to higher cost routes, whereas high duration flows are redirected to lower cost routes. This in a way acts in favour of the switch conditions because it helps overloaded switches by giving them flows that do not last enough to cause extra overload.

4.3.2 Use of Gathered Statistics

The proposed algorithm installs flow table rules by a decision mechanism (Figure 4.5) that uses gathered statistics. These decisions are grouped into the following steps:

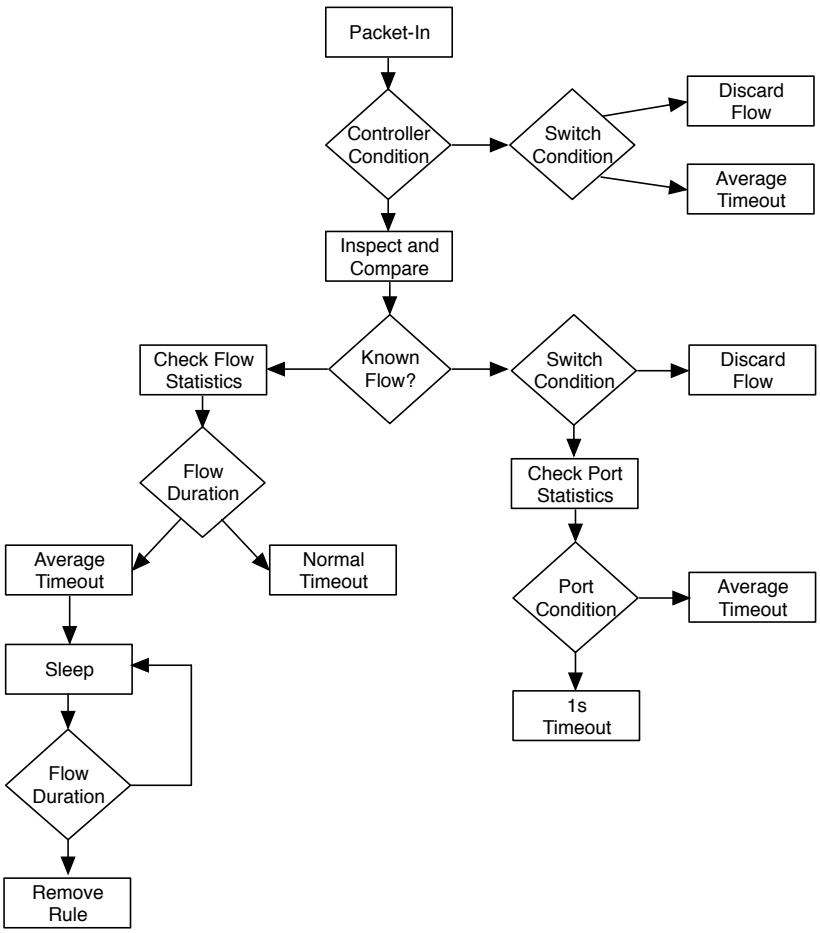


Figure 4.5: Controller Decision Mechanism Diagram

1. When a Packet-In message arrives at the controller, the controller checks its own condition. If it is overloaded then it checks the switch condition. If the switch is overloaded as well then the flow is discarded, if not then an average timeout time is given to the flow and a flow table rule is installed.

2. The controller inspects the packet headers and compares them to the flows database saved on the controller. If the flow already exists in the database then it proceeds to Step 3. If it does not exist then it proceeds to Step 4.

3. The controller goes through the flow statistics and compares its known duration with the average duration of all the flows of the relevant port. If the flow duration is above the average flow duration then it will create and install a flow table rule with hard timeout time equal to the average flow duration. If the flow duration is equal or less to the average flow duration then it will create and install a flow table rule with idle timeout time equal to the flow duration and proceed to Step 6.

4. The controller checks the switch condition, if the switch is overloaded then the flow is discarded, if it is not overloaded it proceeds to Step 5.

5. The controller goes through the port statistics and finds the average flow duration. It then inspects the port condition. If the port is close to its capacity then it will install a flow table rule with a hard timeout time equal to 1 second. If the port is not close to its capacity it will install a flow table rule with a hard timeout time equal to the average flow duration

6. Regularly inspect the flows with idle timeout flow table rules. If the duration of the flow exceeds the average flow duration then it will remove the flow table rule.

Knowing that on average a switch has to expect about 10^5 flows and that the number of flow table rule entries is limited as shown in Table 3.3 in Chapter 3, the risk of having a full flow table which means an unresponsive switch is very high. In order to overcome this problem, our proposed algorithm gives higher priority to low duration flows and installs their flow table rules using idle timeout time. This allows the controller to receive less Packet-in Messages, resulting in higher response rate. On the other hand it installs lower priority flows using hard timeout time in order to have more control on those longer time flows.

4.4 Experiments and Analysis

4.4.1 Scenario 1 - Static Interarrival Time

Scenario 1 was performed in order to examine the algorithm performance with a variety of different interarrival times. The topology (shown in Figure 4.6) consisted of three physical machines, two acting as hosts (sender and receiver) and one of them being a POX controller. An HP Procurve switch was used as the OpenFlow switch. The experiment was performed with five different interarrival times, 30 repetitions for each interarrival time, with each repetition lasting for 300 seconds. The increased amount of repetitions helped in the elimination of experimental errors.

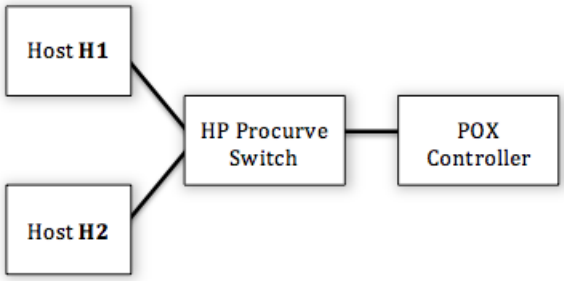


Figure 4.6: Scenario 1 - Topology

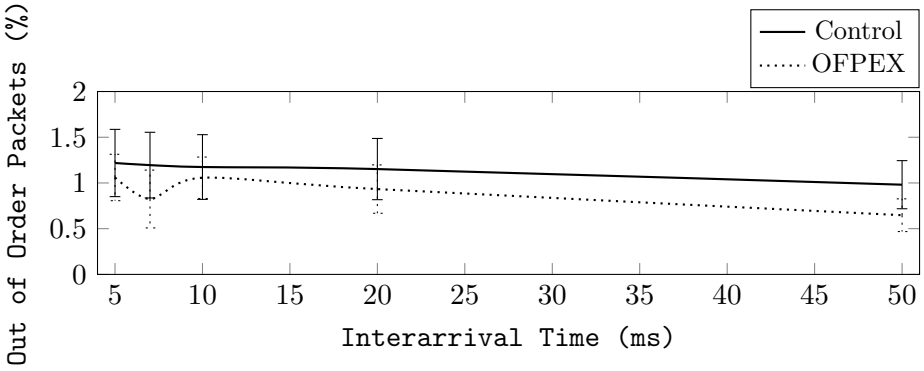


Figure 4.7: Scenario 1 - Interarrival Time vs Packets Out of Order Percentage

The proposed algorithm showed some very good results in scenario 1 experiments (summarised in Tables 4.1 and 4.2). More specifically, it has decreased the number of out-of-order packets by up to 33.67% which is a very important

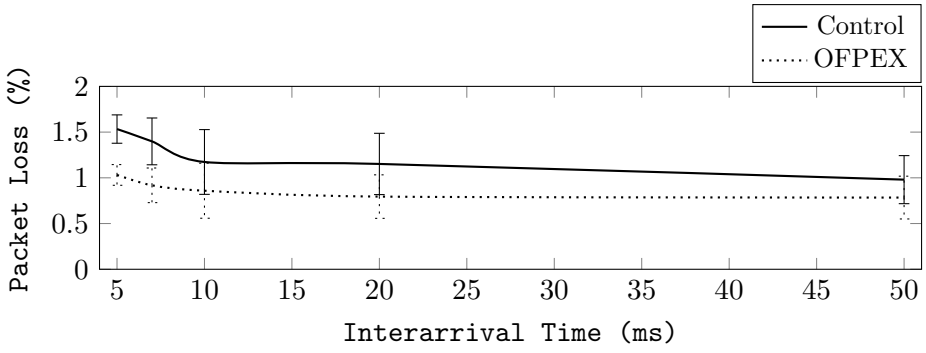


Figure 4.8: Scenario 1 - Interarrival Time vs Packet Loss Percentage

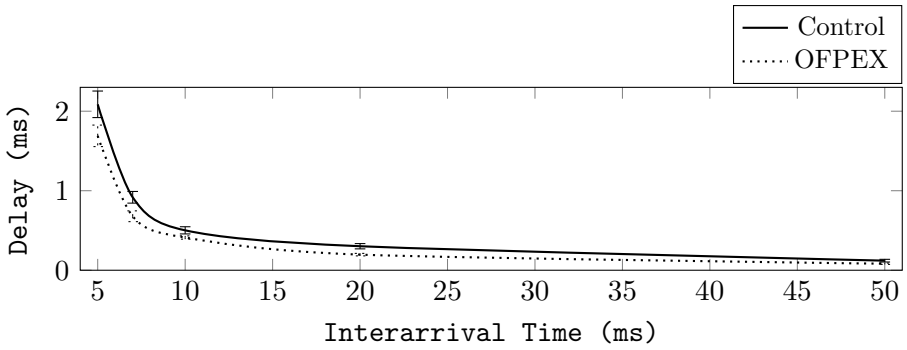


Figure 4.9: Scenario 1 - Interarrival Time vs Delay

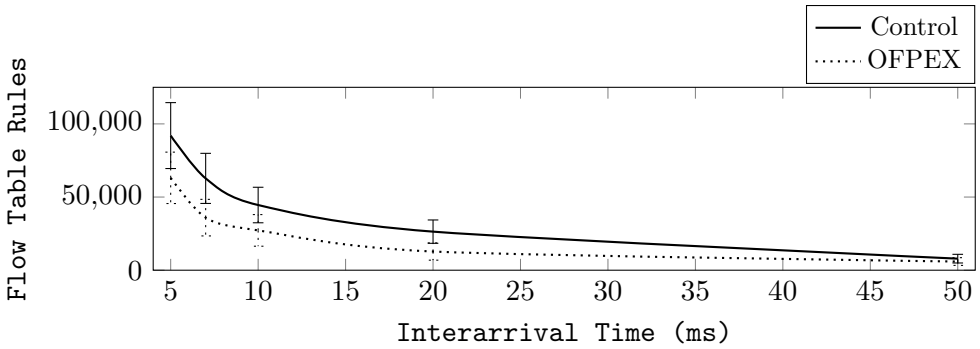


Figure 4.10: Scenario 1 - Interarrival Time vs Average Number of Flow Table Rules

	Control					OFPEX				
Interarrival Time (ms)	5	7	10	20	50	5	7	10	20	50
Out of Order Packets (%)	1.22	1.19	1.17	1.15	0.98	1.06	0.82	1.06	0.93	0.65
Packet Loss (%)	1.53	1.40	1.17	1.15	0.98	1.03	0.92	0.86	0.79	0.78
Delay (ms)	2.09	0.92	0.50	0.30	0.12	1.69	0.68	0.41	0.20	0.08
Flow Table Rules	92014	62757	44553	26359	7898	63132	35929	27202	12769	5923

Table 4.1: Scenario 1 - Readings Summary

	Diff. (%)				
Interarrival Time (ms)	5	7	10	20	50
Out of Order Packets (%)	13.11	31.09	9.40	19.13	33.67
Packet Loss (%)	32.68	34.29	26.50	31.30	20.41
Delay (ms)	19.14	26.09	18.00	33.33	33.33
Flow Table Rules	31.39	42.75	38.94	51.56	25.01

Table 4.2: Scenario 1 - Readings Percentage Change

performance improvement. There seems to be a sweet spot at 7ms interarrival time, at which point the algorithm performs really well (see Figure 4.7). This is most probably due to the refresh rate of the algorithm matching the interarrival time of packets, therefore is easier for the algorithm to have up to date statistics. Packet loss has also been decreased, reaching up to 34.29% decrease. As shown in Figure 4.8, the algorithm performed really well in this section, and the uncertainty in the readings is also less than the control experiment without the use of the algorithm.

Delay has also been decreased, with the decrease ranging from 18% up to 33.33% as shown in Figure 4.9. Finally, the number of flow table rules present at the switch has been decreased by up to 51.56% as shown in Figure 4.10

4.4.2 Scenario 2 - Dynamic Interarrival Time

Scenario 2 was performed in order to examine the algorithm performance with a variety of different interarrival times. The topology used (shown in Figure 4.6) was the same as in Scenario 1. The traffic used had a variable number of flows at each time instance, and each of those flows had variable interarrival times ranging from 0.5ms to 50ms. The experiment was repeated thirty times, with each repetition lasting 300 seconds. For each repetition, the exact same traffic with the same patterns was used.

Scenario 2 results were very impressive, due to the fact that the traffic used in this scenario is more close to a real life unpredictable traffic, with variable interarrival time as well as bursts of packets. The proposed algorithm results indicate that the algorithm can perform really well in such a scenario as shown in Table 4.3. As shown in Figure 4.11 it has decreased the number of out-of-order

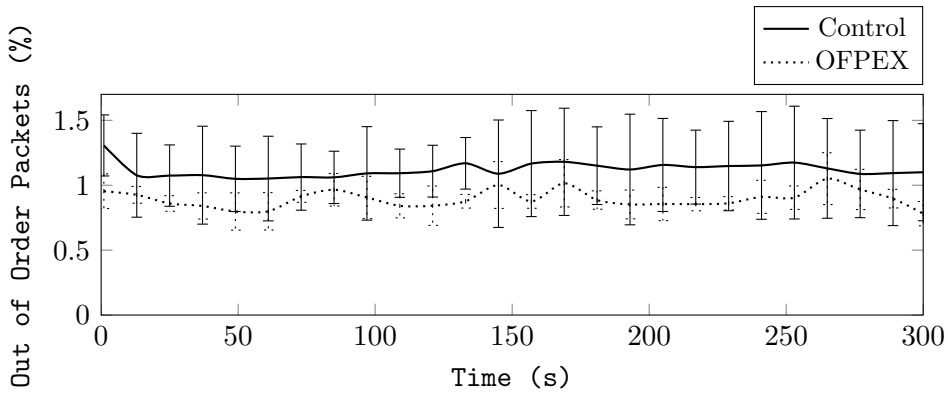


Figure 4.11: Scenario 2 - Packets Out of Order Percentage

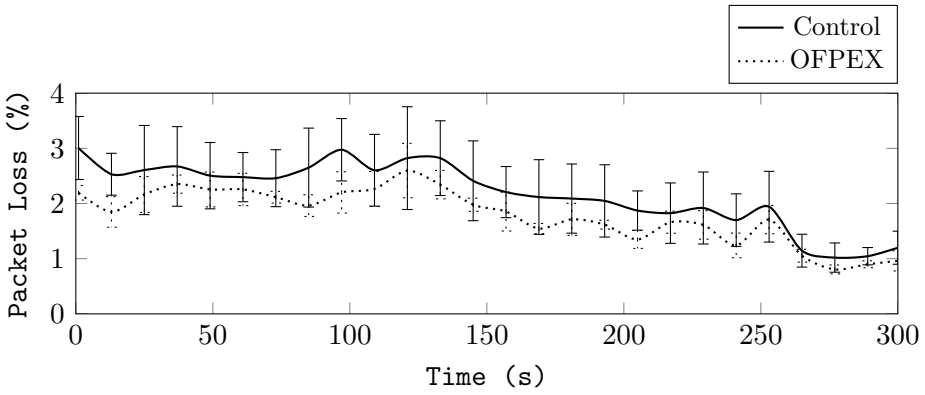


Figure 4.12: Scenario 2 - Packet Loss Percentage

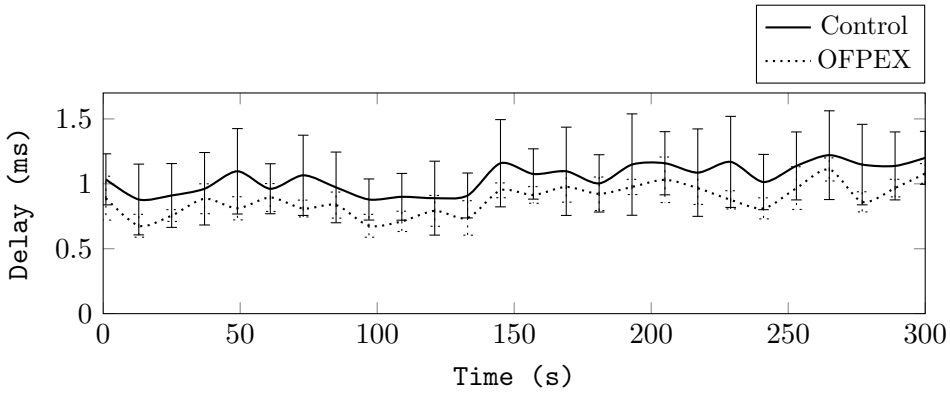


Figure 4.13: Scenario 2 - Delay

	Control	OFPEX	Diff. (%)
Out of Order Packets (%)	1.12 ± 0.32	0.89 ± 0.11	20.54
Packet Loss (%)	2.17 ± 0.55	1.79 ± 0.22	17.51
Delay (ms)	1.05 ± 0.27	0.88 ± 0.10	16.19
Flow Table Rules	70700 ± 17350	58808 ± 7090	16.82

Table 4.3: Scenario 2 Readings Summary

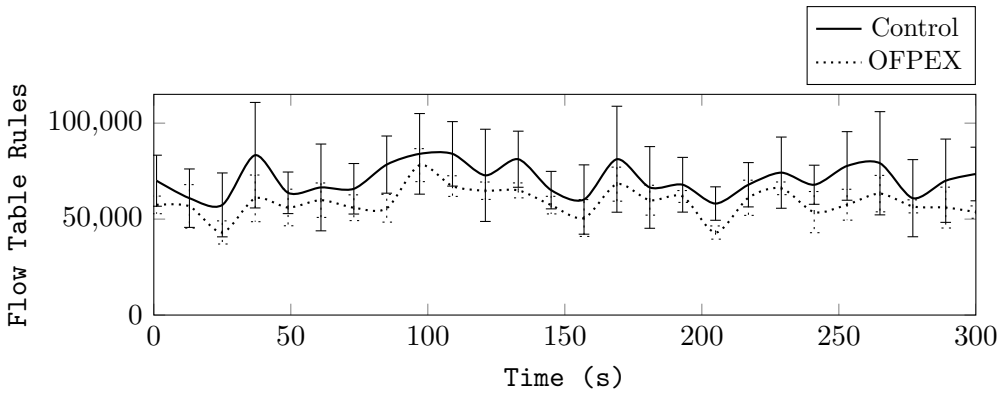


Figure 4.14: Scenario 2 - Flow Table Rules

packets by 20.54% as well as the standard deviation in the readings. In addition the packets lost have been decreased by 17.51% with a significant decrease in the standard deviation as well as shown in Figure 4.12. Finally, both delay as well as the number of flow table rules have been decreased 16.19% and 16.82% respectively as shown in Figures 4.13 and 4.14.

4.5 Summary and Discussions

In this chapter we introduced a performance enhancement algorithm for OpenFlow networks (OFPEX) that makes good use of flow statistics as well as the interarrival time of packets. The algorithm makes sure that it does not overload the controller and the statistics collected and analysed are valid. A threshold of 90% CPU utilisation was also set in place, to prevent the algorithm from affecting the network performance by consuming CPU resources when needed the most.

The proposed algorithm stores information such as an estimated flow duration which is calculated using the arrival time and the timeout time of the flow, source and destination addresses and ports as well as a timestamp for each flow. Finally, it stores the packet interarrival time which is calculated using two methods. Using these statistics, the algorithm can efficiently calculate better timeout times for flow table rules, as well as manage existing flow table rules.

Using experimental scenarios, we have found that the proposed algorithm performs better, compared to the same scenario without the use of the proposed algorithm. More specifically, in some cases in which the traffic used resembled

real-life traffic, the proposed algorithm was able to decrease the number of out-of-order packets by 20.54%, packet loss by 17.71%, delay by 16.19% and the number of flow tables rules present at the switch by 16.82%. The algorithm has also achieved some higher performance improvements but those came in traffic patterns that were controlled following the same interarrival time for the whole duration of the experiment.

Chapter 5

Distributed Mininet Placement Algorithm for Fat-Tree Topologies

5.1 Introduction

Distributed Mininet implementations have been extensively used in order to overcome Mininet's scalability issues. Even though they have achieved a high level of success, they still have problems and can face bottlenecks due to the insufficient placement techniques. This is mainly due to the fact that all of the distributed Mininet implementations are using placement algorithms such as round-robin, which have not been created with networking experimentation in mind. Such algorithms can cause bottlenecks in an experimental scenario due to reasons such as link capacity limitations, adding delays that should not be present or even distributing resource hungry parts of the experiment into not powerful machines.

This chapter presents a new placement algorithm for distributed Mininet emulations with optimisation for Fat-Tree topologies. The proposed algorithm overcomes possible bottlenecks that can appear in emulations due to uneven distribution of computing resources or physical links. To distribute the emulation experiment evenly, the proposed algorithm assigns weights to each available machine as well as the communication links depending on their capabilities. In addition, it performs a static code analysis in order to assign the appropriate weights to the emulated topology. Finally, using the weights of the distributed machines as well as the experimental topology, it places the experimental topology components

accordingly.

The remainder of this chapter is organised as follows:

Section 5.2 (Distributed Mininet Analysis): Introduces existing distributed Mininet implementations and provides an analysis of the placement algorithms used.

Section 5.3 (Proposed Placement Algorithm): Presents the proposed placement algorithm by listing the requirements as well as discussing the algorithm operations.

Section 5.4 (Experimental Scenarios): Describes the topologies as well as the tests that have been carried out in each experimental scenario.

Section 5.5 (Experimental Results Analysis): Presents a thorough analysis of the experimental results.

Section 5.6 (Summary and Discussions): Provides the conclusion, summarising every important aspect of the chapter.

5.2 Distributed Mininet Analysis

Mininet, uses Linux network namespaces in order to create lightweight virtual nodes allowing the users to execute real code as well as standard Linux network applications on each of the virtual nodes created. This results in high resource requirements, in the form of CPU and RAM, which causes scalability issues. To extend Mininet's scalability, several distributed solutions have been proposed and implemented. DOT [133] introduced several features that solve some of Mininet's issues but is not actively maintained and it does not support the latest Mininet implementations. The developers on Mininet have added Mininet Cluster [134] prototype which it tackles some of the issues, but the development stayed in the prototype phase, with no timeline for the final release. Mininet CE [135] acts as a control layer that combines instances of Mininet forming a larger cluster that can be used for large scale emulation. A slightly different approach has been taken by SDN Cloud DC [136] which introduces a number of new modules that enhance both Mininet as well as POX [116] in order to create an SDN based Data Centre that can enable large scale experimentation. The best solution to several Mininet

problems came in the MaxiNet [137] project. MaxiNet is an actively maintained project which introduces several new features missing from Mininet, such as the resource monitoring, Docker container support, as well as time dilation [138]. With time dilation, MaxiNet solves another resource related Mininet limitation.

The only drawback of MaxiNet is the fact that even though it can spread the virtual nodes evenly across virtual or physical machines (workers), it has no notion of the workers performance as well as the network performance. Therefore, a resource hungry part of the network can end up in a not resourceful worker with limited link capacity, thus becoming the bottleneck of the network. With the use of METIS [139] graph partitioning library, MaxiNet splits the emulated network into x equal weight partitions where x is the number of workers available. The weight of each partition comes from two factors. The emulated link bandwidth, as well as the number of links each emulated node has. To optimise the network, it uses minimal edge cut and avoids limited physical links. MaxiNet allows users to specify if they want a specific node to be placed on a specific worker but in a very large automatically created topology is very hard for the user to manually assign each node to a worker.

Mininet Cluster, on the other hand, comes with a variety placement algorithms such as *RandomPlacer* which places nodes randomly or *RoundRobinPlacer* which it just places nodes equally around all workers. Also, it has *SwitchBinPlacer* which places switches into evenly-sized bins around each worker and *HostSwitchBinPlacer* which places switches and hosts into evenly-sized bins and places them around to each worker.

All these placement algorithms used both in MaxiNet and Mininet Cluster have not specifically designed to examine the network components and the resources they need. Thus a lot of them end up having several cross-server links that force physical links to become bottlenecks in an experimentation scenario. Also, they do not examine the worker capabilities; thus they may place a high demanding bin into a limited resource worker, causing inaccuracies in the final results. Today's data centre networks follow Fat-Tree topologies [6, 140, 141] which in the case of the placement algorithms available in both MaxiNet and Mininet Cluster, a lot of cross-server links will be created, and the risk of bottlenecks will be increased.

5.3 Proposed Placement Algorithm

5.3.1 Requirements

Designing a placement algorithm for distributed OpenFlow emulations has to fulfil several requirements in order to avoid negative effects on the emulation results as shown in Section 5.2. In the proposed algorithm, the following requirements were taken into consideration:

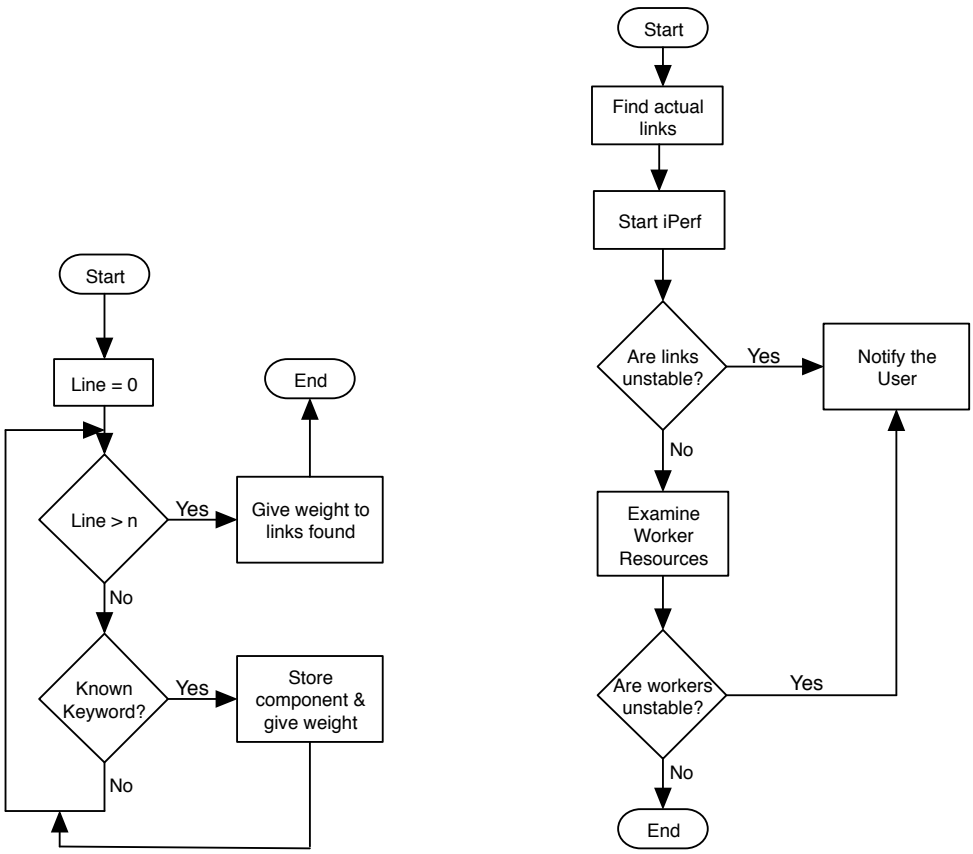
- The proposed algorithm should be aware of workers and links capabilities
- Worker resources should be close to emulation resources if possible
- The number of cross-server links must be kept as low as possible
- Notify the user in case of insufficient resources or possible problems with workers
- Fat-Tree topology optimisation

5.3.2 Overview

For the algorithm to achieve the requirements, a set of operations that perform individual tasks have been created. These operations consist of static code analysis, link capacity measuring, and bin creation.

5.3.2.1 Static Code Analysis

The static code analysis operation (as shown in Figure 5.1a) examines the emulation code written by the user to get a view of the experiment that is about to be implemented as well as the needs of each component. It examines known Mininet API "keywords" related to the topology components. Using the components found as well as their characteristics it creates a weights table and assigns them with relative weights. The weights are proportional to the traffic that each component has to handle, the number of nodes connected to each component (if the component is a switch) as well as the number of links connected to the component. Finally, it gives weight to each link using the maximum amount of traffic the link might face



(a) Module 1 - Static Code Analysis **(b)** Module 2 - Link Capacity

Figure 5.1: Proposed Placement Algorithm Operations

during the emulation. To find the volume of traffic, it searches for common traffic generation tools code such as iPerf [128] and Ostinato [142]. If the emulation uses traffic generation techniques that are not known to the module, then it uses the link capacity specified in code by the user as a weight indicator. If none of the two is provided, then using the number of nodes connected to the link it gives relative weights.

5.3.2.2 Link Capacity Measuring

The link capacity measuring operation measures the capacity of the links connecting the workers using iPerf, and assigns relative weights to each of those links. If the link is unstable, which can be caused by traffic travelling through the link prior to the experiment or if it faces a lot of packet drops during the iPerf test, then the user is notified. Such small details can affect Mininet experiments, that is why user has to get notified and either change the experimental platform or

agree to proceed with the existing one. In addition to link capacity, the operation examines the resources available to each worker. These resources include CPU power as well as RAM availability. In order for Mininet to run smoothly and not cause any effects on the final results, the CPU usage has to be in a "calm" state and the RAM has to be sufficient to accommodate Mininet's memory needs. The algorithm can spot possible limitations that the workers might have, and it will notify the user. In such a case is again up to the user to make alterations or agree to proceed.

5.3.2.3 Bin Creation

Bin creation operation creates bins of switches and hosts that match the individual workers weight, with a limited number of cross-server links. Due to the fact that a lot of production networks use the Fat-Tree or variations of the Fat-Tree topology, this part of the algorithm creates bins that are optimised for use in such topologies. Therefore, one of the goals that always tries to fulfil is to never split a pod apart if possible. The resulting bins are not meant to be of equal weight, but they should match the weight of each worker. If the algorithm suspects that the topology might face problems due to limited resources, it will notify the user before running the emulation. Having said that, the algorithm will work with any topology, but it has been tested and specifically optimised to handle Fat-Tree topologies with extra care when forming bins.

5.4 Experimental Scenarios

To examine the proposed algorithm, a Fat-Tree experimental topology (Figure 5.2) was used with variable number of Core Open vSwitches as well as Pods according to the scenario needs. All of the scenarios were repeated in three different experimental environments, with three, four and five physical machines, in the topology shown in Figure 5.3. In each of the scenarios, the resources available by the workers or the amount of traffic present in the experimental topology was changed. This allowed testing the functionality of some of the individual modules of the proposed placement algorithm. All of the scenarios were tested using

MaxiNet, first with MaxiNet's default placement algorithm (SwitchBinPlacer) as a control experiment and a basis to compare with, and then with the proposed algorithm. In order to eliminate experimental error, all of the experiments were repeated 30 times. Furthermore, throughout the duration of each experiment, the workers performance was monitored for non-experiment related issues. This is due to the fact that in emulations, non-experiment related processes can use resources which can affect the experimental results. For all the scenarios, 5400RPM SATA hard discs have been used.

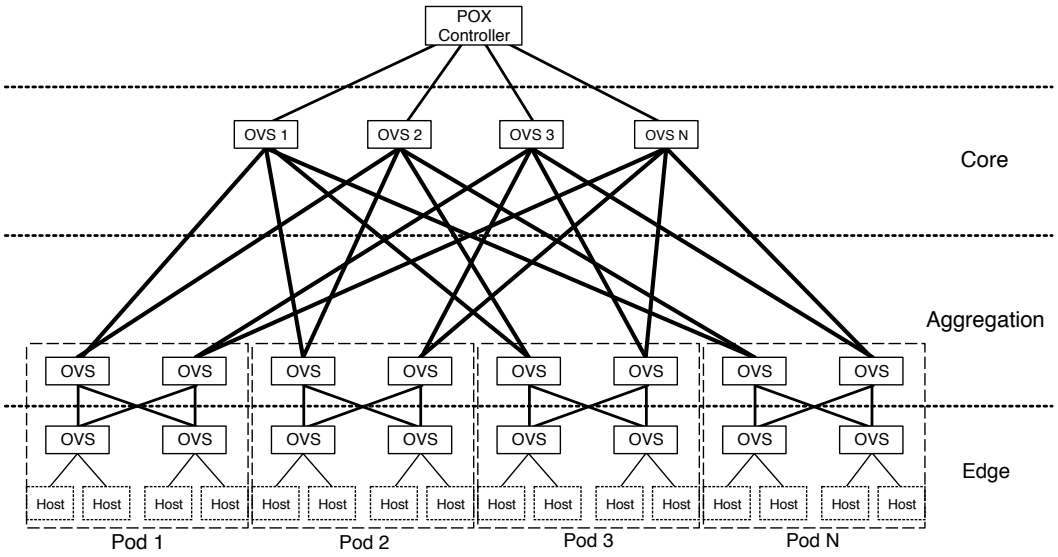


Figure 5.2: Experimental Topology

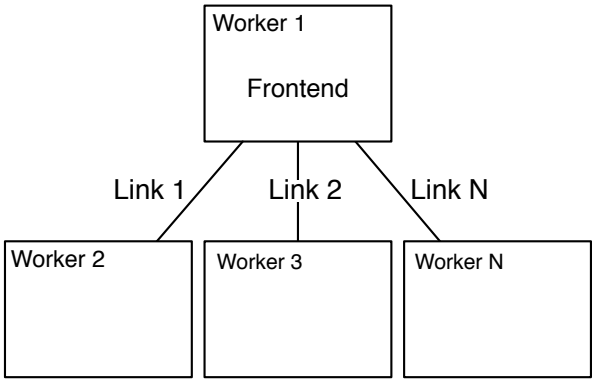


Figure 5.3: Workers Topology

Scenario 1A		Scenario 1B	
Component	**Characteristics**	**Component**	**Characteristics**
	Emulated Topology		
Pod 1	2Gbps intra-pod traffic	Pod 1	2Gbps intra-pod traffic
Pod 2	500Mbps to Pod 3	Pod 2	500Mbps to Pod 3
Pod 3	500Mbps to Pod 2	Pod 3	500Mbps to Pod 2
Pod 4	250Mbps to Pods 2 & 3	Pod 4	250Mbps to Pods 2 & 3
		Pod 5	500Mbps to Pod 3
	Workers Topology		
Link 1	1Gbps Capacity	Link 1	1Gbps Capacity
Link 2	600Mbps Capacity	Links 2, 3	600Mbps Capacity
Workers 1, 3	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD	Workers 1, 3, 4	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD
Worker 2	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD	Worker 2	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD

Scenario 1C	
Component	**Characteristics**
Emulated Topology	
Pod 1	2Gbps intra-pod traffic
Pod 2	500Mbps to Pod 3
Pod 3	500Mbps to Pod 2
Pod 4	250Mbps to Pods 2 & 3
Pod 5	2Gbps intra-pod traffic
Pod 6	250Mbps to Pods 4 & 5
Workers Topology	
Link 2	1Gbps Capacity
Links 1, 3, 4	600Mbps Capacity
Workers 1, 3, 4	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD
Workers 2, 5	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD

Table 5.1: Scenario 1 Characteristics

5.4.1 Scenario 1 - Weight Assignment

Scenario 1 purpose was to examine if the proposed algorithm gives the correct weights to both components and links, and assigns them to the appropriate physical workers. In order to achieve that, a variety of traffic bandwidths was used in the experimental topology as well as different resource capabilities in physical workers and links (summarised in Table 5.1).

In "Scenario 1A" the Fat-Tree topology consisted of four pods and four Open vSwitches. *Pod 1* had 2Gbps of network traffic travelling within the Pod (intra-pod). No traffic was travelling from *Pod 1* to any other pod. The rest of the pods had no intra-pod traffic, but *Pod 2* was exchanging 500Mbps traffic with *Pod 3* and *Pod 4* was sending 250Mbps traffic to *Pod 2* and 250Mbps to *Pod 3*. In the workers topology, physical *Link 1* had a 1Gbps capacity and *Link 2* only

600Mbps. Finally, *Workers 1* and *3* had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk whereas *Worker 2* had 16GB RAM, 8 cores at 2.4Ghz and 50Gb of hard disk.

"Scenario 1B" shared the same characteristics as scenario 1A, but this time the number of pods was five, with *Pod 5* sending 500Mbps of traffic to *Pod 3*. In addition, four workers were used, with *Worker 4* having 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk and *Link 3* that connected *Worker 4* with *Worker 1* having 600Mbps capacity.

Finally, "Scenario 1C" had six pods with *Pod 5* having 2Gbps of intra-pod traffic and *Pod 6* sending 250Mbps of traffic to *Pod 4* and *Pod 5*. The rest of the pods shared the same characteristics as Scenario 1A. The workers topology consisted of five workers and four links. Workers 1, 3 and 4 had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk whereas workers 2 and 5 had 16GB RAM, 8 cores at 2.4Ghz and 50Gb of hard disk. *Link 2* had 1Gbps capacity whereas links 1, 3 and 4 had 600Mbps capacity.

5.4.2 Scenario 2 - Component Assignment

Scenario 2 purpose was to examine if the proposed algorithm will assign the components of the emulated scenario accordingly in order to avoid having the physical links becoming the bottleneck of the experiment. That is why the characteristics of both the emulated and physical components (summarised in Table 5.2) were very closely chosen in order to cause a bottleneck if any component is misplaced. In "Scenario 2A", the experimental topology consisted of four pods. *Pod 1* was exchanging 2Gbps of traffic with *Pod 2* whereas. *Pod 3* was exchanging 1Gbps of traffic with *Pod 4*. Both *Pod 3* and *Pod 4* had 1Gbps of intra-pod traffic. Physical links *Link 1* and *Link 2* had a 1Gbps capacity. Finally, *Worker 1* had 16GB RAM, 8 cores at 2.4Ghz and 50Gb of hard disk, whereas *Workers 2 & 3* had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk.

In "Scenario 2B", the experimental topology consisted of five pods. Pods 1-4 shared the same characteristics as in Scenario 2A whereas *Pod 5* was sending 500Mbps of traffic to *Pod 1* and 500Mbps to *Pod 2*. Physical links *Link 1-Link 3* had a 1Gbps capacity. Finally, *Worker 1* had 16GB RAM, 8 cores at 2.4Ghz and

Scenario 2A		Scenario 2B	
Component	**Characteristics**	**Component**	**Characteristics**
	Emulated Topology		
Pod 1	2Gbps to Pod 2	Pod 1	2Gbps to Pod 2
Pod 2	2Gbps to Pod 1	Pod 2	2Gbps to Pod 1
Pod 3	1Gbps to Pod 4 & 1Gbps intra-pod traffic	Pod 3	1Gbps to Pod 4 & 1Gbps intra-pod traffic
Pod 4	1Gbps to Pod 3 & 1Gbps intra-pod traffic	Pod 4	1Gbps to Pod 3 & 1Gbps intra-pod traffic
		Pod 5	500Mbps to Pod 1 & 500Mbps to Pod 2
	Workers Topology		
Links 1, 2	1Gbps Capacity	Links 1, 2, 3	1Gbps Capacity
Worker 1	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD	Worker 1	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD
Workers 2, 3	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD	Workers 2, 3, 4	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD

Scenario 2C	
Component	**Characteristics**
Emulated Topology	
Pod 1	2Gbps to Pod 2
Pod 2	2Gbps to Pod 1
Pod 3	1Gbps to Pod 4 & 1Gbps intra-pod traffic
Pod 4	1Gbps to Pod 3 & 1Gbps intra-pod traffic
Pod 5	500Mbps to Pod 1 & 500Mbps to Pod 2
Pod 6	500Mbps to Pod 1 & 500Mbps to Pod 2
Workers Topology	
Links 1, 2, 3, 4	1Gbps Capacity
Worker 1	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD
Workers 2, 3, 4, 5	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD

Table 5.2: Scenario 2 Characteristics

50Gb of hard disk, whereas Workers 2-4 had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk.

Finally, In "Scenario 2C", six pods have been used. Pods 1-4 shared the same characteristics as in Scenario 2A whereas Pods 5-6 shared the same characteristics as *Pod 5* in Scenario 2A. Physical links *Link 1-Link 4* had a 1Gbps capacity. Finally, *Worker 1* had 16GB RAM, 8 cores at 2.4Ghz and 50Gb of hard disk, whereas Workers 2-5 had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk.

5.4.3 Scenario 3 - Increasing Topology Size

Scenario 3 purpose was to push the proposed algorithm to its limits by keeping a minimal physical workers topology and keep increasing the emulated Fat-Tree topology size. Using this approach, the performance as well as the different reactions of the algorithm can be observed (summarised in Table 5.3). In all of the scenarios, the workers topology consisted of three workers. *Worker 1* had 16GB

Scenario 3A		Scenario 3B	
Component	**Characteristics**	**Component**	**Characteristics**
		Emulated Topology	
Pod 1	1Gbps to Pod 2	Pod 1	1Gbps to Pod 2
Pod 2	1Gbps to Pod 1	Pod 2	1Gbps to Pod 1
Pod 3	1Gbps to Pod 4	Pod 3	1Gbps to Pod 4
Pod 4	1Gbps to Pod 3	Pod 4	1Gbps to Pod 3
		Pod 5	2Gbps intra-pod traffic
Scenario 3C		**All Scenarios**	
Component	**Characteristics**	**Component**	**Characteristics**
Emulated Topology		**Workers Topology**	
Pod 1	1Gbps to Pod 2	Links 1, 2	1Gbps Capacity
Pod 2	1Gbps to Pod 1	Worker 1	8 Cores at 2.4GHz, 16GB
Pod 3	1Gbps to Pod 4		RAM, 50GB HDD
Pod 4	1Gbps to Pod 3	Workers 2, 3	4 Cores at 2.4GHz, 8GB
Pod 5	1Gbps to Pod 6		RAM, 32GB HDD
Pod 6	1Gbps to Pod 5		

Table 5.3: Scenario 3 Characteristics

RAM, 8 cores at 2.4Ghz and 50Gb of hard disk, whereas *Workers 2 & 3* had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk. Both physical links *Link 1* and *Link 2* had a 1Gbps capacity. In "Scenario 3A" the experimental topology consisted of four pods. *Pod 1* was exchanging 1Gbps of traffic with *Pod 2* and *Pod 3* was exchanging 1Gbps of traffic with *Pod 4*. "Scenario 3B", the experimental topology consisted of five pods. Pods 1-4 shared the same characteristics as in Scenario 3A whereas *Pod 5* had 2Gbps of intra-pod traffic.

Finally, "Scenario 3C", had a six pod emulated topology. Pods 1-4 shared the same characteristics as in Scenario 3A whereas *Pod 5* was exchanging 1Gbps of traffic with *Pod 6*.

5.5 Experimental Results Analysis

The experimental scenarios yielded very important results which indicate the performance of the proposed algorithm compared to the default MaxiNet placement algorithm. The setup time in all the scenarios, when using the proposed algo-

rithm is increased. This is an expected result since the proposed algorithm does not immediately start placing components around randomly. It goes through the operations discussed in Section 5.3 in order to perform its weight calculations, then find the most appropriate workers and links and then start placing the emulated topology components to the relevant workers. Even though it performs the discussed calculations, the increase in the setup time is almost negligible compared to the benefits it brings to the important emulated topology performance improvements. The percentage increase in setup time ranges from 12.87% (Scenario 3A) to 48.74% (Scenario 2A), with the average being 31.49% and a standard deviation (σ) of 14.77.

Parameter	**MaxiNet Default Placement Algorithm**		
	1A	**1B**	**1C**
Packet Loss (%)	31.61	33.17	29.58
Delay (ms)	13.74	14.81	13.51
Setup Time (s)	35.30	36.13	39.52
Teardown Time (s)	12.56	13.21	15.45
CPU Usage (%)	86.67 σ=23.09	78.10 σ=25.44	87.25 σ=24.55
RAM Usage (%)	68.11 σ=24.43	65.74 σ=29.47	68.24 σ=28.87
Parameter	**Proposed Placement Algorithm**		
	1A	**1B**	**1C**
Packet Loss (%)	5.73	6.05	5.52
Delay (ms)	4.91	4.95	4.83
Setup Time (s)	47.42	50.09	51.84
Teardown Time (s)	13.42	13.90	15.84
CPU Usage (%)	88.33 σ=2.89	86.59 σ=3.06	86.18 σ=2.36
RAM Usage (%)	83.67 σ=7.51	83.28 σ=6.42	82.54 σ=7.11

Table 5.4: Scenario 1 Experimental Results

Similarly, the teardown time was increased as well. This is an unexpected result since in the proposed algorithm experiments, the teardown technique used is the MiniNet's default. The most probable cause for this increase is the fact that

MaxiNet stops hosts and switches in order. Since the proposed algorithm does not spread the topology in order, but by weight, that means, during teardown MaxiNet has to go around the workers several times until all the components are stopped. The percentage increase in teardown time ranges from 2.52% (Scenario 1C) to 19.39% (Scenario 2B), with the average being 12.60% and $\sigma = 6.11$.

Parameter	**MaxiNet Default Placement Algorithm**		
	2A	**2B**	**2C**
Packet Loss (%)	53.12	50.22	54.76
Delay (ms)	19.54	18.47	19.81
Setup Time (s)	37.05	38.21	40.08
Teardown Time (s)	14.54	14.65	15.31
CPU Usage (%)	88.33 σ=11.55	87.41 σ=10.41	88.10 σ=9.81
RAM Usage (%)	81.53 σ=13.86	81.72 σ=11.25	82.47 σ=12.47
Parameter	**Proposed Placement Algorithm**		
	2A	**2B**	**2C**
Packet Loss (%)	7.25	9.75	8.94
Delay (ms)	6.19	7.51	7.27
Setup Time (s)	55.11	56.32	57.11
Teardown Time (s)	17.34	17.49	18.23
CPU Usage (%)	91.42 σ=2.47	88.34 σ=3.17	89.37 σ=2.91
RAM Usage (%)	85.63 σ=1.53	84.26 σ=1.29	85.14 σ=2.57

Table 5.5: Scenario 2 Experimental Results

Except from setup and teardown times, the rest of the parameters favour the proposed algorithm. In Scenario 1, the proposed algorithm decreased the packet loss on average by 81.8% ($\sigma = 0.23$), whereas in Scenario 2 the average packet loss decrease reached 83.54% ($\sigma = 2.36$). Finally in Scenario 3 the average packet loss was 85.38% ($\sigma = 0.38$). Overall, the number of packets lost was decreased by 83.87% ($\sigma = 2.05$) with scenario 2A having the most significant decrease by 86.35% and 2B having the least significant decrease by 80.59%. This is a very good indication that the proposed algorithm has taken advantage of all the available

resources by allocating the emulated topology to the appropriate physical workers in order to utilise efficiently both the workers as well as the available links.

Parameter	**MaxiNet Default Placement Algorithm**		
	3A	**3B**	**3C**
Packet Loss (%)	48.35	50.17	65.06
Delay (ms)	17.14	18.61	21.56
Setup Time (s)	36.55	36.81	37.24
Teardown Time (s)	14.03	14.87	15.02
CPU Usage (%)	76.29 σ=10.27	82.34 σ=11.37	92.51 σ=12.68
RAM Usage (%)	74.58 σ=11.91	78.29 σ=14.85	84.63 σ=14.21
Parameter	**Proposed Placement Algorithm**		
	3A	**3B**	**3C**
Packet Loss (%)	7.03	7.59	9.24
Delay (ms)	6.11	6.27	7.36
Setup Time (s)	41.26	41.58	42.26
Teardown Time (s)	16.30	16.51	16.94
CPU Usage (%)	78.17 σ=2.69	81.65 σ=3.41	89.39 σ=3.26
RAM Usage (%)	74.92 σ=2.37	79.45 σ=2.86	88.31 σ=2.97

Table 5.6: Scenario 3 Experimental Results

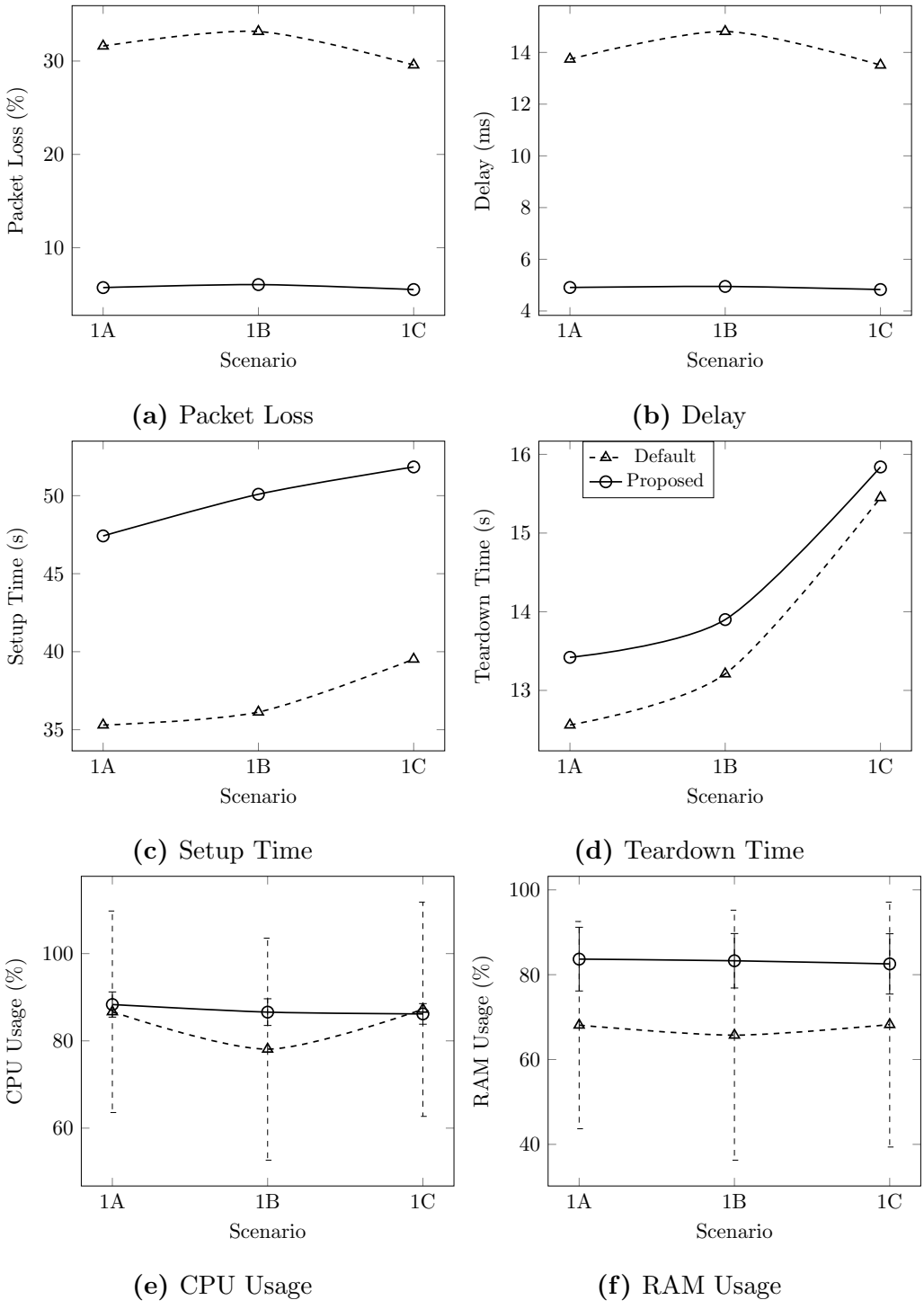
Delay was also decreased, on average by 64.76% ($\sigma = 2.39$) which is again a very good indication of optimal utilisation of available resources. In Scenario 1 the delay was decreased by 65.03% ($\sigma = 1.09$), in Scenario 2 by 63.65% ($\sigma = 3.68$) and in Scenario 3 by 65.51% ($\sigma = 0.84$).

The best indication of the work done by the proposed algorithm comes from both the CPU as well as the RAM usage readings. Even though the average value for CPU usage is very close for both algorithms, the standard deviation of the reading indicates that the proposed algorithm has utilised all of the workers CPUs almost equally. In Scenario 1B, MaxiNet's default algorithm reached a standard deviation of 25.44 in CPU usage which is very high compared to the 3.06 of the proposed algorithm. Since the CPU usage value consists of the CPUs of

all the workers' CPUs present in the experiment, a high value of standard deviation indicates that some of the CPUs are underutilised and some are overutilised. Therefore, the emulated components are not assigned the best possible way. In Scenario 1, the proposed algorithm increased the CPU usage average by 3.85% but it has managed to decrease the standard deviation by 88.63% (from 24.36 to 2.77). In Scenario 2, the proposed algorithm increased CPU usage by 2.00% but again it has decreased standard deviation by 73.09% (from 10.59 to 2.85). Finally in Scenario 3, the proposed algorithm decreased CPU usage by 0.58% as well as the standard deviation by 72.73% (from 11.44 to 3.12).

The same outcome happens with RAM where in some workers is underutilised or in some cases overutilised reaching up to 100%. Reaching 100% of RAM or CPU usage in emulation means that the workers do not have enough resources to run smoothly the emulated scenario, which will increase the number of packet losses or the delays in packet travel times. In some cases the operating system will start using the "Swap" memory which is much slower that RAM, leading in extra delays in the experimental results. This is true in all of the scenarios since both delay and packet loss percentages are significantly higher in the default MaxiNet placement algorithm compared to the proposed algorithm. Overall, in Scenario 1, the proposed algorithm increased RAM usage by 23.49% but decreased standard deviation by 74.59% (from 27.59 to 7.01), and in Scenario 2 it has increased RAM by 3.79% but decreased standard deviation by 85.63% (from 12.53 to 1.80). In Scenario 3 it has once again increased RAM by 2.10% and decreased standard deviation by 80% (from 13.66 to 2.73). All of the experimental results are summarised in Tables 5.4, 5.5 and 5.6.

On close inspection of network traffic, MaxiNet's default "SwitchBinPlacer" algorithm placed all of the scenarios the exact same way, ignoring the differences in traffic and link capacities. This is the reason why the emulated topologies did not perform so well with the default placement algorithm. In order for the default algorithm to reach the performance of the proposed algorithm, a lot more physical resources would be needed and even if provided it is not guaranteed that the emulated topology will get the resources needed since random placement can cause bottlenecks.

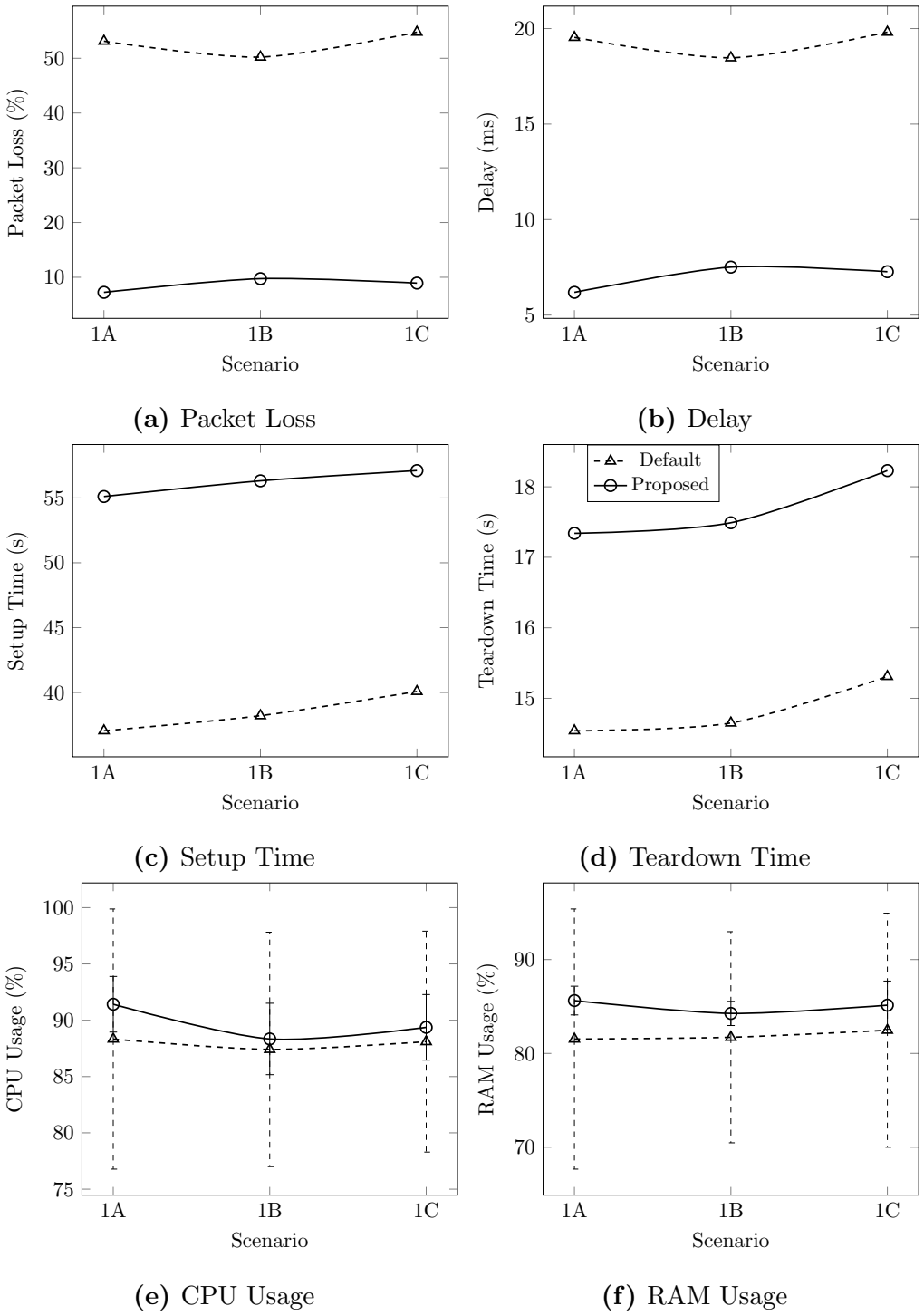


(a) Packet Loss **(b)** Delay

(c) Setup Time **(d)** Teardown Time

(e) CPU Usage **(f)** RAM Usage

Figure 5.4: Scenario 1 Readings



(a) Packet Loss

(b) Delay

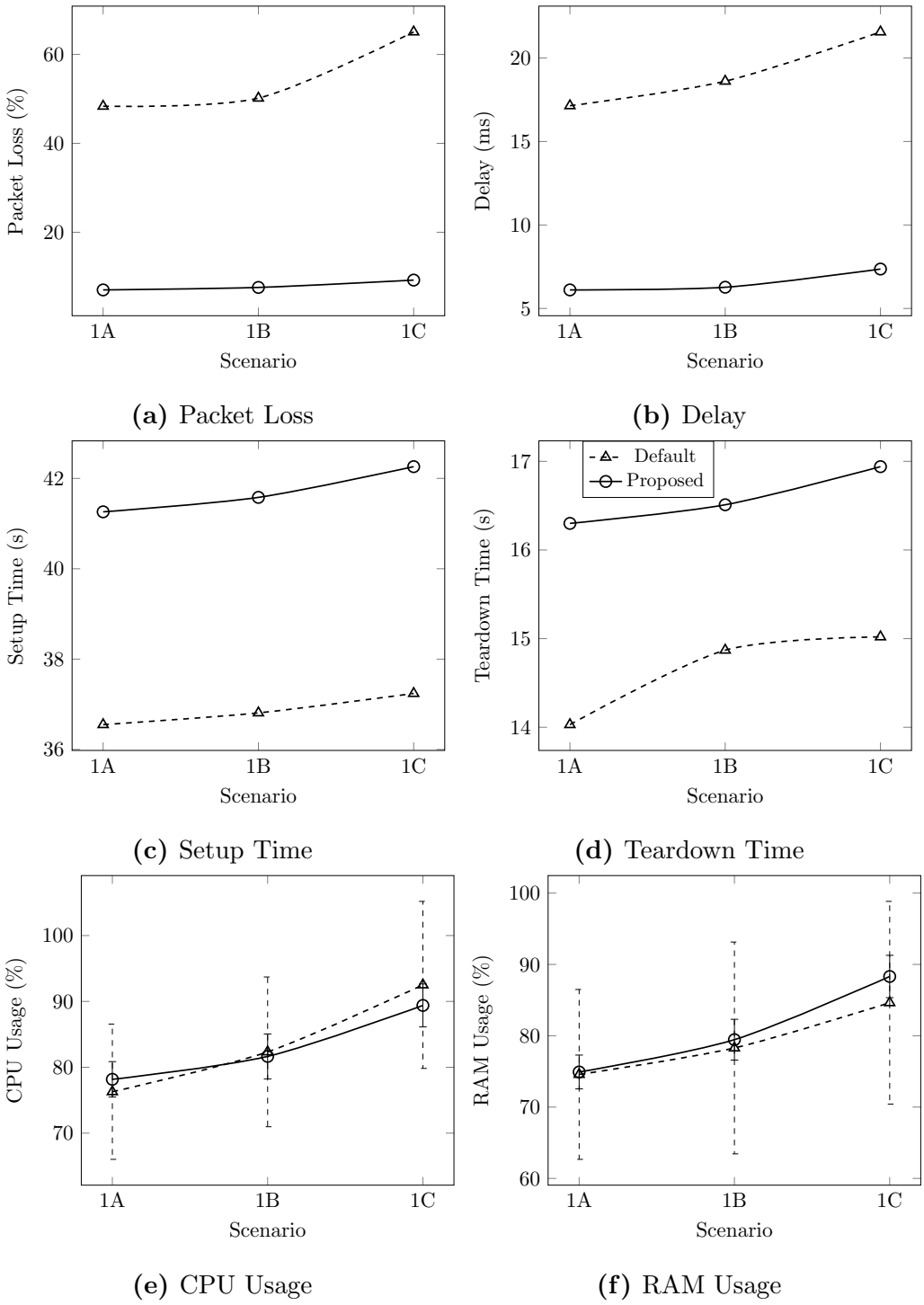
(c) Setup Time

(d) Teardown Time

(e) CPU Usage

(f) RAM Usage

Figure 5.5: Scenario 2 Readings



(a) Packet Loss **(b)** Delay

(c) Setup Time **(d)** Teardown Time

(e) CPU Usage **(f)** RAM Usage

Figure 5.6: Scenario 3 Readings

5.6 Summary and Discussions

In this chapter, we proposed a new placement algorithm for distributed Mininet network emulators. The proposed algorithm assigns weights to various components present in an emulated scenario such as hosts, switches, links as well as the traffic that will be present in the emulation. It then assigns weights to the available workers depending on their resources (CPU, RAM) as well as the links that connect them together. Finally, matching the emulated components weights with workers and links weights, it assigns each emulated component to the most appropriate worker. The algorithm is optimised for Fat-Tree topologies, in such a way that it does its best not to break apart pods especially if there is an indication of increased intra-pod traffic. The proposed algorithm compared to MaxiNet's default placement algorithm manages to decrease packet losses by up to 86.35% and delay by up to 65.51%. Also, it has managed to perform a better workers utilisation, indicated by the CPU usage standard deviation which was decreased by up to 88.63%.

Chapter 6

OpenFlow Performance Enhancement Algorithm In Large Topologies Using Distributed Mininet

6.1 Introduction

As shown in Chapters 5 and 7, Mininet is highly depended on CPU capabilities and it is very hard to create large topologies using just one typical High-End server. In order to overcome this problem, distributed Mininet implementations have been created by the networking community. There are several distributed Mininet implementations, with the most advanced being MaxiNet.

In Chapter 5 we discussed the fact that even though distributed Mininet implementations bring a lot of performance and scalability improvements, there are still problems with the way virtual components are allocated to the available infrastructure. Therefore we proposed a Placement Algorithm (Chapter 5) that is aware of the infrastructure capabilities and allocates virtual components accordingly.

Is this chapter, MaxiNet with the proposed Placement Algorithm (shown in Chapter 5) was used in order to examine the proposed OpenFlow performance enhancement algorithm OFPE shown in Chapter 3.

The remainder of this Chapter is organised as follows:

- Section 6.2 (Experimental Scenarios): Describes the setup, scenarios as well as the topologies used for performing the experiments on MaxiNet.
- Section 6.3 (Experimental Results Analysis): Provides a thorough analysis of the results obtained from the experimental scenarios.
- Section 6.4 (Summary and Discussions): Provides the conclusions together with a summary of the most important results.

6.2 Experimental Scenarios

In order to examine the performance of the proposed OFPE Algorithm (Chapter 3), we used the topology as well as parts of the scenarios used in Chapter 5. As shown in Figure 6.1, we used a Fat-Tree topology with variable number of Core Open vSwitches as well as Pods according to the scenario needs. For each scenario the appropriate experimental environment (Figure 6.2) was used, by using more workers for scenarios that demanded more resources. Each scenario with the exact characteristics was repeated with and without the use of the proposed algorithm in order to compare our results. Finally, the experiments were repeated 30 times and the average value of readings was calculated in order to minimise experimental error. In addition, throughout the duration of each experiment the workers performance was monitored for abnormalities caused by non-experiment related issues (Details of those issues are discussed in Chapter 7). As discussed in Chapters 5 and 7, due to their nature, emulation environment might be affected by processes that do not belong to the experiment but still use resources or cause delays to the performance of the Operating System and therefore affecting both MaxiNet and the controller.

6.2.1 Scenario 1 - Restrictions in Physical Topology

In the first scenario we have used the same characteristics as the ones used in Chapter 5 Scenario 1. In this case the purpose of the experiment was to examine the performance algorithm in various aspects. First of all it had to create

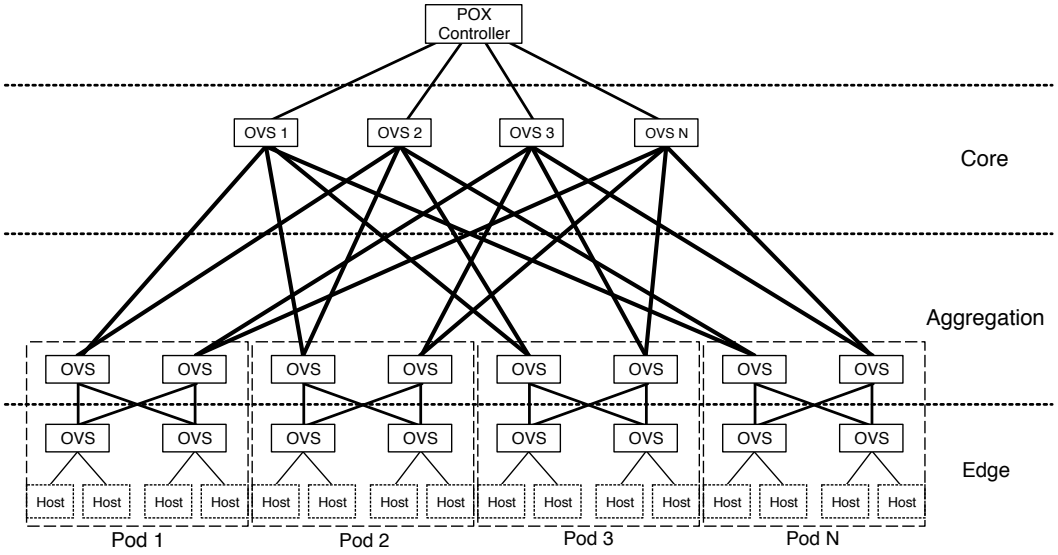


Figure 6.1: Experimental Topology

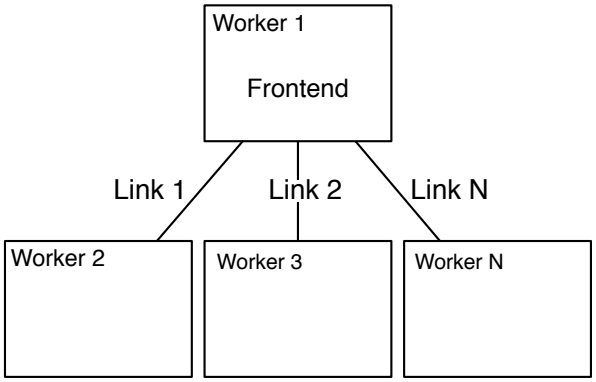


Figure 6.2: Workers Topology

the appropriate routes both inside pods as well as routes connecting the pods together. Then it had to handle various requests coming from all the switches and try to install the flow tables in the correct way so it will not affect the network performance.

As summarised in Table 6.1, the scenario had six pods with *Pod 1* and *Pod 5* having 2Gbps of network traffic travelling within the Pod (intra-pod) and no traffic was travelling from *Pod 1* and *Pod 5* to any other pod. *Pod 2* was exchanging 500Mbps traffic with *Pod 3*, and *Pod 4* was sending 250Mbps traffic to *Pod 2* and 250Mbps to *Pod 3*. Finally, *Pod 6* was sending 250Mbps of traffic to *Pod 4* and *Pod 5*.

The workers topology consisted of five workers and four links. Workers 1, 3 and

4 had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk whereas workers 2 and 5 had 16GB RAM, 8 cores at 2.4Ghz and 50Gb of hard disk. *Link 2* had 1Gbps capacity whereas links 1, 3 and 4 had 600Mbps capacity.

Component	Characteristics
Emulated Topology	
Pod 1	2Gbps intra-pod traffic
Pod 2	500Mbps to Pod 3
Pod 3	500Mbps to Pod 2
Pod 4	250Mbps to Pods 2 & 3
Pod 5	2Gbps intra-pod traffic
Pod 6	250Mbps to Pods 4 & 5
Workers Topology	
Link 2	1Gbps Capacity
Links 1, 3, 4	600Mbps Capacity
Workers 1, 3, 4	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD
Workers 2, 5	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD

Table 6.1: Scenario 1 Characteristics

6.2.2 Scenario 2 - Stressing The Controller

In the second scenario we have used the same characteristics as the ones used in Chapter 5 Scenario 2. In this case the purpose of the experiment was to stress the controller even more by increasing the amount of traffic. This will theoretically lead to higher response times as well as a decrease in performance areas such as delay and out-of-order packets.

As summarised in Table 6.2, the scenario had six pods with *Pod 3* and *Pod 4* having 1Gbps of network traffic travelling within the Pod (intra-pod). *Pod 1* was exchanging 2Gbps of traffic with *Pod2*, whereas *Pod 3* was exchanging 1Gbps of traffic with *Pod 4*. Finally, *Pod 5* and *Pod 6* were sending 500Mbps of traffic to *Pod 1* and 500Mbps of traffic to *Pod 2*

The workers topology consisted of five workers and four links, with all of the links having the same capacity at 1Gbps. Worker 1 had 16GB RAM, 8 cores at

2.4Ghz and 50Gb of hard disk, whereas workers 2-5 had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk.

Component	Characteristics
Emulated Topology	
Pod 1	2Gbps to Pod 2
Pod 2	2Gbps to Pod 1
Pod 3	1Gbps to Pod 4 & 1Gbps intra-pod traffic
Pod 4	1Gbps to Pod 3 & 1Gbps intra-pod traffic
Pod 5	500Mbps to Pod 1 & 500Mbps to Pod 2
Pod 6	500Mbps to Pod 1 & 500Mbps to Pod 2
Workers Topology	
Links 1, 2, 3, 4	1Gbps Capacity
Worker 1	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD
Workers 2, 3, 4, 5	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD

Table 6.2: Scenario 2 Characteristics

6.2.3 Scenario 3 - Stressing The Controller & The Workers

In the third scenario we have used the same characteristics as the ones used in Chapter 5 Scenario 3. The purpose of this experiment was to stress both the controller as well as all the links and the workers present in the topology.

As summarised in Table 6.3, the scenario had six pods with *Pod 1* exchanging 1Gbps of traffic with *Pod2*, *Pod 3* exchanging 1Gbps of traffic with *Pod 4* and *Pod 5* exchanging 1Gbps of traffic with *Pod6*.

The workers topology consisted of three workers and two links. The links had a 1Gbps capacity each, and worker 1 had 16GB RAM, 8 cores at 2.4Ghz and 50Gb of hard disk, whereas workers 2-3 had 8GB of RAM, 4 cores at 2.4GHz and 32GB of hard disk.

Component	Characteristics
Emulated Topology	
Pod 1	1Gbps to Pod 2
Pod 2	1Gbps to Pod 1
Pod 3	1Gbps to Pod 4
Pod 4	1Gbps to Pod 3
Pod 5	1Gbps to Pod 6
Pod 6	1Gbps to Pod 5
Workers Topology	
Links 1, 2	1Gbps Capacity
Worker 1	8 Cores at 2.4GHz, 16GB RAM, 50GB HDD
Workers 2, 3	4 Cores at 2.4GHz, 8GB RAM, 32GB HDD

Table 6.3: Scenario 3 Characteristics

6.3 Experimental Results Analysis

The experimental scenarios yielded some very important results which indicate the performance of the proposed algorithm compared to scenarios that make no use of the algorithm. In the first scenario the highest improvement came in the form of packet loss as shown in the results summarised in Table 6.4. In the control experiment packet loss was at 5.52%, whereas with the use of the algorithm it was lowered to 3.84%, yielding in a 30.43% change. Delay on the other hand changed by 16.98% going from 4.83ms to 4.01ms. Finally, packets out-of-order have been lowered by 26.22% from 2.67% to 1.97%. The decrease in both the amount of packets lost as well as the amount of out-of-order packets, indicates that the proposed algorithm allows the controller to act faster and serve more traffic, without multiple packets having to visit the controller. As a result, the controller is less busy and is able to handle the requests thus decrease the number of out-of-order packets. In addition, by being able to serve the requests it means that the switch's buffer does not get full very often, that is why the packet loss has been decreased.

Parameter	Control	OFPE	Change (%)
Packet Loss (%)	5.52	3.84	30.43
Delay (ms)	4.83	4.01	16.98
Packets Out-of-Order (%)	2.67	1.97	26.22
Total Packets Sent (10^8)	1.61	1.61	

Table 6.4: Scenario 1 Experimental Results

In the second scenario, the results obtained have been identical to the first scenario as summarised in Table 6.5. Packet loss has been decreased by 31.77% from 8.94% to 6.1% whereas delay has been decreased by 27.37% from 7.27ms to 5.28ms. Finally, the amount of out-of-order packets has been decreased by 35.92% from 4.51% to 2.89%. Having these indications is even more clear that the proposed algorithm is able to alleviate the pressure from the controller. This is due to the fact that this scenario had significantly more traffic than the first scenario but once again the results obtained, especially in the packet loss and out-of-order packets indicate that the controller is able to react and serve the traffic more efficiently.

Parameter	Control	OFPE	Change (%)
Packet Loss (%)	8.94	6.1	31.77
Delay (ms)	7.27	5.28	27.37
Packets Out of Order (%)	4.51	2.89	35.92
Total Packets Sent (10^8)	2.67	2.67	

Table 6.5: Scenario 2 Experimental Results

Finally, in the third scenario the proposed algorithm performed as expected by achieving higher performance that the control experiment as summarised in Table 6.6. Packet loss was decreased by 28.79% from 9.24% to 6.58% and delay by 30.30% from 7.36ms to 5.13ms. Finally, the amount of out-of-order packets has been highly decreased by 44.99% from 5.89% to 3.24%. These results verify the findings of both scenario 1 and scenario 2 and indicate that the algorithm can

perform very well in any situation and under high pressure from increased amount of traffic as well as in multi-switch environments.

Parameter	**Control**	**OFPE**	**Change (%)**
Packet Loss (%)	9.24	6.58	28.79
Delay (ms)	7.36	5.13	30.30
Packets Out of Order (%)	5.89	3.24	44.99
Total Packets Sent (10^8)	1.60	1.6	

Table 6.6: Scenario 3 Experimental Results

6.4 Summary and Discussions

In this Chapter, we have used the placement algorithm proposed in Chapter 5 to prepare a MaxiNet experimental environment in order to test the OpenFlow performance enhancement algorithm proposed in Chapter 3. With the use of MaxiNet we have been able to test the OpenFlow performance enhancement algorithm in a bigger and more realistic topology which yielded more realistic and useful data.

In order to measure its performance we have used various scenarios that put pressure on both the proposed algorithm as well as the controller which carries the operations of the proposed algorithm. The first scenario had a restricted physical topology in order to examine the algorithm performance in creating the appropriate routes. The second scenario put more stress on the controller in order to examine its reaction time and performance under stress. Finally, in the third scenario both the controller as well as the emulation environment were under stress again in order to examine how it responds and how well it handles the traffic. The third scenario examined the performance of both the performance enhancement algorithm (Chapter 3) as well as the placement algorithm (Chapter 5).

In the experimental results, the proposed algorithm showed consistent performance by achieving better network performance in all of the scenarios. That performance increase came in the form of a decrease in packet loss, delay as well as out-of-order packets. More specifically, in some of the scenarios it has managed to

decrease packet loss by up to 31.77%, decrease delay by up to 30.30% and decrease out-of-order packets by up to 44.99%.

Chapter 7

Performance Benchmarking of SDN Experimental Platforms

7.1 Introduction

In order to validate SDN related work, simulation, emulation or actual testbeds environments using the OpenFlow protocol are used. Some examples of simulators include fs-sdn [143] which promises fast, accurate simulations as well as scalability advantages over its competitors. Furthermore, there is NS-3 [144] which has OpenFlow support but it is restricted due to the fact that it does not utilise the "switch to controller" communication protocol; it creates an object that implements the controller behaviour. Finally, another well known simulator is EstiNet 9.0 [145], which except from a simulator it can act as an emulator. It is able to run real-world controllers, and claims to solve problems (only when simulation mode is used) faced due to system schedulers during experimentation.

The most popular and tested SDN experimental platform is Mininet [82] emulator, which is a prototyping system that supports OpenFlow. The advantage of emulators over simulators is the fact that an emulator is much closer to a real world implementation. In an emulator a real world controller is used with a proper "switch to controller" communication. Furthermore, usually what is tested in emulators can be directly implemented in a production network using the exact same programming code and parameters as in the emulation. That results in better testing as well as bug proofing. On the other hand, emulators are dependent on

the system scheduler thus it is expected to face some issues when it comes to high demand emulations, and therefore they are not as scalable as simulators. This is one of the problems that Distributed OpenFlow Testbed (DOT) [133] solves by distributing the emulated environment across several physical machines as shown in Chapter 5.

When it comes to testbeds, the experimental platform used consists of OpenFlow enabled switches and servers. There is a huge number of OpenFlow enabled switches as well as NetFPGA [36], an open source hardware and software platform for research and experimentation. There is also Pantou [146] project which uses a custom router firmware called OpenWrt [147] in order to allow low-end home routers to act as OpenFlow switches.

All these experimenting environments lack of a way of performance testing. It is very important for each research idea to be tested in an environment that is suitable to the experiment needs. This will provide results that can be as close to real-life implementation as possible. For example if the experimental scenario involves a huge topology then an emulator that lacks scalability cannot be used as the experimenting platform. Therefore, the selection of the appropriate experimental platform is very important and as a result a way of benchmarking and rating all the available experimental environments is vital.

This Chapter presents and explains a number of performance tests that can be performed on each of the available experimental platforms in order to evaluate their performance in several areas. As a result, using these tests a ratings table can be created in order to allow researchers to choose the one that suits their needs. Furthermore, this Chapter goes a step ahead to perform, present and analyse all the proposed tests on the Mininet platform.

The remainder of this Chapter is organised as follows:

- Section 7.2 (Proposed Performance Benchmarking Tests): Describes all of the proposed performance benchmarking tests.

- Section 7.3 (Experiments and Analysis): Describes the setup for performing the tests on Mininet, as well as each scenario individually. Finally, it provides a thorough analysis of the results obtained.

- Section 7.4 (Summary and Discussions): Provides the conclusion as well as a summary of the most important information of the chapter.

7.2 Proposed Performance Benchmarking Tests

In order to review, benchmark and rate the available experimental platforms, a series of different experiments in specific areas have to be performed. First of all, each of the topologies chosen have to isolate one or more bottlenecks of each platform. In addition, each topology has to be compared to results coming from a same shape topology with the only difference of being smaller or larger in number of components present. It is unfair to compare for example a Tree with a Fat-Tree topology.

In order to compare each topology and find the performance of each platform, here is the list of all the metrics that have been taken into account:

1. Setup Time
2. Teardown Time
3. CPU Usage
4. Scalability
5. CPU Cores Load Balancing (CCLB)
6. RAM Usage
7. Initial Ping Delay (IPD)
8. Average Ping Delay (APD)
9. No Response Failure Rate (NRFR)
10. Fair Share of Resources

Setup as well as teardown times are an important measure of *(a)* the efficiency of the platform in the use of the available resources and *(b)* the scalability. CPU usage is important in order to evaluate the amount of system resources needed for

each topology with a specific number of nodes. As a result, it is a pretty good indication of the scalability of the platform under test. In addition, with CPU metrics one can confirm if a higher profile system will result in more accurate experimental results or the system specifications are negligible. Furthermore, it allows spotting and eliminating inconsistent readings caused by a fully loaded CPU. Finally, the CPU usage tests consist of two readings, one is the Initial CPU Usage (I-CPU), which is the CPU usage by the platform after it has created the specified topology but before it has begun with the specified tests. The second reading is the CPU usage during the experiment (CPU-DE), meaning is the average CPU usage during the time that the specified tests are executed.

CPU Cores Load Balancing (CCLB) is a measure of *(a)* the scalability and *(b)* the ability of the platform under test to initialise all the available CPU cores equally (i.e. takes advantage of a multi-core CPU). CCLB is measured by calculating the standard deviation of the readings of the average of each core, therefore a value close to 0 indicates an excellent load balancing. Random-access Memory (RAM) will indicate *(a)* how resource hungry the platform is, *(b)* if it will benefit from systems with more RAM and *(c)* scalability.

Ping delay is an important indication on how close the platform is to real life implementation. There are two important types of Ping delays, the Initial Ping Delay (IPD) and the Average Ping Delay (APD). The IPD is highly affected by the time it takes for the controller to add a flow table rule to the OpenFlow switch. The APD is the average of all the ping delays excluding IPD. Using an SDN experimental platform without initialising a controller (i.e. eliminating the installation of rules in the network device flow table) is unrealistic and cannot be compared to real-life. On the other hand adding the huge IPD into the APD will result in a significant increase that will not reflect the entire test as well as real-life implementation with pro-active controllers. Thus, splitting the Ping delay into IPD and APD is the best solution.

No Response Failure Rate (NRFR) is the percentage of failed attempts for communication during Ping Delay test. This appears regularly in emulations due to the fact that each test has to be allocated some processing time by the Operating System Scheduler. The OS Scheduler is not designed specifically for

the experimental platform and as a result it does not allocate processing time efficiently from the platform's point of view, and therefore it fails to perform as in real world.

Finally, Fair Share of Resources (FSR) indicates platform's performance in resource allocation. In this Chapter FSR is represented by the Coefficient of Variation (CV) of the delay when all the hosts in the topology perform Ping command at the same time. CV is given by $C_v = {}^{\sigma}/_{\mu} * 100$, where σ is the standard deviation of delay and μ is the average delay. In some topologies is impossible to get FSR results, for example the linear 2 hosts topology will always have only 2 hosts running ping therefore FSR results cannot be obtained.

7.3 Experiments and Analysis

Due to the fact that Mininet is widely used in SDN/OpenFlow experimentation, all the tests described in this section have been performed using Mininet. Five different topologies have been used with variable number of switches or nodes. The number of switches and or nodes in each of the experiments is denoted by N, where N had the values of 1, 100, 500 and 1000. Each experiment lasted for 300 seconds and in order to minimise experimental error, each experiment was repeated 30 times. All of the experiments have been performed on two different systems (see Table 7.1), in order to evaluate Mininet's behaviour in limited resources at first in the Low-End machine and then in a system that has more resources in the High-End machine.

Mininet documentation suggests the use of pre-build virtual images of Mininet for simplicity and in order to avoid the need to add libraries to the system. Such an approach is not considered to be the best for benchmarking since the operating system will allocate only a fraction of its resources to Mininet's virtual environment. Therefore for these experiments, Mininet was installed from source into a fully functional Ubuntu 14.04.1 LTS Linux operating system. Through the native installation, Mininet is capable to use the full resources of the system.

In addition, due to the fact that Mininet supports several types of switches, all of the experiments were repeated using Open vSwitch [117] (OVS), Indigo Virtual

	Low-End Machine	High-End Machine
OS	Ubuntu 14.04.1 LTS	
Kernel Version	3.13.0-32	
CPU Vendor	Intel	
Architecture	x86-64	
Cores	4	8
CPU GHz per core	2.4	
Cache (MB)	4	
RAM (MB)	8192	16384
Hard Disk (GB)	32	50
Virtualisation	VT-x	
Hypervisor	QEMU KVM	

Table 7.1: Mininet Experimental Machines Specifications

Switch (IVS) [148] and Mininet's Reference Switch. The reasoning behind the use of different switches is to investigate the effect of the switches on the overall performance and also inspect how well Mininet can handle real world switches like OVS and IVS, instead of just the performance of the reference switch which is specifically optimised for Mininet. Finally, in all of the results, the experimental error is presented in the form of standard deviation.

After tests completion, an initial analysis clearly indicated *(a)* both Reference and OVS switches results were identical and *(b)* IVS faced a lot of problems and gave inconsistent results in topologies with high number of nodes. Due to the fact that the difference between Reference and OVS was tiny enough to be considered as experimental error and therefore negligible, and because IVS problems indicated that Mininet might not be optimised for IVS yet, only OVS results are presented, which is currently the leading industry and research virtual switch.

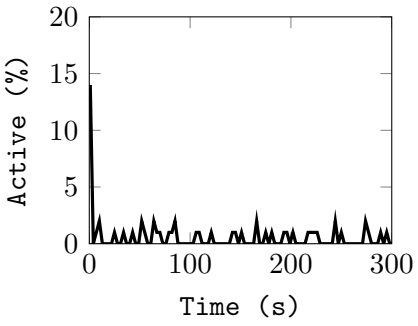
7.3.1 Default System Performance

Before running the actual experimental topologies, a default system benchmarking was performed in order to evaluate the default performance of the system and be able to find any problems and abnormalities in the experimental scenario readings.

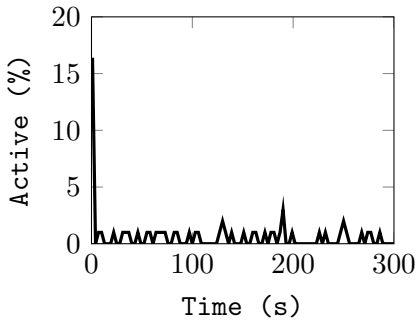
The processes running by the system were chosen to be exactly the same as the ones running in the rest of the scenarios. The only exceptions were Mininet processes since we where not running Mininet. Both wired and wireless networking was disabled to isolate the system, whereas throughout the experiments all the processes were monitored in order to make sure that they remained unchanged. In order to observe the behaviour of the system itself, we left the system running for 300 seconds and observed the CPU, RAM, Network and Disk usage. As with all the other scenarios, the experiment was repeated 30 times to minimise experimental error.

Parameter	**Low-End**	**High-End**
CPU Average Usage (%)	0.59	0.54
CPU Maximum Usage (%)	37.80	16.40
CPU Minimum Usage (%)	0	0
CPU Usage Standard Deviation	2.42	1.23
Network I/O Total Read (KB)	6.50	6.50
Network I/O Average Read (KB)	0.02	0.02
Network I/O Maximum Read (KB)	1.30	1.31
Network I/O Minimum Read (KB)	0	0
Network I/O Read Standard Deviation	0.17	0.17
Network I/O Total Write (KB)	6.50	6.51
Network I/O Average Write (KB)	0.02	0.02
Network I/O Maximum Write (KB)	1.30	1.29
Network I/O Minimum Write (KB)	0	0
Network I/O Write Standard Deviation	0.17	0.17
System Total RAM (MB)	8002.3	16104.8
Average Active RAM (MB)	180.60	180.13
Minimum Active RAM (MB)	170.22	170.57
Maximum Active RAM (MB)	190.04	191.01
Active RAM Standard Deviation	5.59	6.2
Average Disk Busy (%)	3.09	1.55
Maximum Disk Busy (%)	10.1	8.9
Minimum Disk Busy (%)	0	0
Disk Busy Standard Deviation	2.108	1.25

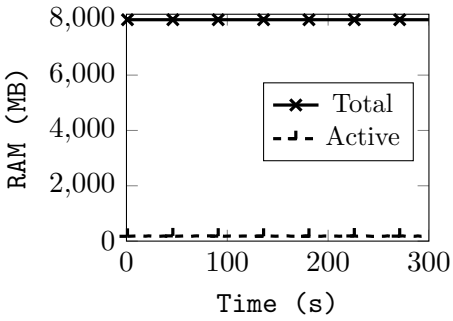
Table 7.2: Default System Performance



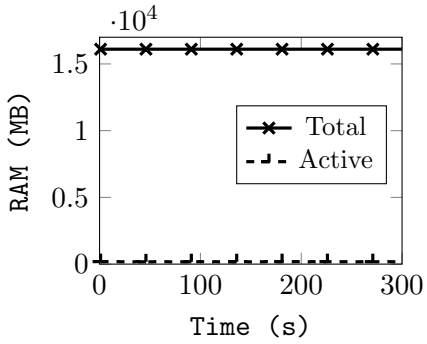
(a) Low-End CPU



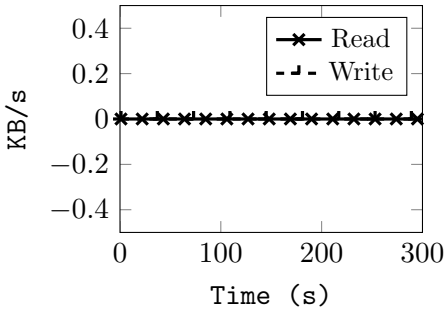
(b) High-End CPU



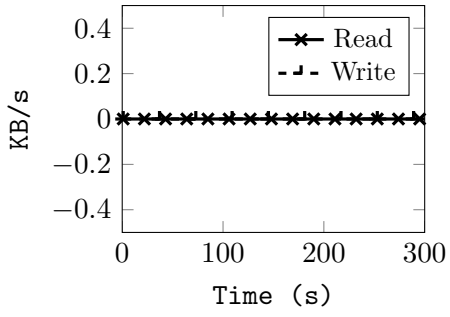
(c) Low-End RAM



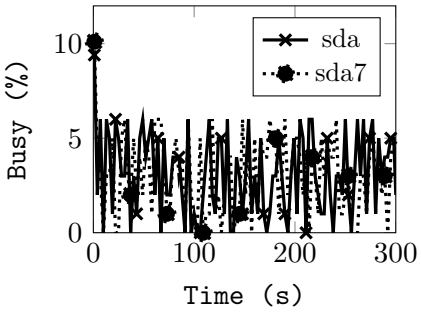
(d) High-End RAM



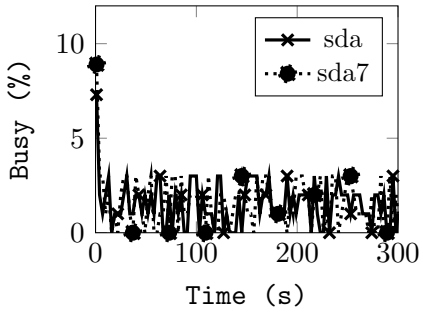
(e) Low-End Network I/O



(f) High-End Network I/O



(g) Low-End Disk Busy %



(h) High-End Disk Busy %

Figure 7.1: Default System Performance

As shown in Figures 7.1a and 7.1b, throughout the experiment both Low-End and High-End machines CPUs were using only a small fraction of their processing

power. These resources were used mainly by the operating system processes as well as the software used to monitor the system. The summary Table 7.2 shows that only 0.59% for the Low-End and 0.54% for the High-End of the processing power was used. Network I/O performances shown in Figures 7.1e and 7.1f, confirms that by disabling wired and wireless networking it will prevent the system from accessing the outside word and use processing power for purposes not related to our experimentation. RAM usage (Figures 7.1c and 7.1d) is almost unchanged for the duration of the experiment. Table 7.2 shows that the maximum active RAM is at about 190MB whereas the minimum active is at about 170MB. Finally, Figures 7.1g and 7.1h show that the hard drive has some activity in both reading and writing; but this can be taken to be *a.* the software that takes system measurements since it outputs the measurements in a text document in real-time and *b.* the swap memory used by Ubuntu Linux. Even though the disk busy percentage has some peaks, these peaks are only for *sda* and *sda7*, therefore the average disk busy percentages is only 3.09% for the Low-End and 1.55% for the High-End Machine (Table 7.2).

7.3.2 Scenario 1 - Dumbbell-Shaped Topology

The first topology namely "Dumbbell-Shaped" topology as shown in Figure 7.2, examines Mininet's performance with a bottleneck link, the link between Switch 1 and Switch 2. For Ping and Bandwidth (using iPerf [128]) tests, $Host_{1N}$ was communicating with $Host_{2N}$ resulting in a one-to-one connection with the appropriate switch for each host.

In Dumbbell-Shaped topology, setup and teardown times are not affected by the system resources since both Low and High-End systems results are identical, but both of them increase linearly as the number of hosts in the system increases. Furthermore, in both cases the teardown time is always higher than the setup time ranging from 3 up to 7 times higher as shown in Figure 7.3. I-CPU snows an anomaly, at N=1 and N=100 is higher in High-End system even though it should normally be higher in the Low-End system for any number of N. CPU-DE is always higher in the High-End system even though there are more resources available. In the Low-End system at N=1 CCLB is not very efficient, but once the

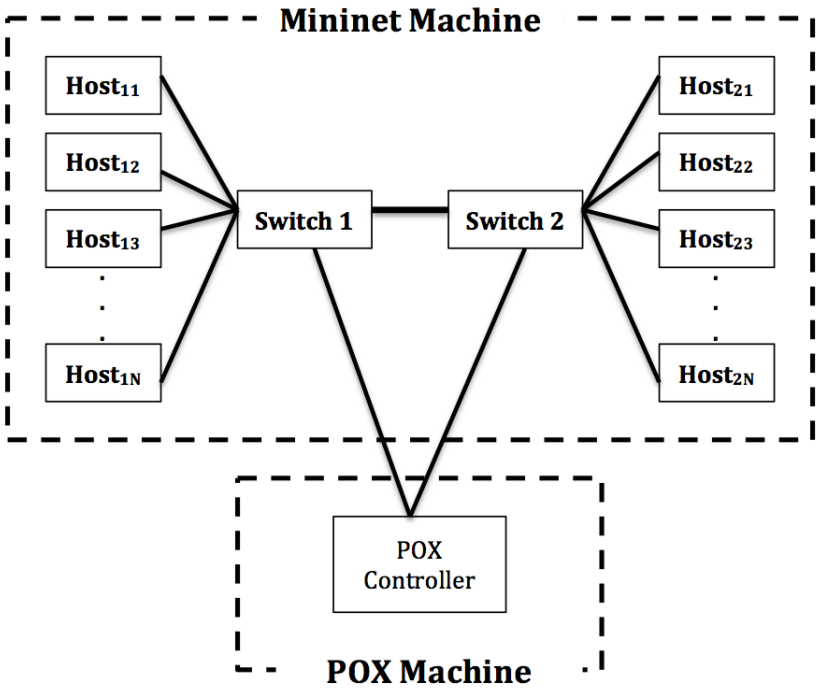
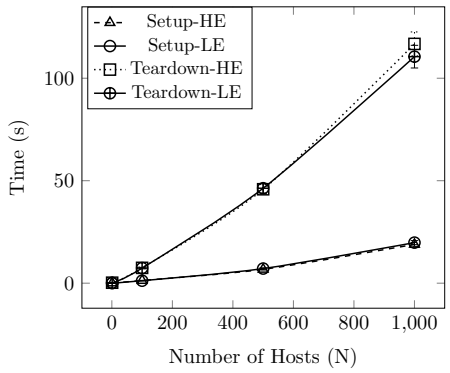


Figure 7.2: Scenario 1 - Dumbbell-Shaped Topology

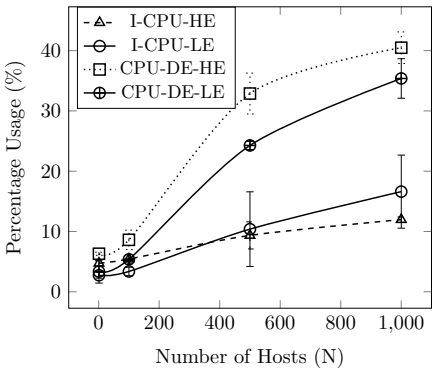
number of hosts increases it becomes more efficient. In High-End server slightly more RAM was used which in combination with CPU readings means that Mininet took an advantage of the availability of resources. In addition, even though APD is identical for both systems at any number of hosts N, when it comes to N=500 and N=1000 the IPD is lower in the High-End system. Finally, NRFR shows failed responses only at N=1000, with the Low-End system reaching 86.4% whereas High-End system is at 42.9%. All of the results for the Dumbbell-Shaped topology are summarised in Table 7.3.

Low-End System				
	Number of Hosts (N)			
	1	100	500	1000
Setup (s)	0.045 ± 0.002	1.21 ± 0.06	7.13 ± 0.36	19.79 ± 0.99
Teardown (s)	0.156 ± 0.008	7.37 ± 0.368	46.33 ± 2.32	110.53 ± 5.527
I-CPU (%)	2.74 ± 0.48	3.37 ± 0.699	10.39 ± 6.19	16.61 ± 6.06
CPU-DE (%)	3.35 ± 1.91	5.38 ± 0.19	24.27 ± 0.67	35.38 ± 3.28
CCLB	2.21	0.22	0.78	3.79
RAM (MB)	253 ± 4.39	478 ± 5.51	1394 ± 100.82	2549 ± 189.68
IPD (ms)	8.76 ± 0.49	12.6 ± 1.01	83.8 ± 8.02	93.1 ± 5.19
APD (ms)	0.06 ± 0.13	0.07 ± 0.14	0.09 ± 0.13	0.099 ± 0.17
NRFR (%)	0.0	0.0	0.0	86.35
FSR (%)	62.30	279.02	517.21	47.30
High-End System				
	Number of Hosts (N)			
	1	100	500	1000
Setup (s)	0.077 ± 0.004	1.309 ± 0.065	6.76 ± 0.34	18.93 ± 0.95
Teardown (s)	0.235 ± 0.0118	7.48 ± 0.374	45.83 ± 2.291	116.75 ± 5.84
I-CPU (%)	4.74 ± 0.41	5.43 ± 0.70	9.35 ± 2.25	11.97 ± 0.44
CPU-DE (%)	6.31 ± 0.26	8.62 ± 1.60	32.88 ± 3.39	40.48 ± 2.62
CCLB	0.28	1.71	3.62	2.80
RAM (MB)	316 ± 8.54	559 ± 50.90	1542 ± 141.6	2847 ± 167.9
IPD (ms)	5.55 ± 0.35	16 ± 1.36	57.8 ± 5.44	66.8 ± 3.88
APD (ms)	0.06 ± 0.17	0.07 ± 0.18	0.08 ± 0.12	0.08 ± 0.11
NRFR (%)	0.0	0.0	0.0	42.94
FSR (%)	124.00	328.05	513.85	530.07

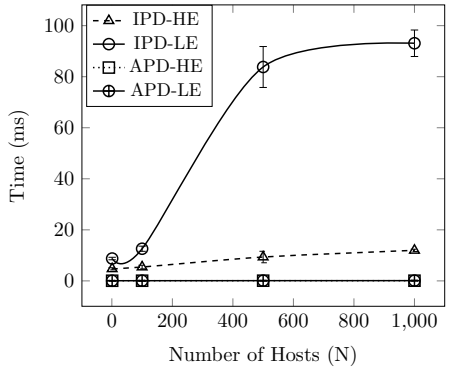
Table 7.3: Dumbbell-Shaped Topology OVS Summary



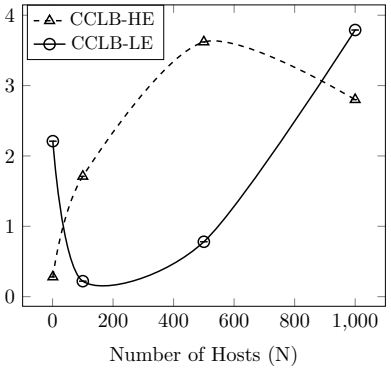
(a) Setup & Teardown Times



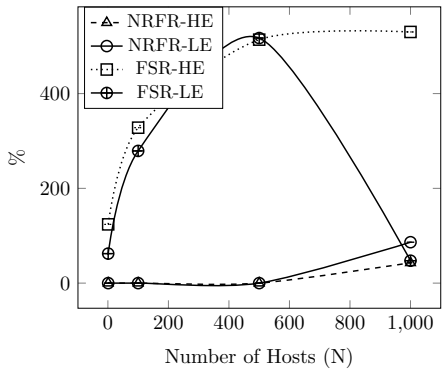
(b) I-CPU & CPU-DE Perc. Usage



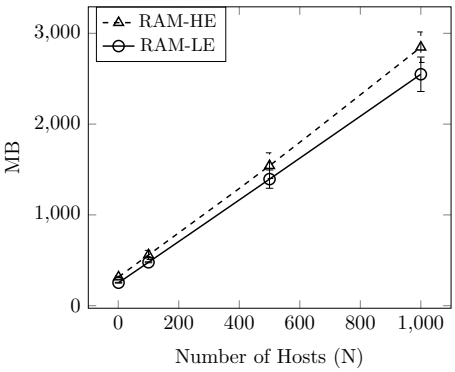
(c) IPD & APD Times



(d) CCLB



(e) NRFR & FSR



(f) RAM

Figure 7.3: Dumbbell-Shaped Topology Results

7.3.3 Scenario 2 - One-to-Many Topology

The second topology namely "One-to-Many" topology as shown in Figure 7.4, similar to the first topology it has a bottleneck link but this time it forms a one-to-many connection in which $Host_1$ pings a number of other nodes (i.e. from $Host_2$ up to $Host_N$).

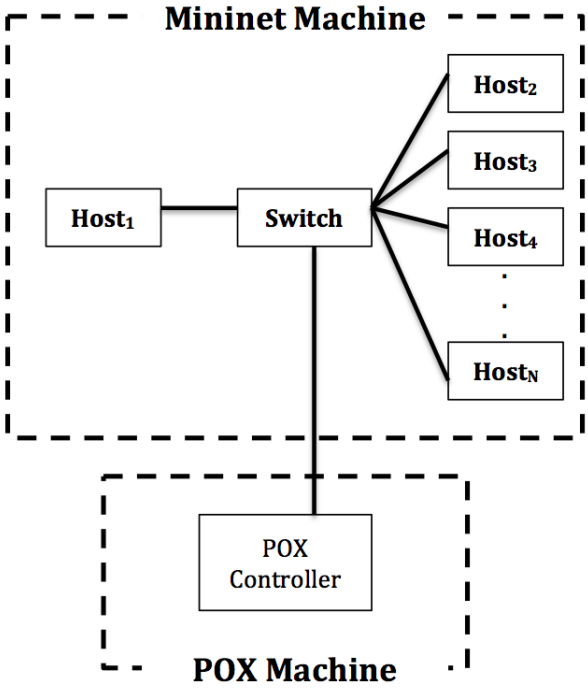


Figure 7.4: Scenario 2 - One-to-Many Topology

In One-to-Many topology both the setup and teardown times were significantly lower than Dumbbell-Shaped topology but once again the teardown time was significantly higher than the setup time. I-CPU usage was low for both Low and High end systems, but at N=500 and N=1000 the High-End system uses half the CPU Low-End system uses as shown in Figure 7.5.

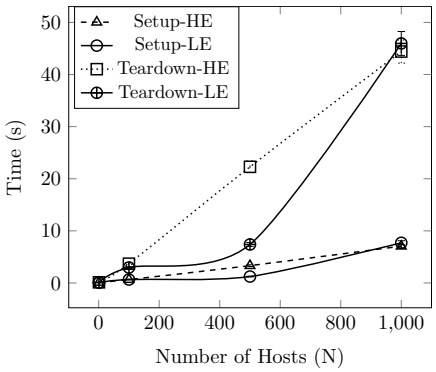
CCLB was identical to Dumbbell-Shaped topology, indicating that at high number of hosts it becomes more efficient. RAM usage was less for One-to-Many topology compared to Dumbbell-Shaped topology but again in the High-End system more of the available RAM was used compared to the Low-End system. Furthermore, even though the High-End system performed better in APD, in IPD the Low-End system performed much better especially in scenarios with higher number of switches (at 500 hosts Low-End IPD was 61% lower than High-End

and at 1000 hosts 51% lower). In addition, NRFR is identical for both systems and it is also much lower than in Dumbbell-Shaped topology. Full summary of the One-to-Many topology is provided in Table 7.4.

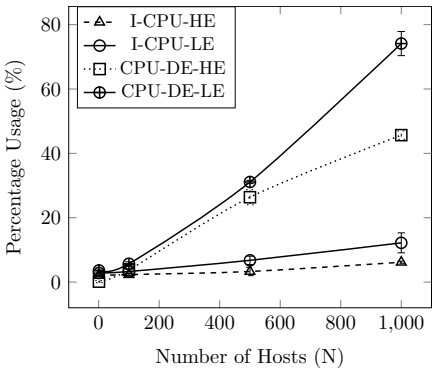
Comparing Dumbbell-Shaped with One-to-Many topologies it is clear that the more nodes are present in the system the more time it takes for both setup and teardown, therefore One-to-Many is faster than Dumbbell-Shaped. For the exact same reason, One-to-Many uses less RAM than Dumbbell-Shaped. In addition the IPD is much higher for Dumbbell-Shaped due to the fact that the controller has to setup flow table rules for two switches, but APD is unaffected.

Low-End System				
	Number of Hosts (N)			
	1	100	500	1000
Setup (s)	0.059 ± 0.003	0.66 ± 0.033	1.23 ± 0.062	7.71 ± 0.39
Teardown (s)	0.098 ± 0.005	2.96 ± 0.15	7.39 ± 0.37	45.97 ± 2.29
I-CPU (%)	2.83 ± 0.25	3.30 ± 0.68	6.77 ± 1.29	12.20 ± 3.1
CPU-DE (%)	3.63 ± 0.1	5.71 ± 0.59	31.11 ± 0.49	74.12 ± 3.75
CCLB	0.12	0.68	2.84	4.33
RAM (MB)	227 ± 0.88	342 ± 34.39	464 ± 15.24	1411 ± 141.8
IPD (ms)	3.65 ± 0.25	5.5 ± 0.49	8.8 ± 0.58	23.8 ± 1.5
APD (ms)	0.07 ± 0.09	0.07 ± 0.06	0.08 ± 0.07	0.08 ± 0.06
NRFR (%)	0.0	0.0	0.0	30.91
FSR (%)	56.37	376.23	351.68	231.76
High-End System				
	Number of Hosts (N)			
	1	100	500	1000
Setup (s)	0.103 ± 0.005	0.652 ± 0.033	3.32 ± 0.166	7.06 ± 0.353
Teardown (s)	0.122 ± 0.006	3.697 ± 0.185	22.29 ± 1.115	44.41 ± 2.22
I-CPU (%)	2.09 ± 0.06	2.33 ± 0.33	3.3 ± 1.302	6.18 ± 0.329
CPU-DE (%)	0.15 ± 0.32	3.81 ± 0.83	26.38 ± 2.35	45.67 ± 1.31
CCLB	0.34	0.889	2.51	1.397
RAM (MB)	396 ± 9.17	438 ± 30.52	941 ± 19.49	1570 ± 155.77
IPD (ms)	3.19 ± 0.22	5.96 ± 0.56	22.7 ± 1.71	48.6 ± 2.91
APD (ms)	0.05 ± 0.08	0.04 ± 0.05	0.06 ± 0.07	0.06 ± 0.04
NRFR (%)	0.0	0.0	0.0	30.91
FSR (%)	75.01	387.75	274.27	221.74

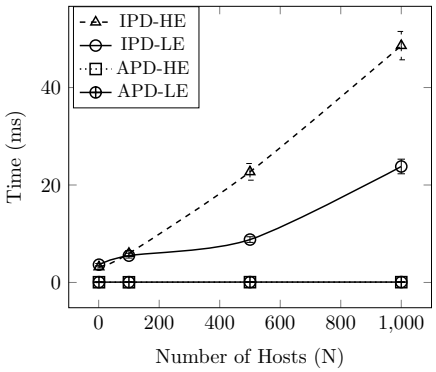
Table 7.4: One-to-Many Topology OVS Summary



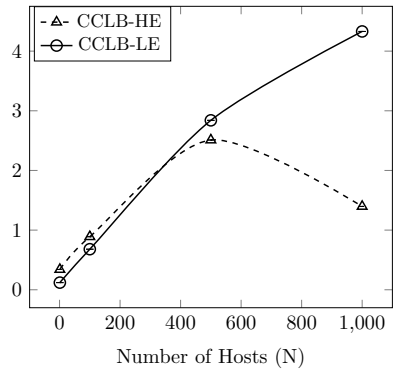
(a) Setup & Teardown Times



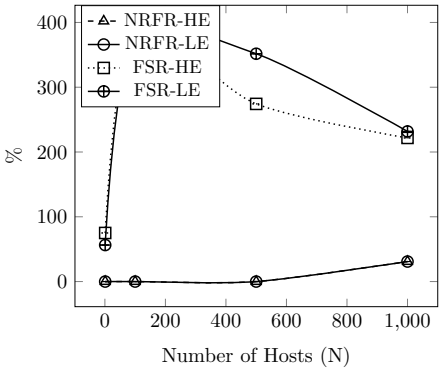
(b) I-CPU & CPU-DE Perc. Usage



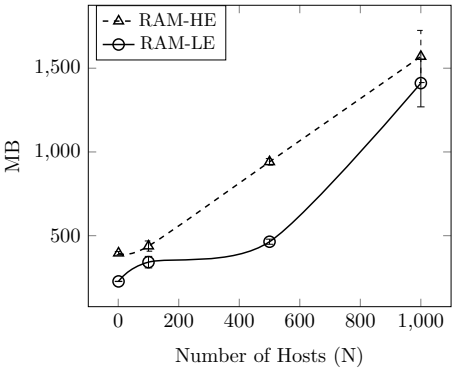
(c) IPD & APD Times



(d) CCLB



(e) NRFR & FSR



(f) RAM

Figure 7.5: One-to-Many Topology Results

7.3.4 Scenario 3 - Linear with 2 Hosts Topology

The third topology as shown in Figure 7.6 is a "Linear" arrangement meaning one switch connected next to each other. This topology was used to examine the effect of several switches on both the ping delay and resource sharing fairness. It consists of only two hosts, one connected to $Switch_1$ and one to $Switch_N$. Ping and iPerf test were performed between hosts $Host_1$ and $Host_2$.

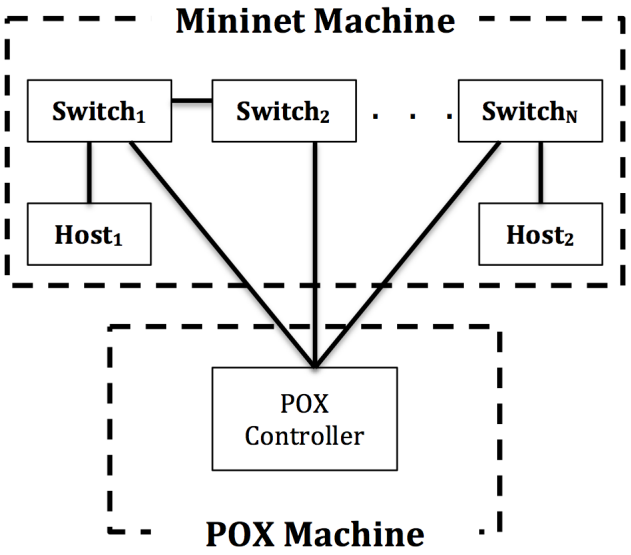


Figure 7.6: Scenario 3 - Linear with 2 Hosts Topology

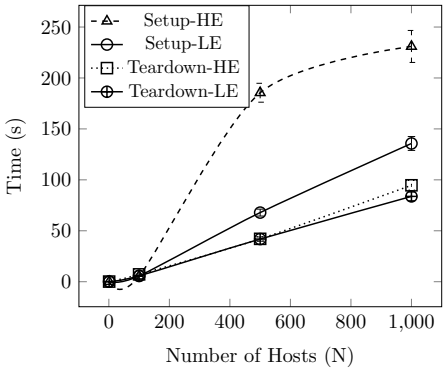
Linear topology with 2 hosts confirmed the suspicion that it takes more time to setup a switch than a host and much less time to teardown a switch than a host. This topology provided two unexpected results, and the first one is the fact that at N=500 and N=1000 switches the Low-End system performed better than the High-End system in both setup and teardown times.

I-CPU had a significant increase from N=1 to N=500 switches in both Low and High-End systems. In all the experiments the High-End system used less I-CPU than the Low-End. CPU-DE was almost identical for both systems. CCLB value indicated once again that load balancing becomes more efficient as the number of switches increases. RAM followed the trend of the previous scenarios which finds the High-End system always using more RAM than the Low-End system. The second unexpected result and the most significant one is APD and IPD. Except from 1 switch, in all the other number of switches the Low-End performed

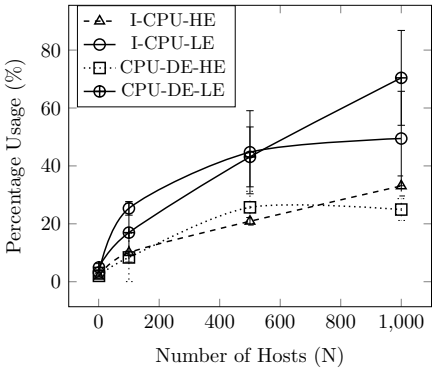
much better than the High-End. Another noticeable number here is the standard deviation of APD which is significantly higher than the average, meaning that the readings have a huge difference between them and the delay is not smooth, in fact is highly unstable. NRFR value is identical for both systems, before at N=100 is zero and then tends to increase as N increases. All of the results are summarised in Table 7.5.

Low-End System				
	Number of Switches (N)			
	1	100	500	1000
Setup (s)	0.109 ± 0.005	5.73 ± 0.287	67.85 ± 3.39	135.7 ± 6.79
Teardown (s)	0.112 ± 0.006	5.73 ± 0.309	41.87 ± 2.094	83.75 ± 4.187
I-CPU (%)	3.17 ± 0.875	25.34 ± 2.35	44.72 ± 14.35	49.46 ± 16.31
CPU-DE (%)	4.95 ± 1.30	16.96 ± 10.19	43.12 ± 10.34	70.41 ± 16.38
CCLB	1.51	11.77	15.02	15.42
RAM (MB)	231 ± 2.58	366 ± 22.05	866 ± 68.47	1642 ± 103.53
IPD (ms)	4.92 ± 0.27	789 ± 50.63	2300 ± 225.8	4740 ± 305.7
APD (ms)	0.07 ± 0.07	16.49 ± 120.26	51.72 ± 306.8	143.35 ± 683.07
NRFR (%)	0.0	0.0	64.79	88.35
High-End System				
	Number of Switches (N)			
	1	100	500	1000
Setup (s)	0.104 ± 0.005	5.47 ± 0.273	185.5 ± 9.276	230.95 ± 15.70
Teardown (s)	0.137 ± 0.007	7.205 ± 0.36	42.09 ± 2.104	94.65 ± 4.732
I-CPU (%)	1.89 ± 0.001	10.03 ± 0.54	20.88 ± 1.31	33.08 ± 3.46
CPU-DE (%)	2.03 ± 0.829	8.39 ± 8.35	25.695 ± 5.53	24.93 ± 3.769
CCLB	0.88	4.03	5.91	8.93
RAM (MB)	308 ± 12.17	433 ± 54.41	961 ± 51.43	1737 ± 158.41
IPD (ms)	1.93 ± 0.15	2183 ± 185.09	3653 ± 312.18	7304 ± 965.2
APD (ms)	0.06 ± 0.07	16.001 ± 119.35	103.4 ± 462.4	353.7 ± 748.2
NRFR (%)	0.0	0.0	64.08	85.48

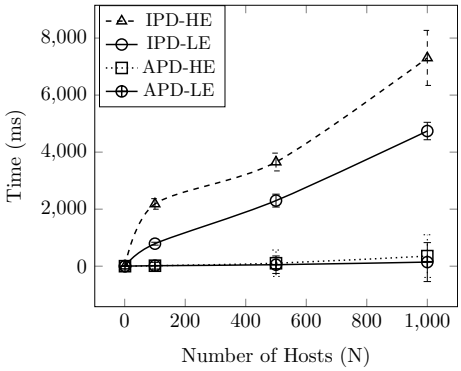
Table 7.5: Linear 2 Hosts Topology OVS Summary



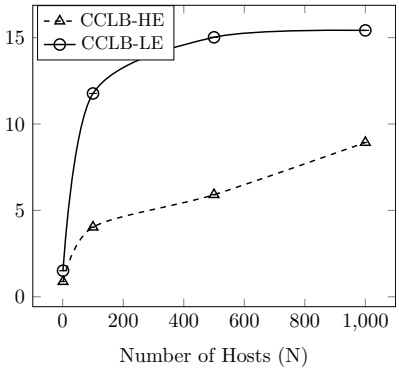
(a) Setup & Teardown Times



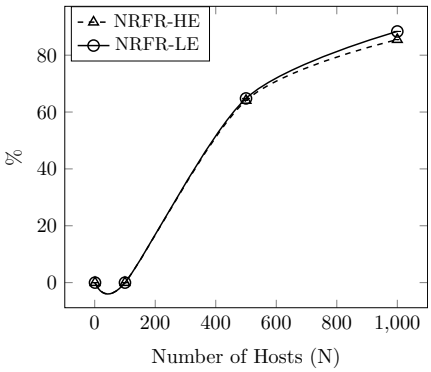
(b) I-CPU & CPU-DE Perc. Usage



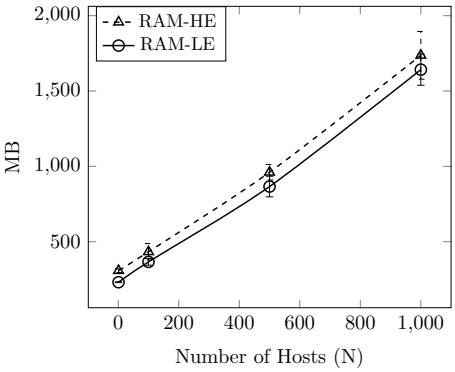
(c) IPD & APD Times



(d) CCLB



(e) NRFR



(f) RAM

Figure 7.7: linear 2 Hosts Topology Results

7.3.5 Scenario 4 - Linear with N Hosts Topology

The fourth topology as shown in Figure 7.8 shares the same "Linear" characteristics as the topology of scenario 3. The only difference is that in this topology each switch is connected to one host and ping and iPerf tests were performed between hosts $Host_1$ and $Host_N$. Even though the extra hosts ($Host_2$ *up to* $Host_{N-1}$) are not actively participating in the experiment, they exist in order to examine if some resources are still assigned to them.

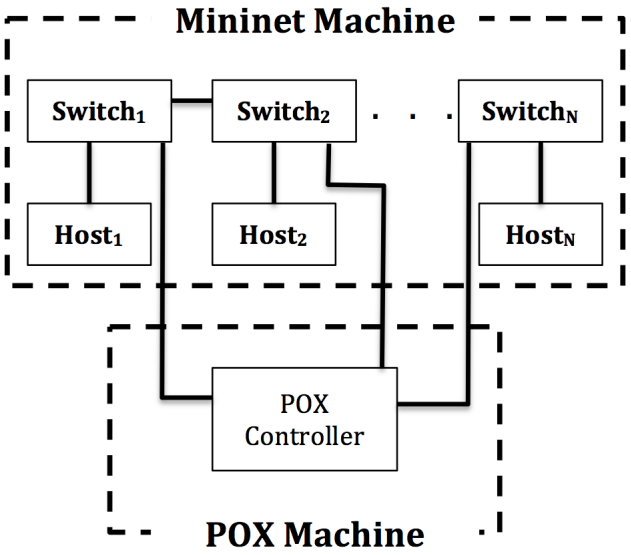


Figure 7.8: Scenario 4 - Linear with N Hosts Topology

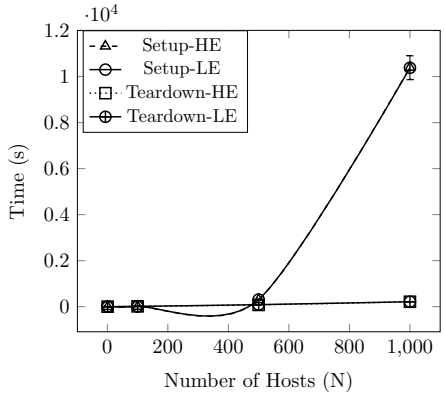
In Linear topology with N hosts once again after a certain number of switches the setup time became higher than the teardown time. At 1000 switches the difference is huge with the setup time being about 50 times higher than teardown time. A noticeable result is the fact that the uncertainty in setup time is more that twice as much as the actual teardown time. Compared to Linear 2 Hosts topology, both Low and High-End systems performed roughly the same.

I-CPU had a less significant increase from 1 to 100 switches compared to Linear 2 Hosts topology. In all the experiments both Low and High-End systems used about the same I-CPU. Following the pattern of all the previous topologies, CCLB becomes more efficient as the number of switches increases. RAM didn't followed the trend of the previous scenarios. In this case both systems used about the same RAM except at 1000 switches where the High-End system used 200MB less than the Low-End system. Furthermore, in this topology the High-End sys-

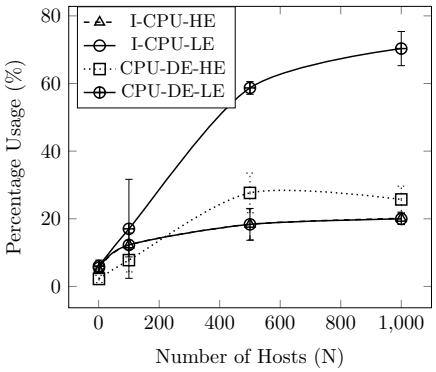
tem performed much better in IPD at large number of switches whereas at low number of switches the Low-End system performed better. In APD, the High-End system performed better except at 100 switches where Low-End system performed better. The number of NRFR showed that at high number of switches it increases dramatically. All of the results are summarised in Table 7.6.

Low-End System				
	Number of Switches (N)			
	1	100	500	1000
Setup (s)	0.057 ± 0.003	8.24 ± 0.412	310.68 ± 15.53	10382.5 ± 519.13
Teardown (s)	0.106 ± 0.005	12.006 ± 0.6	87.91 ± 4.396	213.43 ± 10.67
I-CPU (%)	5.52 ± 0.377	12.27 ± 3.54	18.35 ± 4.71	19.97 ± 1.66
CPU-DE (%)	6.03 ± 1.77	17.03 ± 14.64	58.69 ± 1.88	70.33 ± 5.05
CCLB	2.04	2.10	2.14	3.61
RAM (MB)	327 ± 30.20	577 ± 14.03	1643 ± 230.16	3329 ± 47.31
IPD (ms)	2.4 ± 0.146	1141 ± 66.298	16946 ± 1002.77	130573 ± 3429.63
APD (ms)	0.06 ± 0.05	15.07 ± 115.76	2226.27 ± 4546.2	6042 ± 7134.65
NRFR (%)	0.0	0.0	33.34	95.1
FSR (%)	65.62	410.37	713.42	821.49
High-End System				
	Number of Switches (N)			
	1	100	500	1000
Setup (s)	0.089 ± 0.004	6.72 ± 0.336	299.23 ± 14.96	10397.95 ± 519.898
Teardown (s)	0.128 ± 0.006	13.54 ± 0.677	74.78 ± 3.739	214.29 ± 10.715
I-CPU (%)	5.02 ± 0.677	12.27 ± 1.299	18.35 ± 4.59	20.12 ± 1.85
CPU-DE (%)	2.20 ± 1.09	7.78 ± 3.5	27.65 ± 5.86	25.72 ± 3.93
CCLB	1.16	3.75	4.21	6.26
RAM (MB)	327 ± 25.58	585 ± 74.37	1678 ± 83.55	3129 ± 73.5
IPD (ms)	3.06 ± 0.249	2628 ± 188.76	16855 ± 1149.33	73093 ± 3553.72
APD (ms)	0.04 ± 0.05	24.63 ± 172.35	2045.34 ± 4353.71	4385 ± 5632.8
NRFR (%)	0.0	0.0	6.25	63.29
FSR (%)	79.32	386.64	564.80	728.24

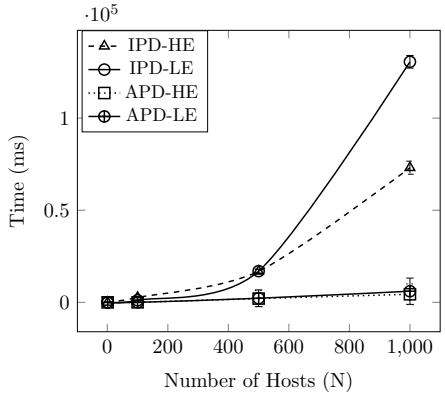
Table 7.6: Linear N Hosts Topology OVS Summary



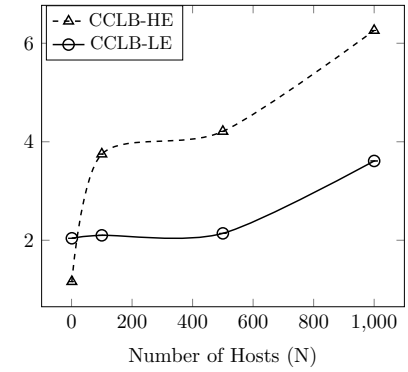
(a) Setup & Teardown Times



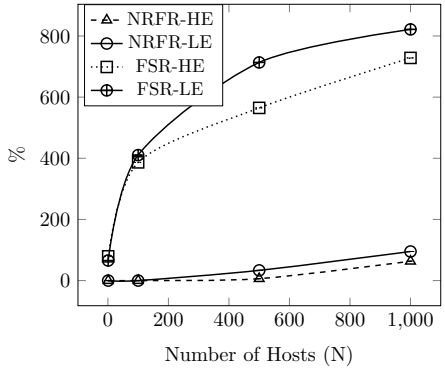
(b) I-CPU & CPU-DE Perc. Usage



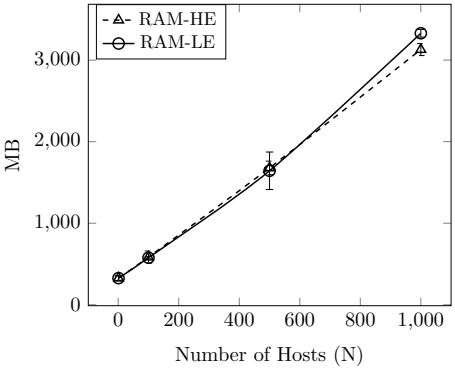
(c) IPD & APD Times



(d) CCLB



(e) NRFR & FSR



(f) RAM

Figure 7.9: Linear N Hosts Topology Results

7.3.6 Scenario 5 - Host-Switch-Host Topology

The fifth topology as shown in Figure 7.10 is a Host-Switch-Host (HSH) topology meaning each of the switches is connected to two separate hosts, and none of the switches are connected together. Its scope is mainly scalability and sharing of resources evaluation. Ping and iPerf tests were performed between hosts $Host_{N-1}$ and $Host_{N-2}$ and switch $Switch_N$ had the request handling responsibility.

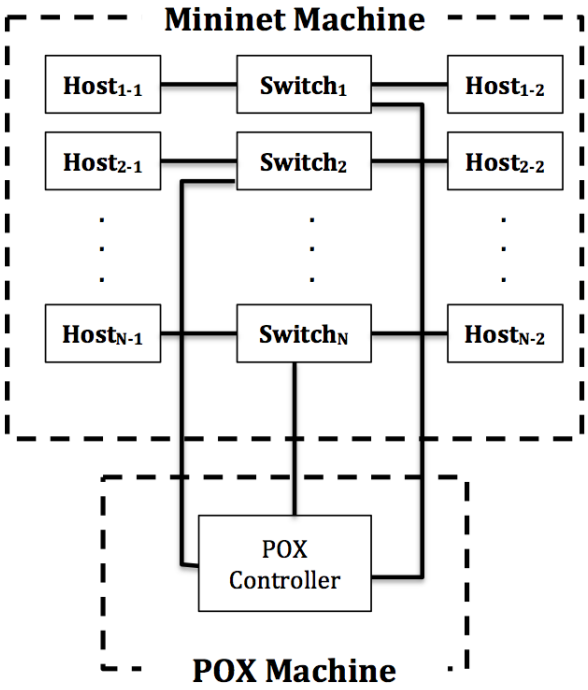


Figure 7.10: Scenario 5 - Host-Switch-Host Topology

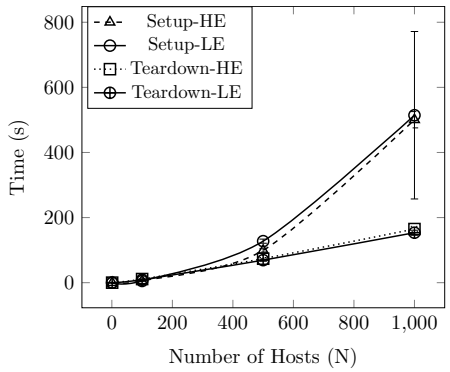
In HSH topology, the setup time is much higher than the teardown time. Compared to Dumbbell-Shaped and One-to-Many topologies (where teardown is higher than setup), the main difference is the number of switches in the topology, therefore it seems that switches take more time to setup and less time to teardown. Furthermore, HSH has from 3 to 3000 hosts but the results indicate that it is not the number of hosts that affects the result since both Dumbbell-Shaped and One-to-Many have increasing number of hosts but teardown remains higher than setup time.

In HSH I-CPU usage in Low-End system is significantly higher than in High-End system whereas CPU-DE for both systems is almost identical. CCLB followed the same pattern as in Dumbbell-Shaped and One-to-Many, becoming more ef-

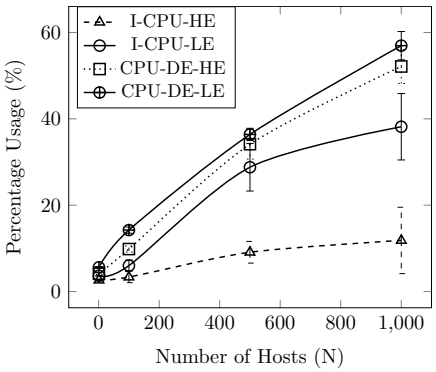
ficient as the number of switches increases. Also RAM usage is almost identical for both systems but in almost all the cases the High-End system uses slightly more RAM. APD is about the same for both systems, as well as the IPD except from the 1000 switches experiment where High-End performs slightly better. In HSH topology, the value of NRFR remains at 0 which shows that all the ping packets reached their destination, therefore they are not affected by the number of switches in the network. All the experimental results for the HSH topology are summarised in Table 7.7.

Low-End System				
	Number of Switches (N)			
	1	100	500	1000
Setup (s)	0.135 ± 0.007	5.274 ± 0.264	127.54 ± 6.34	514.4 ± 257.2
Teardown (s)	0.094 ± 0.005	9.228 ± 0.46	69.22 ± 3.46	153.5 ± 7.67
I-CPU (%)	3.65 ± 0.03	6.01 ± 1.29	28.79 ± 5.51	38.17 ± 7.69
CPU-DE (%)	5.62 ± 0.18	14.24 ± 0.73	36.39 ± 1.42	56.95 ± 3.28
CCLB	0.21	0.84	1.64	3.78
RAM (MB)	237 ± 18.08	569 ± 26.38	2123 ± 75.99	3735 ± 340.49
IPD (ms)	3.34 ± 0.32	13.4 ± 1.16	62.2 ± 5.34	150 ± 15.64
APD (ms)	0.07 ± 0.07	0.07 ± 0.05	0.08 ± 0.09	0.09 ± 0.11
NRFR (%)	0.0	0.0	0.0	0.0
FSR (%)	98.04	321.82	805.00	636.72
High-End System				
	Number of Switches (N)			
	1	100	500	1000
Setup (s)	0.125 ± 0.006	5.75 ± 0.288	99.31 ± 4.965	500.57 ± 25.03
Teardown (s)	0.124 ± 0.006	11.18 ± 0.559	73.71 ± 3.685	164.78 ± 8.239
I-CPU (%)	2.65 ± 0.03	3.35 ± 1.29	9.1 ± 2.51	11.84 ± 7.69
CPU-DE (%)	4.13 ± 0.19	9.84 ± 0.71	34.13 ± 3.46	52.12 ± 3.92
CCLB	0.20	0.75	3.69	4.19
RAM (MB)	310 ± 13.14	649 ± 27.12	2078 ± 12.49	3893 ± 288.58
IPD (ms)	3.4 ± 0.26	15.9 ± 0.88	59.8 ± 5.38	138 ± 9.68
APD (ms)	0.03 ± 0.03	0.09 ± 0.27	0.07 ± 0.07	0.08 ± 0.13
NRFR (%)	0.0	0.0	0.0	0.0
FSR (%)	79.11	323.72	491.38	838.85

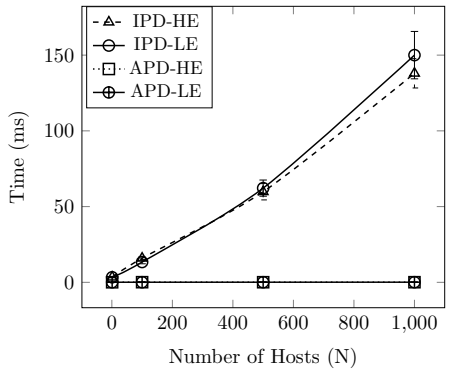
Table 7.7: HSH Topology OVS Summary



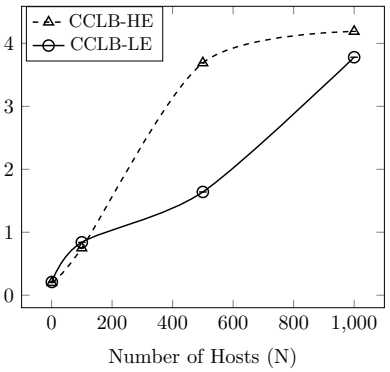
(a) Setup & Teardown Times



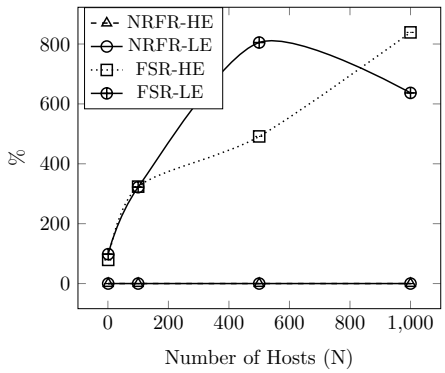
(b) I-CPU & CPU-DE Perc. Usage



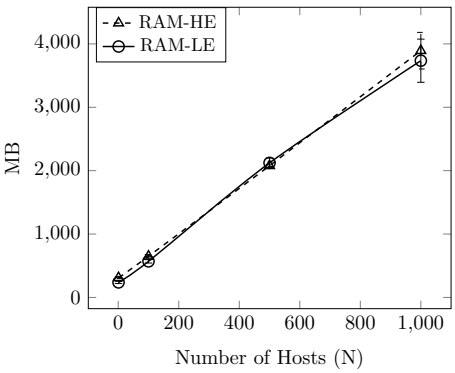
(c) IPD & APD Times



(d) CCLB



(e) NRFR & FSR



(f) RAM

Figure 7.11: HSH Topology Results

7.4 Summary and Discussions

In this Chapter a series of performance tests that can be used in order to examine various SDN experimental platforms are presented. These performance tests indicate the time needed for a platform to create and destroy a topology, the CPU percentage used by each topology both at topology creation and during experimentation as well as the RAM needed. Additionally, both Initial and Average Ping Delays are measured as well as the number of ping packets that failed to reach the destination. Finally, the fairness in sharing of resources by the platform is measured.

Using five scenarios that had the purpose of exposing several bottlenecks and critical performance areas, Mininet Emulator was tested using the proposed set of performance metrics. From the results it is concluded that a) setup time is highly affected by the number of switches, at low number of switches teardown time is much higher than setup time whereas exactly the opposite happens in scenarios with a high number of switches. b) Mininet uses more RAM for the same topology if more RAM is available. c) The number of failed ping packets increases as the number of links included in the packets path increases. d) Initial Ping Delay is huge compared to Average Ping Delay. e) Load balancing between CPU cores becomes more efficient as the number of nodes, in the topology, increases.

Chapter 8

OpenFlow Software & Hardware Performance Evaluation

8.1 Introduction

The OpenFlow protocol has been implemented on several platforms including simulators, emulators, physical switches, NetFPGA or even cheap home routers. All these platforms might have a complete or partial implementation of OpenFlow but the real question is how well do they perform. Through their performance one can observe how feasible it is to implement the OpenFlow protocol on various platforms.

This chapter presents a performance evaluation of OpenFlow implementation on three different platforms. A high profile enterprise OpenFlow Switch (HP Procurve 3500-24), a low profile home router with custom firmware (TP-Link with OpenWrt) and a very common OpenFlow emulator (Mininet). All of the detailed figures of this Chapter are presented in Appendix A.

The sections of this chapter can be summarised as follows:

- Section 8.2 (HP-Procurve, OpenWrt and Mininet Specifications): Introduces the platforms that will be used for the feasibility study.
- Section 8.3 (Performance Evaluation Scenarios): Presents details of the experiments performed as well as the purpose of each of the experiments.

- Section 8.4 (Experimental Results Analysis): Analyses the results of the experiments.
- Section 8.5 (Summary and Discussions): Concludes the chapter, summarising all the important findings and pointing out some useful comparisons resulted from experimentation and analysis.

8.2 HP Procurve, OpenWrt and Mininet Specifications

HP Procurve 3500-24 switch is a 24-port OpenFlow-enabled switch. Due to the fact that this switch's firmware is managed by a vendor, OpenFlow versions tend to be integrated in a very slow pace. Currently this switch supports OpenFlow up to version 1.3, but one of the major drawbacks is that not all of the OpenFlow v1.3 functions are supported. On the other hand, it provides full support for OpenFlow v1.0. According to the vendor, HP, this switch has very good specifications, some of which are listed in Table 8.1. Due to the fact that it has VLAN ability, users can create several OpenFlow VLAN switches as long as the number of physical ports left can support them.

OpenWrt [147] is a highly extensible GNU/Linux distribution for embedded devices. Unlike many other distributions for routers, OpenWrt is built from the ground up to be a full-featured, and easily modifiable operating system. In practice, this means that users can have all the features their devices can support, powered by a Linux kernel that's more recent than most other distributions. This eliminates the need for users to wait from vendors to implement new functions and also provides them with the ability to create their own functions.

OpenWrt OpenFlow support was developed as part of the "Pantou" project [146], and tested on several devices. It has been proven that it is capable to run on devices having Broadcom [149] chipsets as well as TP-Link TL-WR1043ND [150] router which runs on an Atheros [151] chipset. The main limitations of OpenWrt when it acts as an OpenFlow switch are two. *A)* It has a limited number of ports since it is built on home routers. *B)* Home routers do not have the CPU

Component	Details
CPU	PowerPC 8540 at 666MHz
Flash Memory	4MB
Compact Flash	128MB
RAM	256MB DDR
Wired Ports	20 RJ-45 10/100
Dual-personality ports	4 RJ-45 10/100/1000 or Mini-GBIC
USB	1 USB v.2.0
100Mb Latency	$<3.4\mu s$
1000Mb Latency	$<2.9\mu s$
Throughput	8.9 million 64-byte pps
Switching Capacity	12Gbps
Routing table size	10000 entries
MAC address table size	64000 entries

Table 8.1: HP Procurve Specifications

capabilities that a switch has. For the purpose of the scenarios described in section 8.3, TP-Link TL-WR1043ND router has been used. Hardware details about this router are shown in Table 8.2.

Component	Details
CPU	400MHz
Chipset	Atheros AR9132
Wireless NIC	Atheros 9100
Wireless Standard	11 b/g/n
Flash Memory	8MB
RAM	32MB
Wired Ports	5 gigE
USB	1 USB v.2.0

Table 8.2: TP-Link TL-WR1043ND Specifications

Mininet is highly dependent on the hardware capabilities due to the fact that as an emulator it uses CPU power to create and run a virtual environment for every device the user creates. Implementing network scenarios in Mininet can be done through a simple Command Line Interface (CLI) or by using custom scenar-

ios implemented in Python programming language. CLI is preferred for simple scenarios but if the user wants to implement complex scenarios then this has to be done using Python. Using custom scripts the user can take full control of all the devices created by Mininet (such as hosts, switches or links). In order to help researchers to build their own custom networks, Mininet provides its own Application Programming Interface (API) which comes with a very useful documentation. One of the downsides of the documentation is that it was written for OpenFlow version 1.0, whereas users can use up to OpenFlow version 1.3. However, most of the structure as well as the commands remain almost the same therefore it does not cause a lot of problems. It is an open source software and is maintained by the community, which gives the advantage of fast response to problems or bugs found during experimentation.

8.3 Performance Evaluation Scenarios

In order to perform the performance evaluation, it was decided to run a series of tests on the three platforms and then analyse the results. Here is a list of the scenarios used:

Scenario 1 - UDP Bandwidth

(a) Maximum UDP Bandwidth

(b) Maximum UDP Bandwidth Stability

Scenario 2 - Multiple Streams

Scenario 3 - Bidirectional Traffic

Scenario 4 - Rate Limiting

Scenario 5 - TCP Bandwidth

Scenario 6 - TCP and UDP Bandwidth

The first and fifth scenarios were set up to determine the maximum bandwidth that can be achieved. This will indicate how powerful the platforms under test

are in the area of packet handling and how comparable Mininet and OpenWrt router are against an enterprise level switch. Scenario 1.b was conducted due to the fact that after running the scenario 1.a the results indicated that Mininet and OpenWrt values were not very consistent (as shown in Section 8.4). Therefore, it was decided to run an extra experiment for those two platforms only, in order to find out the stability they can achieve at the maximum bandwidth.

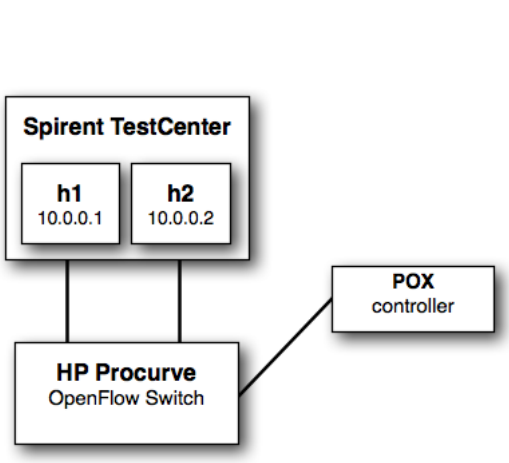
The second scenario added some extra streams to the traffic in order to stress the platforms under test more, and observe how they react when they have more unknown packets and how well they handle them. Similarly, the third scenario was all about the ability of the platforms to handle bidirectional traffic.

The fourth scenario checked one of the main features of OpenFlow which is rate limiting. This allowed us to conclude how well some features of the OpenFlow Protocol are designed on these three platforms. The sixth and final scenario consisted of a mixture of TCP and UDP traffic at their maximum bandwidth.

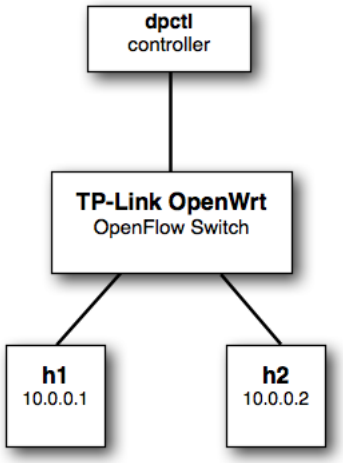
All of the scenarios have been set up using a two hosts topology and a manual controller as shown in Figure 8.1

For the purpose of HP Procurve switch testing, Spirent TestCenter [152] was used to create traffic and record all the appropriate performance metrics whereas for both Mininet and TP-Link OpenWrt iPerf was used as the traffic source.

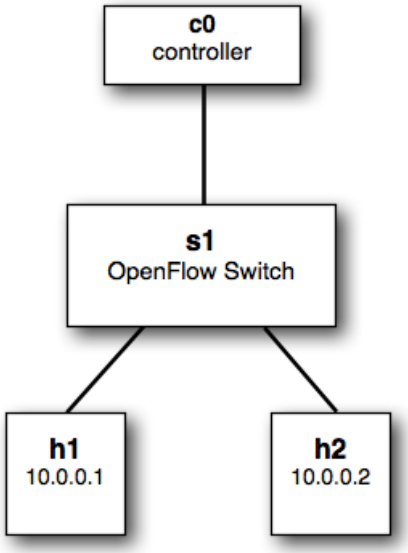
For the purpose of Mininet experiments, the machine shown in Table 8.3 has been used. Having in mind that Mininet *(a)* is highly depended on the hardware and *(b)* is affected by background processes, except from network metrics such as bandwidth, delay and packet loss, it is important to know the impact Mininet has on the machine used to perform the experiment. Therefore, readings from the system such as overall CPU usage, individual processor usage, Random Access Memory (RAM) usage as well as network I/O activity were taken. To ensure a fair comparison, system benchmarking (Section 8.4.1) before running Mininet was taken in order to know the system's default performance. In all of the Mininet experiments, OpenFlow version 1.3 was used.



(a) HP Procurve



(b) OpenWrt



(c) Mininet

Figure 8.1: Two Hosts Topology Used By Each Platform

Component	Details
Processor	2.13GHz Intel Core 2 Duo P7450 (64-bit)
Microprocessor Cache	Level 2 cache 3MB
RAM	4GB DDR2 800MHz
Hard Drive	250GB SATA 5400rpm
Operating System	Ubuntu 12.04 LTS

Table 8.3: Mininet Experimenting Machine

8.4 Experimental Results Analysis

In this section, the experimental results as well as a thorough analysis of the six scenarios is presented.

8.4.1 Mininet System Default Performance

In the system default performance, the processes running by the system were chosen to be exactly the same as the ones running in all of the scenarios. The exception was the fact that Mininet was not running. That allowed us to monitor the system without Mininet running, in order to normalise our Mininet results afterwards. Both wired and wireless networking were disabled to isolate the system, whereas throughout the experiment all the processes were monitored in order to make sure that they remained unchanged.

Throughout the experiment both cores of the CPU were using only a small fraction of their processing power. These resources were used mainly by operating system processes as well as the software used to monitor the system. The summary table (Table 8.4) shows that only around 0.50% of the processing power of each CPU core was used. RAM on the other hand is almost unchanged for the duration of the experiment. Table 8.4 shows that the maximum active RAM is at 276.6MB whereas the minimum active is at 275.1MB. Finally, the hard drive has some activity in both reading and writing; but this can be taken to be the software that takes system measurements since it outputs the measurements in a text document in real-time. Even though the disk busy percentage has some peaks, these peaks are only for *sda* and *sda7*, therefore the average disk busy percentages is only 0.19% (Table 8.4 and Appendix A Figure A.1).

8.4.2 Scenario 1.a - Bandwidth

Scenario 1.a was set up in order to find the maximum throughput that can be achieved by the three platforms. Furthermore, performance metrics such as delay, packet loss, latency and CPU usage were recorded. All the flow table entries were defined manually prior to the experiment in order to eliminate the controller's

Parameter	Value	Parameter	Value
Core 1 Average Usage (%)	0.59	System Total RAM (MB)	4002.30
Core 1 Maximum Usage (%)	37.80	Average Active RAM (MB)	276.02
Core 1 Minimum Usage (%)	0	Minimum Active RAM (MB)	275.10
Core 1 Usage Standard Deviation	2.42	Maximum Active RAM (MB)	276.60
Core 2 Average Usage (%)	0.54	Active RAM Standard Deviation	0.46
Core 2 Maximum Usage (%)	16.40	Total Disk Write (KB)	3627
Core 2 Minimum Usage (%)	0	Average Disk Write (KB)	1.73
Core 2 Usage Standard Deviation	1.23	Maximum Disk Write (KB)	115.70
Network I/O Total Read (KB)	6.50	Minimum Disk Write (KB)	0
Network I/O Average Read (KB)	0.02	Disk Write Standard Deviation	8.56
Network I/O Maximum Read (KB)	1.30	Total Disk Read (KB)	5359.40
Network I/O Minimum Read (KB)	0	Average Disk Read (KB)	2.55
Network I/O Read Standard Deviation	0.17	Maximum Disk Read (KB)	1322.30
Network I/O Total Write (KB)	6.50	Minimum Disk Read (KB)	0
Network I/O Average Write (KB)	0.02	Disk Read Standard Deviation	52.31
Network I/O Maximum Write (KB)	1.30	Average Disk Busy (%)	0.19
Network I/O Minimum Write (KB)	0	Maximum Disk Busy (%)	20
Network I/O Write Standard Deviation	0.17	Minimum Disk Busy (%)	0
		Disk Busy Standard Deviation	0.98

Table 8.4: Default CPU Usage - Summary

performance from affecting the experiment. In this scenario host *h1* was sending UDP traffic at predefined bandwidths to host *h2* via the OpenFlow Switch (HP-Procurve, TP-Link OpenWrt, Mininet OpenVSwitch). In the case of the HP Procurve switch, the experiment was repeated twice, one using the RJ-45 10/100 ports and one using the RJ-45 10/100/1000 ports.

8.4.2.1 TP-Link OpenWrt

The maximum bandwidth that was achieved did not exceed 40Mbps, whereas delay increased significantly at lower sender (host *h1*) bandwidths and decreased as the sender's bandwidth was increased. At 1Mbps apart from the significantly higher delay average, the standard deviation was also very high, which indicates that the value was not stable (Table 8.5). Upon repeating the 1Mbps bandwidth experiment, the same results were obtained which proves that it is not an initial-

isation problem. The average latency was around 0.75ms. Looking at the CPU performance it is clear that it increases as the bandwidth increases. The same happened to the packet loss percentage.

Having in mind that during the experiments, the CPU Usage measuring tool was running and consuming some of the CPU's processing power, it can be concluded that the router can perform slightly better but the change will be almost insignificant. Looking at the packet loss percentage, the delay as well as the CPU usage it is concluded that the best bandwidth to perform the rest of the experiments would be at 20Mbps. At 20Mbps the average CPU usage was 52.6% which allows another 47.4% CPU resources to be used for custom functions that will be implemented in the future. (Appendix A Figures A.2-A.6)

Sender Bandwidth (Mbps)	**1**	**10**	**20**	**25**	**30**	**40**	**50**
Average Achieved Bandwidth (Mbps)	1	9.97	19.88	24.64	29.21	37.68	36.95
Bandwidth Standard Deviation	0.01	0.28	0.26	0.40	0.55	1.86	0.96
Average Delay (ms)	0.76	0.14	0.10	0.15	0.14	0.20	0.17
Delay Standard Deviation	1.01	0.27	0.21	0.22	0.25	0.27	0.19
Packet Loss (%)	0	0.21	0.57	1.53	2.62	5.64	26.10
Average CPU Usage (%)	7.59	29.65	52.55	59.93	67.82	78.90	78.17
Maximum CPU Usage (%)	76	45	77	74	84	95	94
Minimum CPU Usage (%)	0	3	3	0	0	3	2
CPU Usage Standard Deviation	4.86	8.50	17.10	17.29	19.53	23.02	22.56

Table 8.5: TP-Link Scenario 1.a - Summary

8.4.2.2 Mininet

Scenario 1.a performed in Mininet yielded some very interesting results (Table 8.6). The highest bandwidth Mininet can reach is at 130Mbps for the machine shown in Table 8.3. This is also confirmed by the percentage of packets lost at 130Mbps which is 0%. Due to the fact that 130Mbps is the maximum, upon experimenting, it would be better to use a lower bandwidth than 130Mbps in order to avoid

any problems that may arise due to pushing Mininet to its limits. Furthermore, at 100Mbps, the average CPU usage on both cores has the lowest standard deviation. This means at that speed we have the lowest variability in the results, therefore the CPU is at a smooth state where processes are equally shared and processed. Delay on the other hand follows an unorthodox pattern. Delay is high at lower bandwidths, then after 100Mbps it begins to drop until the bandwidth reaches 130Mbps. Afterwards it increases rapidly and becomes four times bigger at 200Mbps. The most interesting result of this experiment is the CPU performance of a sample of three different bandwidths. As shown, at both 1Mbps and 100Mbps the processes are spread evenly to the two cores of the CPU. However, at very high bandwidths, Mininet has the tendency to overload the first core and then use the second core for the remaining processes. (Appendix A Figures A.7-A.12)

8.4.2.3 HP-Procurve

In contrast to the other OpenFlow platforms, there was no need to repeat the experiment with a number of different bandwidths. HP Procurve switch was able to reach the maximum bandwidth supported by the ports from the first run. More specifically, RJ-45 10/100 ports run was able to reach an average of 100Mbps bandwidth with excellent stability. Delay performance was more impressive than bandwidth, with an average of 0.0004 microseconds (μs). Latency, with an average of 15μs, was not as impressive as the rest of the results. Using RJ-45 10/100/1000 ports, the bandwidth again reached the maximum supported at an average of 1000Mbps. Delay was slightly increased with an average of 0.0048μs whereas latency was reduced at an average of 4μs. All the results are summarised in Table 8.7 and Appendix A Figures A.13-A.14.

8.4.3 Scenario 1.b - Bandwidth Stability

In scenario 1.b, using the bandwidth it was concluded to be the best for experimenting in scenario 1.a, the overall stability of Mininet and OpenWrt was tested. Topology and controller settings were exactly the same as in scenario 1.a. During the experiment, CPU Usage measuring tool was not used in order to get a more isolated performance measurement.

Sender Bandwidth (Mbps)	1	50	100	110	200
Average Achieved Bandwidth (Mbps)	1	50.04	100.50	110.50	130.10
Bandwidth Standard Deviation	0.002	0.011	0.034	1.929	0.459
Average Delay (ms)	0.01	0.01	0.01	0.01	0.04
Delay Standard Deviation	0.008	0.007	0.006	0.006	0.008
Packet Loss (%)	0	0	0.01	0.38	35.85
Core 1 Average Usage (%)	2.20	34.48	53.51	59.09	96.02
Core 1 Maximum Usage (%)	57	61.40	62.90	100	100
Core 1 Minimum Usage (%)	0	9.90	8.80	44.20	10.80
Core 1 Usage Standard Deviation	3.35	11.75	4.50	9.46	17.07
Core 2 Average Usage (%)	2.04	32.01	52.07	56.06	20.20
Core 2 Maximum Usage (%)	26.30	58.60	61.70	100	100
Core 2 Minimum Usage (%)	0	5.70	7.60	16.90	10.50
Core 2 Usage Standard Deviation	1.79	11.27	4.68	10.30	16.88
Network I/O Total Read (KB)	37580	1881406	3777886	4171356	7622886
Network I/O Average Read (KB/s)	125.27	6271.40	12593	13905	25409.60
Network I/O Maximum Read (KB/s)	126.80	6286.80	12626	13936	25466.50
Network I/O Minimum Read (KB/s)	59	4365.50	8087.3	7073.90	18330.30
Network I/O Read Standard Deviation	4.29	142.11	307.01	395.70	579.16
Network I/O Total Write (KB)	37582	1881408	3777511	4155330	4890018
Network I/O Average Write (KB/s)	125.27	6271.36	12591.70	13851.10	16300.10
Network I/O Maximum Write (KB/s)	127.20	6293	12631.70	13965.10	16744.30
Network I/O Minimum Write (KB/s)	59	4365.50	8088.80	7001.90	11434.60
Network I/O Write Standard Deviation	4.29	142.11	307.70	463.90	383.69

Table 8.6: Mininet Scenario 1.a - Summary

	10/100 Ports	10/100/1000 Ports
Sender Bandwidth (Mbps)	100	1000
Average Achieved Bandwidth (Mbps)	100	1000
Average Delay (μs)	0.0004	0.005
Packet Loss (%)	0	0
Average Latency (μs)	15.09	4.04

Table 8.7: HP Procurve Scenario 1 - Summary

8.4.3.1 TP-Link OpenWrt

The results of scenario 1.b proved that the router is stable enough to justify further experimentation. The bandwidth is very stable with an average of 19.99Mbps (Table 8.8). Delay can be considered very stable as well. Although delay had some spikes that reached 0.45ms, the average was 0.07ms with a standard deviation of 0.07. Finally, packet loss percentage never exceeded 0.09% with the average being 0.021%. (Appendix A Figures A.15-A.16)

Average Bandwidth (Mbps)	19.99
Maximum Bandwidth (Mbps)	20.44
Minimum Bandwidth (Mbps)	12.41
Bandwidth Standard Deviation	0.17
Average Delay (ms)	0.07
Maximum Delay (ms)	2.22
Minimum Delay (ms)	0.02
Delay Standard Deviation	0.07
Average Packet Loss (%)	0.02
Maximum Packet Loss (%)	0.09
Minimum Packet Loss (%)	0.001
Packet Loss Standard Deviation	0.03

Table 8.8: TP-Link Scenario 1.b - Summary

8.4.3.2 Mininet

The results of scenario 1.b show that Mininet is stable at 100Mbps for the machine shown in Table 8.3. The average bandwidth is measured to be 100.47Mbps with a standard deviation of 0.33 (Table 8.9). Delay average value came to be 0.0092ms with a standard deviation of 0.0064 whereas the maximum delay recorded was only 0.133ms (Table 8.9). Packet loss on the other hand, was negligible due to the fact that the average loss was 0.0177% with a standard deviation of 0.0004 (Table 8.9). Finally, the processes were evenly split to both CPU cores with an average activity of 53.048% (Table 8.9, Appendix A Figures A.17-A.18).

Average Bandwidth (Mbps)	100.47
Maximum Bandwidth (Mbps)	101.25
Minimum Bandwidth (Mbps)	89.51
Bandwidth Standard Deviation	0.33
Average Delay (ms)	0.01
Maximum Delay (ms)	0.13
Minimum Delay (ms)	0.001
Delay Standard Deviation	0.01
Average Packet Loss (%)	0.02
Maximum Packet Loss (%)	0.11
Minimum Packet Loss (%)	0.0004
Packet Loss Standard Deviation	0.03
Average CPU Usage (%)	53.05
CPU Usage Standard Deviation	2.48

Table 8.9: Mininet Scenario 1.b - Summary

8.4.4 Scenario 2 - Multiple Streams

The second scenario was carried out to test the ability of the platforms to handle more than one UDP data streams. For OpenWrt host *h1* was sending four UDP streams, 5Mbps each, to host *h2*. For HP Procurve, the experiment was repeated twice; in the first run using RJ-45 10/100 ports and in the second run using RJ-45 10/100/1000 ports. Host *h1* sent five UDP streams, 20Mbps each for the RJ-45 10/100 ports run and 200Mbps each for the RJ-45 10/100/1000 ports run, to host *h2* via HP Procurve switch. For Mininet the scenario has been repeated in two different ways. In the first run, the topology included two hosts, a switch and a controller. Host *h1* sent five UDP streams, 20Mbps each, to host *h2* via switch *s1*. In the second run, the topology included six hosts, a switch and a controller. Hosts *h1*, *h2*, *h3*, *h4* and *h5* sent one 20Mbps UDP stream each to host *h6* via switch *s1*.

8.4.4.1 TP-Link OpenWrt

Scenario 2 proved that TP-Link router with OpenWrt firmware performs well during multiple streams. Bandwidth was very stable throughout the duration

of the experiment, with an average of 5Mbps and a standard deviation of 0.01 (Table 8.10). Delay had some spikes that reached 2ms but it performed well with an average of 0.33ms and a standard deviation of 0.23. The average packet loss percentage was very low at 0.0078% with a standard deviation of 0.004. Looking at the three metrics analysed (bandwidth, delay and packet loss) it is concluded that the performance of TP-Link is acceptable. (Appendix A, Figures A.19-A.20)

Average Bandwidth (Mbps)	4.99
Maximum Bandwidth (Mbps)	5.08
Minimum Bandwidth (Mbps)	4.85
Bandwidth Standard Deviation	0.015
Average Delay (ms)	0.33
Maximum Delay (ms)	2.01
Minimum Delay (ms)	0.09
Delay Standard Deviation	0.23
Average Packet Loss (%)	0.01
Maximum Packet Loss (%)	0.01
Minimum Packet Loss (%)	0.003
Packet Loss Standard Deviation	0.004

Table 8.10: TP-Link Scenario 2 - Summary

8.4.4.2 Mininet

The results of scenario 2 show two phenomena. *(a)* In both runs, the bandwidth is very stable at about 20Mbps. There is a slight drop of the bandwidth in the initial seconds of the second run but this is not enough to significantly affect the average value. This could be due to an initialisation problem of Mininet, which will be resolved with further experimentation. *(b)* With the use of two hosts the average delay is around 0.07ms with slight variations, whereas upon using six hosts the delay gets more significant variations but the average delay drops at 0.05ms (Appendix A, Figures A.21-A.22 and Table 8.11).

	Run 1	Run 2
Average Bandwidth (Mbps)	19.99	19.98
Maximum Bandwidth (Mbps)	20.02	20.31
Minimum Bandwidth (Mbps)	19.97	15.76
Bandwidth Standard Deviation	0.01	0.24
Average Delay (ms)	0.07	0.06
Maximum Delay (ms)	0.13	1.06
Minimum Delay (ms)	0.02	0.01
Delay Standard Deviation	0.02	0.06

Table 8.11: Mininet Scenario 2 - Summary

8.4.4.3 HP-Procurve

In both runs of the experiment, the bandwidth was handled very well by the switch. The bandwidth in the first run was very stable at 20Mbps for all streams. The same happened in the second run of the experiment where the bandwidth was very stable at 200Mbps.

Delay on the other hand did not follow the same pattern as bandwidth. In the first run, delay was the same for all streams with an average of 0.002μs, even though they were some variations of 0.0015μs. In the second run the results were slightly different. Firstly, the overall delay was about ten times smaller than the first run. Secondly, not all streams shared the same average delay. All the results of the experiment, including individual delay and bandwidth are summarised in Table 8.12 and Appendix A, Figures A.23-A.24.

8.4.5 Scenario 3 - Bidirectional Traffic

The objective of Scenario 3 was to test the ability of the platforms in handling bidirectional traffic. For OpenWrt both hosts were sending 10Mbps UDP traffic to each other, whereas for Mininet 50Mbps of UDP traffic was used. Finally for HP Procurve, the experiment was repeated twice, for RJ-45 10/100 ports the bandwidth was set at 100Mbps and for RJ-45 10/100/1000 ports the bandwidth was set at 1000Mbps.

	10/100 Ports				
Sender Bandwidth (Mbps)	20				
Stream Number	1	2	3	4	5
Average Achieved Bandwidth (Mbps)	19.99	19.99	19.99	19.99	19.99
Average Delay (μs)	0.002	0.002	0.002	0.002	0.002

	10/100/1000 Ports				
Sender Bandwidth (Mbps)	200				
Stream Number	1	2	3	4	5
Average Achieved Bandwidth (Mbps)	199.99	199.99	199.99	199.99	199.99
Average Delay (μs)	0.0002	0.0003	0.0002	0.0003	0.0003

Table 8.12: HP Procurve Scenario 2 - Summary

8.4.5.1 TP-Link OpenWrt

As shown in Figure A.25a bandwidth had some slight variations throughout the experiment, but were not enough to affect the overall bandwidth performance which resulted in 9.99Mbps for both flows. The standard deviation was 0.1 which again proves that the variations were very small to affect the results (Table 8.13). Delay on the other hand has more serious variations some of which reached 1.06ms but on average it performed well with 0.10ms for the first flow (host *h1* to host *h2*) and 0.13ms for the second flow (host *h2* to host *h1*) as shown in Table 8.13. CPU usage can be considered very unstable due to the fact that the standard deviation resulted in 9.55. The average CPU usage reached 46.6%. (Appendix A, Figures A.25-A.26)

8.4.5.2 Mininet

Throughout the duration of the experiment, Mininet kept the bandwidth stable at around 50Mbps for both flows. The average delay was at 0.024ms for the flow from host *h1* to host *h2* whereas, the delay of the opposite flow was at 0.028ms. Standard deviation of both bandwidth and delay for both flows was kept at very low levels which indicated the stability of the experiment (Table 8.14, Appendix A Figure A.27).

	Host 1 to Host 2	Host 2 to Host 1
Average Bandwidth (Mbps)	9.99	9.99
Maximum Bandwidth (Mbps)	10.69	10.83
Minimum Bandwidth (Mbps)	8.80	8.66
Bandwidth Standard Deviation	0.10	0.11
Average Delay (ms)	0.10	0.13
Maximum Delay (ms)	0.94	1.06
Minimum Delay (ms)	0.02	0.03
Delay Standard Deviation	0.14	0.17
Packet Loss (%)	0.02	0.02
Average CPU Usage (%)	46.63	
Maximum CPU Usage (%)	74	
Minimum CPU Usage (%)	23	
CPU Usage Standard Deviation	9.55	

Table 8.13: TP-Link Scenario 3 - Summary

	h1 to h2	h2 to h1
Average Bandwidth (Mbps)	50.04	50.04
Maximum Bandwidth (Mbps)	50.15	50.12
Minimum Bandwidth (Mbps)	49.59	49.97
Bandwidth Standard Deviation	0.03	0.01
Average Delay (ms)	0.02	0.03
Maximum Delay (ms)	0.11	0.10
Minimum Delay (ms)	0.001	0.004
Delay Standard Deviation	0.02	0.02

Table 8.14: Mininet Scenario 3 - Summary

8.4.5.3 HP-Procurve

The results of scenario 3 for HP-Procurve were found to be very interesting. In the RJ-45 10/100 ports run, the bandwidth for both flows was 100Mbps which is the maximum that the RJ-45 10/100 ports can support. Delay was showing some slight variations, at an average of 0.0004μs. In the RJ-45 10/100/1000 ports run, the bandwidth reached 860Mbps for both ports, whereas delay increased as well at an average of 0.0025μs for the first stream namely, *h1* to *h2*, and 0.003μs for

the second stream namely, *h2* to *h1*. As shown in Table 8.15, even though RJ-45 10/100/1000 ports can handle up to 1000Mbps, using bidirectional traffic, HP Procurve switch could only handle 870Mbps. (Appendix A, Figures A.28-A.29)

	10/100 Ports		10/100/1000 Ports	
	h1 to *h2*	*h2* to *h1*	*h1* to *h2*	*h2* to *h1*
Sender Bandwidth (Mbps)	100	100	870	870
Average Achieved Bandwidth (Mbps)	100	100	870	870
Average Delay (μs)	0.0004	0.0004	0.003	0.003

Table 8.15: HP Procurve Scenario 3 - Summary

8.4.6 Scenario 4 - Rate Limiting

Scenario 4 objective was to test the rate limiting function present in the platforms. Mininet and TP-Link router rate limiting was part of OpenFlow version 1.3. In HP Procurve switch there is no such OpenFlow function due to the fact that it supports OpenFlow version 1.0. HP has implemented its own rate limiting function which is part of the Procurve switch firmware. For TP-Link, host *h1* was sending 20Mbps UDP stream to host *h2*, and the experiment was repeated three times, each of them having different rate limit value. Namely, the rate limit values used were 1Kbps, 5Mbps and 10Mbps. The exact same process was performed in Mininet as well, with the only difference that the traffic stream used was 100Mbps and the rate limits were 1Kbps, 25Mbps and 50Mbps. For HP-Procurve topology included two hosts (host *h1* and host *h2*), with *h1*, sending 100Mbps stream for the RJ-45 10/100 ports and 1000Mbps stream for the RJ-45 10/100/1000 ports, to *h2* via HP procure switch. On the switch a rate limit value was defined. The experiment was repeated two times for each type of ports, each time using a different rate limit. Namely, the rate limits were 1Kbps and 50Mbps for the RJ-45 10/100 ports test and 1Kbps and 500Mbps for the RJ-45 10/100/1000 ports test.

8.4.6.1 TP-Link OpenWrt

The rate limiting function performed satisfactory for all the limiting values. The average bandwidth was kept at the rates specified. More specifically, it was kept at 0.001Mbps for the 1Kbps limit, 5.36Mbps for the 5Mbps limit and 10.7Mbps for the 10Mbps rate limit (Table 8.16, Appendix A Figure A.30). The delay of 1Kbps rate limit was much higher than the delay of the rest of the limiting rates. At some points the delay exceeded 5ms, but on average it reached 1.56ms for 1Kbps rate limit. Delay for the rest of the rate limits was 0.06ms.

	1Kbps	**5Mbps**	**10Mbps**
Average Bandwidth (Mbps)	0.001	5.36	10.70
Maximum Bandwidth (Mbps)	0.002	10.67	13.84
Minimum Bandwidth (Mbps)	0.001	5.34	9.85
Bandwidth Standard Deviation	0.0001	0.31	0.20
Average Delay (ms)	1.56	0.06	0.06
Maximum Delay (ms)	5.32	0.79	0.60
Minimum Delay (ms)	0	0.02	0.02
Delay Standard Deviation	0.97	0.06	0.06
Average Packet Loss (%)	99.99	73.11	46.47
Maximum Packet Loss (%)	99.99	73.47	46.94
Minimum Packet Loss (%)	99.99	17.24	30.76
Packet Loss Standard Deviation	0.0001	3.24	0.98

Table 8.16: TP-Link Scenario 4 - Summary

8.4.6.2 Mininet

The overall performance of OpenFlow rate limiting function as well as Mininet was very stable although at higher bandwidths the limit was not precise. Apart from a pick at the initial second of the experiments which is caused by rate limit initialisation, the rest of the experiment was stable. At 1Kbps rate limit run, the average bandwidth achieved was exactly 1Kbps. The standard deviation proves this as well (Table 8.17). Delay was about 0.82ms (Table 8.17) whereas the load on the CPU cores was not evenly arranged. The network I/O was very stable although it has a drop at the 65th second. From the network I/O it was clear that

the amount of read data was much more than the write which confirms that the incoming traffic is more than the traffic resulting after the 1Kbps rate limit was used.

On the other hand the results are not so stable when it comes to 25Mbps and 50Mbps rate limits. At 25Mbps the average bandwidth achieved was 26.81Mbps with a standard deviation of 1.54 whereas at 50Mbps the bandwidth achieved was 53.59Mbps with a standard deviation of 2.35 (Table 8.17). This shows that at higher bandwidths the rate limiting was not accurate enough, although it can actually limit the rate, it did not produce the precision provided at 1Kbps. This cannot be caused by the CPU since at higher limits, the processes are spread to the two cores evenly. Finally, Network I/O was once again very stable for both read and write. (Appendix A, Figures A.31-A.33)

	1Kbps	**25Mbps**	**50Mbps**
Average Bandwidth (Mbps)	0.001	26.81	53.59
Maximum Bandwidth (Mbps)	0.001	53.43	94.13
Minimum Bandwidth (Mbps)	0.001	26.72	52.18
Bandwidth Standard Deviation	6.52×10^{-19}	1.54	2.35
Average Delay (ms)	0.82	0.01	0.01
Maximum Delay (ms)	2.09	0.03	0.04
Minimum Delay (ms)	0	0.002	0.001
Delay Standard Deviation	0.34	0.004	0.01
Average Core 1 Usage (%)	48.77	52.20	54.39
Maximum Core 1 Usage (%)	100	62.10	100
Minimum Core 1 Usage (%)	19	18.60	15.40
Core 1 Usage Standard Deviation	31.81	4.43	9.89
Average Core 2 Usage (%)	79.22	52.72	54.67
Maximum Core 2 Usage (%)	100	64.90	100
Minimum Core 2 Usage (%)	10.20	20.60	14.20
Core 2 Usage Standard Deviation	32.10	4.22	9.79

Table 8.17: Mininet Scenario 4 - Summary

8.4.6.3 HP-Procurve

For HP-Procurve, scenario 4 provided some very interesting results. It was evident from the results obtained that when the rate limit is at 1Kbps, the average bandwidth achieved is 100Kbps in both RJ-45 10/100 and RJ-45 10/100/1000 ports. It is suspected that this is not a switch problem but it is the packet length that is affecting the limit. Although further experimentation using smaller packet lengths will clear this out, it is not recommended to go beyond the standard packet length for testing because in a real life situation, all of the hosts will most probably be sending standard packet length packets. On the other hand for both 50Mbps and 500Mbps experiment, the bandwidth was very stable at the rate limit specified.

Looking at the delay for 1Kbps limit, the results were not identical for both port types. For RJ-45 10/100 ports, the delay starts at 0.005μs and ends up at 0.0015μs. For RJ-45 10/100/1000 ports the delay was stable throughout the experiment at 0.05μs. Delay for 50Mbps limit on 10/100 ports was stable at 0.06μs whereas at 500Mbps limit for RJ-45 10/100/1000 ports it drops slightly at 0.048μs. (Appendix A, Figures A.34-A.35)

8.4.7 Scenario 5 - TCP Bandwidth

Scenario 5 tested the ability of the platforms to handle TCP traffic. Furthermore, it detected the maximum performance they can achieve with TCP traffic.

8.4.7.1 TP-Link OpenWrt

The findings of scenario 5 proved that TCP bandwidth was unstable. The bandwidth was changing throughout the experiment within the range of 13 to 26Mbps. The average bandwidth for both the client and the server was 19.9Mbps with a standard deviation of 3.9 as shown in Table 8.18. CPU usage was also unstable with an average of 58.6% and a standard deviation of 7.6. (Appendix A, Figure A.36)

	Client	Server
Average Bandwidth (Mbps)	19.90	19.89
Maximum Bandwidth (Mbps)	26.45	26.21
Minimum Bandwidth (Mbps)	13.63	14.36
Bandwidth Standard Deviation	3.88	3.90
Average CPU Usage (%)	58.59	
Maximum CPU Usage (%)	83	
Minimum CPU Usage (%)	25	
CPU Usage Standard Deviation	7.58	

Table 8.18: TP-Link Scenario 5 - Summary

8.4.7.2 Mininet

As expected, with TCP traffic the bandwidth was not as stable as with UDP traffic. The bandwidth reached 93Mbps with a standard deviation of 0.37 for the server and 0.59 for the client (Table 8.19), while CPU cores had the same effect as in scenario 1. The first CPU core was overloaded and the remaining processes were transferred to core 2. Both read and write network I/O as well as the active RAM remained stable throughout the duration of the experiment. Hard disk was slightly busy, presumably due to the software used to monitor the system. The overall performance of TCP traffic can be considered as good, even though Mininet was not able to arrange the processes equally to the two CPU cores. It is suspected that if Mininet was able to arrange the processes equally to the two cores, then higher bandwidth would have been achieved and the overall stability of the experiment would be better. (Appendix A, Figures A.37-A.39)

8.4.7.3 HP-Procurve

Scenario 5 results were at an acceptable level if they are compared with Mininet and TP-Link results. Bandwidth for RJ-45 10/100 ports reached 87Mbps, which was 13Mbps less than with UDP traffic. The same happened with the RJ-45 10/100/1000 ports experiment, were the bandwidth reached 870Mbps. (Appendix A, Figure A.40)

	Server	Client
Average Bandwidth (Mbps)	93.15	93.16
Maximum Bandwidth (Mbps)	94.24	94.37
Minimum Bandwidth (Mbps)	89.72	89.13
Bandwidth Standard Deviation	0.37	0.59

	CPU 1	CPU 2
Average Usage (%)	95.52	9.56
Maximum Usage (%)	100	100
Minimum Usage (%)	3	2
Usage Standard Deviation	18.72	16.39

	Read	Write
Network I/O Total (KB)	3626085	36254834
Network I/O Average (KB)	12086.95	12084.90
Network I/O Maximum (KB)	12272	12280.80
Network I/O Minimum (KB)	0	0
Network I/O Standard Deviation	865.56	861.51

Table 8.19: Mininet Scenario 5 - Summary

8.4.8 Scenario 6 - TCP and UDP Bandwidth

The objective of scenario 6 was to find out if the platforms can handle both TCP and UDP traffic simultaneously. For HP-Procurve, Host *h1* was sending both TCP and UDP traffic to host *h2*. The experiment was repeated twice, the first time using RJ-45 10/100 ports and the second time using RJ-45 10/100/1000 ports. For TP-Link, the UDP traffic was predefined at 20Mbps, whereas for Mininet the topology was slightly modified. As shown in Figure 8.2 the topology in this experiment included four hosts a switch and a controller. The reason for the inclusion of four hosts was to enable the use of two of them for UDP and two for TCP. Host *h1* was sending TCP traffic to host *h2* via switch *s1* whereas host *h3* was sending 100Mbps UDP traffic to host *h4* via switch *s1*.

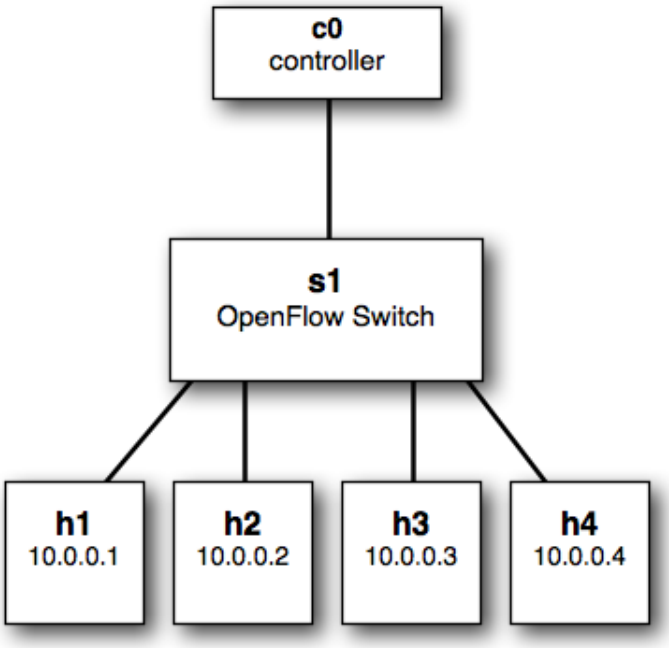


Figure 8.2: Mininet Four Hosts Topology

8.4.8.1 TP-Link OpenWrt

Scenario 6 proved that TCP traffic bandwidth is much more unstable than UDP traffic. On average, TCP bandwidth reached 18Mbps with a standard deviation of 0.6 for the client and 0.4 for the server. On the other hand, UDP bandwidth reached 20Mbps with a standard deviation of 0.006 for the client and 0.12 for the server. Delay for UDP traffic reached 0.3ms with a packet loss of 0.44%. CPU usage on the however, was very high with an average of 84.5% and a standard deviation of 4.22. (Table 8.20, Appendix A Figures A.41-A.42)

8.4.8.2 Mininet

Scenario 6 proved that the maximum traffic Mininet can handle is of the order of 110Mbps due to the fact that while the UDP client was sending 100Mbps, the receiver was receiving just 55Mbps. The remaining 55Mbps were used by the TCP traffic (Table 8.21). After 200 seconds the experimental results became slightly unstable. The bandwidth became slightly unstable until it had a big UDP bandwidth drop at around 260 seconds. Both CPU cores were working at 100% for 15 seconds at that point, and the network I/O became unstable as well. Active

	TCP		UDP	
	Client	**Server**	**Client**	**Server**
Average Bandwidth (Mbps)	18.17	18.17	20	19.91
Maximum Bandwidth (Mbps)	20.97	20.61	20.00	20.24
Minimum Bandwidth (Mbps)	14.68	14.99	19.99	19.49
Bandwidth Standard Deviation	0.60	0.42	0.01	0.12
Average Delay (ms)			0.30	
Maximum Delay (ms)			0.73	
Minimum Delay (ms)			0.17	
Delay Standard Deviation			0.10	
Packet Loss (%)			0.440	
Average CPU Usage (%)		84.54		
Maximum CPU Usage (%)		96		
Minimum CPU Usage (%)		68		
CPU Usage Standard Deviation		4.22		

Table 8.20: TP-Link Scenario 6 - Summary

RAM and hard disk show again an increase, especially in the case of hard disk which reached 100% activity. Delay on the other hand remained unaffected. This increase was presumably caused by the operating system and not by Mininet. This proves that Mininet is highly dependent on the operating system. A good practise would be to repeat these experiments using a real-time Linux kernel. (Appendix A, Figures A.43-A.46)

8.4.8.3 HP-Procurve

The overall bandwidth decreased, with all the flows achieving an average of 47Mbps. UDP delay can be considered very unstable if it is compared with the first scenario results. The average delay was at 0.0008μs with variations that reached 0.0015μs. The average bandwidth of the second run was at 470Mbps, whereas delay again showed some variations with an average of 0.0027μs. (Appendix A, Figures A.47-A.48)

	TCP Server	TCP Client
Average Bandwidth (Mbps)	55.24	55.25
Maximum Bandwidth (Mbps)	74.14	74.45
Minimum Bandwidth (Mbps)	49.99	50.33
Bandwidth Standard Deviation	1.97	2.05

	UDP Server	UDP Client
Average Bandwidth (Mbps)	55.80	100.51
Maximum Bandwidth (Mbps)	74.36	100.54
Minimum Bandwidth (Mbps)	8.26	100.36
Bandwidth Standard Deviation	3.22	0.01

	CPU 1	CPU 2
Average Usage (%)	96.07	25.62
Maximum Usage (%)	100	100
Minimum Usage (%)	17.20	10.30
Usage Standard Deviation	16.75	24.71

	Read	Write
Network I/O Total (KB)	5923036	4254140
Network I/O Average (KB)	19743.45	14180.50
Network I/O Maximum (KB)	19950.20	14448.30
Network I/O Minimum (KB)	10649.30	8711.30
Network I/O Standard Deviation	664.72	457.94

	Delay
Average Delay (ms)	0.04
Maximum Delay (ms)	0.10
Minimum Delay (ms)	0.03
Delay Standard Deviation	0.01

Table 8.21: Mininet Scenario 6 - Summary

8.5 Summary and Discussions

Overall, the experimentation with the three different platforms gave some very useful results which can be used in future experimentation with OpenFlow platforms. In the case of TP-Link router with OpenWrt firmware it proved that a router could not perform at the level of a switch even if it uses a custom firmware. In the case of Mininet it proved that the platform is highly depended on the CPU and OS performance and might not be a suitable platform for every situation. When it comes to the HP-Procurve switch it has proved that it is a solid candidate for OpenFlow experimentation as well as implementation, due to the very solid results it has given to all of the experiments.

More specifically, the following conclusions can be given:

1. The TP-Link OpenWrt router cannot be used to run performance metrics, but it can be used as prototyping hardware since its firmware allows full customisation as well as addition of new functionality. The best bandwidth to run prototyping experiments is at 20 Mbps as shown in the metrics of scenarios 1.a and 1.b.

   Mininet on the other hand can be used for both prototyping and performance metrics, with 100Mbps being the best bandwidth to experiment with as concluded from scenarios 1.a and 1.b.

   HP-Procurve is the best candidate for performance metrics since it is able to reach the maximum performance supported by the hardware. RJ-45 10/100 ports reached the maximum supported bandwidth of 100Mbps with an almost negligible delay of 0.0004μs. RJ-45 10/100/1000 also reached the maximum supported bandwidth of 1000Mbps (1Gbps) with a delay of 0.0048μs which can be considered as excellent. On the other hand it is the worst candidate for prototyping since the firmware is not open and it cannot be edited.

2. The TP-Link OpenWrt router is capable of handling more than one UDP flows coming from the same source. The only drawback is that since one of the ports is used by the controller, only three ports remain for experimenta-

tion. The TP-Link OpenWrt also handles bidirectional traffic although its bandwidth faces some minor stability issues.

HP-Procurve switch achieved the same stability for any number of streams. All of the streams shared the maximum possible bandwidth equally. Delay showed some variations but those variations are minimal. The number of ports is not as limited as the TP-Link OpenWrt router but is not unlimited either.

Mininet on the other hand might not have the bandwidth capabilities HP-Procurve has but is not restricted by the number of ports since any number of ports can be created.

3. OpenFlow's rate limiting function, implemented in OpenFlow version 1.3, is functioning at a very efficient level in the TP-Link OpenWrt if the low CPU processing power of the router is considered. Delay was also increased at low rate limits, which indicates the level of activity of the CPU.

   Mininet showed the same efficiency and the same increase in delay as the TP-Link OpenWrt. Even though at low rate limits CPU cores had the tendency to be fully active, the resulting bandwidth was not affected.

   HP-Procurve has its own rate limit implementation outside OpenFlow. Its rate limit it perfectly smooth, even though the limit rate of 1Kbps resulted in 100Kbps, this is suspected to be a problem caused by the "packet length" and not by the switch.

4. The TP-Link OpenWrt is unstable with TCP traffic as shown by the bandwidth instabilities whereas the CPU usage was only around 60%.

   Mininet proved that it can be used to emulate data centre environments, even though it gave some unexpected results in delay performance. This is due to the fact that "global" flow performed better than "local" flows.

   HP-Procurve switch can handle TCP traffic at an acceptable level. Even though it does not reach the maximum bandwidth supported, the value reached is the one expected for TCP traffic.

5. In the mixture of both TCP and UDP, the TP-Link OpenWrt is not affected. TCP traffic showed the same instabilities whereas, UDP traffic was unaffected and reached an average of 20Mbps with negligible standard deviation.

   Mininet showed the same consistency as the TP-Link OpenWrt. HP-Procurve was also unaffected by the different types of traffic.

6. HP-Procurve switch achieved the highest bandwidth with the highest possible stability and performance. It also has the lowest delay. In all of the experiments, HP-Procurve switch was the only one that showed no packet losses. Which means that even the actual capacity of the switch was not reached.

7. Comparing delay with CPU performance for the TP-Link OpenWrt & Mininet, it is concluded that delay is highly depended on CPU performance. This leads to the conclusion that HP-Procurve can achieve a very high CPU performance and efficiency due to the fact that delay is not affected even at very high loads.

8. Mininet did not emulate HP-Procurve switch results, even if a high-end server was used. Mininet's and TP-Link router's results are almost identical. The only difference is that Mininet has a better CPU availability and can achieve slightly better results.

9. At higher bandwidths (above 130Mbps), Mininet cannot distribute the CPU load equally to all the system processors.

Chapter 9

Conclusions and Future Work

This thesis focuses on the research of SDN performance and the proposal of *(a)* a performance enhancement algorithm OFPE that uses dynamic flow installation and management techniques, *(b)* an extension to the proposed OFPE algorithm, *(c)* the proposal of a novel placement algorithm for distributed Mininet implementations and *(d)* the proposal of a series of experiments for evaluation of SDN experimental platforms. In this chapter, a summary of the work completed is given in Section 9.1, whereas in Section 9.2 potential improvements as well as future applications is presented.

9.1 Conclusions

SDN is an emerging field in the area of computer networks which promises to tackle some of the most challenging trends of modern networking. These challenges can be summarised into four areas (a) changing of traffic patterns, (b) rise of cloud services, (c) IT consumerization and (d) bandwidth exponential growth. Being an emerging field, SDN faces a lot of challenges due to the fact that it changes the traditional way of networking. Up to now, most of the problems that SDN faces are in the area of performance. This is due to the fact that, in order for some of the well-used networking techniques to have the full benefits of using an SDN, they have to be re-implemented with SDN paradigms in mind. This re-implementation of previously used methods results in a long process which leads to performance issues.

These performance issues include:

(a) The increase of delay due to the extra round trip time between the switch and the controller, as well as the processing time needed by the controller.

(b) The packet losses caused by the switch buffer getting full due to controller processing delay, as well as the capacity of the link between the switch and the controller.

(c) The out-of-order packets problem which arises from the fact that not only the initial packet of a flow visits the controller.

On the other hand, all of the issues arise from the introduction of a controller in the system. This can cause several problems, such as a single point of failure if the controller or the link between the switch and the controller fails as well as huge delays or packet losses if the controller's state is not healthy (i.e. overloaded).

This thesis proposes a new performance enhancement algorithm OFPE that improves the overall SDN performance by decreasing delay, packet loss and out-of-order packets, and keeping the controller in a non-overloaded state through the use of dynamic flow installation and management techniques.

In Chapter 2 we provide a comprehensive review of SDN, starting from schemes that existed prior to SDN which helped in better understanding some of the networking problems as well as the solutions that needed to be developed. Then we go through the changes SDN model brought to traditional networking paradigms, as well as present the individual components needed in order for an SDN model to be implemented. We present OpenFlow, the most widely used SDN protocol showing how it has evolved over the years in order to accommodate networking needs. Finally, we present some real use cases of SDN which indicate that the model has been maturing over the years and that several applications in the real world have benefited from its use.

In Chapter 3 we introduce the proposed OpenFlow Performance Enhancement Algorithm OFPE which uses dynamic flow installation and management techniques. OFPE is implemented on the SDN controller and provides better performance by reducing the packet loss and delay as well as the number of out-of-order

packets. In addition, it preserves controller stability as well as it prevents the controller from overloading. Finally, it has topology awareness and route creation functions that allow it to be more robust and able to handle a variety of topologies. We analyse each aspect of OFPE, giving information about each individual module that is used in order for the algorithm to work. In addition, we perform several experimental scenarios in order to test the performance as well as the benefits OFPE can bring to the SDN model. With the use of Mininet, in some of our experiments, the proposed OFPE algorithm has managed to decrease delay by up to 92.56%, packet loss by up to 55.32% and the number of out-of-order packets by up to 69.44%. In Chapter 4 we present an extension of our OFPE algorithm, namely OFPEX, which uses the packets inter-arrival time in order to calculate and predict traffic patterns and manage them accordingly. Due to limitations of statistics in the SDN model which we discuss in Chapter 4, in some calculations OFPEX performs an estimation of the result. In order to tests its performance, a series of experiments have been performed, with the results indicating that it can increase the performance of SDN. Some of the most noticeable results are the decrease of packet loss by 17.71%, delay by 16.19% and the number of out-of-order packets by 20.54%. In addition, it has managed to decrease the number of flow table rules by up to 16.82% which is very important due to the flow table size limitations SDN faces as we discussed in Chapter 4.

In Chapter 5 we discussed the limitations that distributed Mininet implementations face and especially the way they spread out the experimental topology. We proposed a new placement algorithm that assigns weights to both the experimental topology components as well as the physical machines that are used to run the experiment on. With the use of weights, and by making as less cross-server links as possible, the proposed algorithm was able to spread the load evenly, and utilise each physical machine equally. With the use of MaxiNet, we have tested the proposed placement algorithm which was able to decrease packet losses by 86% as well as delay by 68%.

In Chapter 6 we used both of our proposed algorithm, namely the OpenFlow Performance Enhancement Algorithm and the Distributed Mininet Placement Algorithm in order to examine the create a large topology and test the performance

of both. We designed some scenarios that were creating bottlenecks in the network as well as stressing some individual components of the topology. The performance tests indicated decrease in packet loss by up to 31.77%, delay by up to 30.30% and the number of out-of-order packets by up to 44.99%

In Chapters 7 and 8 we presented and performed a series of performance experiments on several experimental pieces of equipment. This included Mininet emulator as well as two testbed switches, the high-end HP-Procurve switch as well as the low-end TP-Link router with the OpenWrt custom firmware. Initially, in Chapter 7 we presented a series of experiments that can be performed on SDN experimental equipment in order to find out their strengths and weaknesses. We tested Mininet and concluded that Mininet favours more RAM if it is available, the number of packet losses increase as the number of links included in the packet's path increase and finally, that load balancing between CPU cores becomes more efficient as the number of nodes in the topology increases. Finally, in Chapter 8 we performed a series of experiments on Mininet, HP-Procurve and the TP-Link router, which indicated their strengths and weaknesses in different aspects. The results indicated that the TP-Link router lacks in performance whereas Mininet is highly depended on the machine that is used to run it. HP-Procurve switch indicated high stability and robustness, but it has no room for prototyping since it is a closed-source system.

9.2 Future Work

There is plenty of room for improvement both for the proposed work in this thesis as well as *(a)* the SDN model in general, *(b)* the way switches interact with controllers in the SDN model. In addition, the proposed work can also be applied as a concept to several other areas. In this Section, a list of potential extensions and recommendations for future work is presented.

(a) The current implementation of the proposed performance enhancement algorithm OFPE works only when one controller is present in the system. The algorithm can be improved in order to work with distributed implementations of SDN controllers. This will remove some workload from the controller

and make the whole algorithm more efficient.

(b) With the current state of the SDN model, the statistics gathered by the controller are limited and cause extra load on the controller-to-switch link. An improvement in the area of statistics (i.e. extra link with more statistics capabilities) will not only help the controller into more accurate and faster predictions, but it will also help alleviate the extra traffic in the controller-to-switch link caused by the statistics traffic.

(c) As proposed in some other SDN related works, giving back to the switch some functions of the control plane may benefit SDN. The proposed performance enhancement algorithm OFPE will be benefited with such an approach since the calculations would be able to take place in the switch. Therefore, the controller-to-switch link will carry only the calculation results and not all the information needed. This will result in less load both on the controller-to-switch link as well as the controller, resulting in better stability of the whole system.

(d) Mininet will be hugely benefited from an in-house traffic generation mechanism. This will not only allow researchers to create their traffic directly in Mininet, it will also alleviate the need for external traffic generation tools which will decrease the number of resources needed by external tools.

(e) Using the proposed future work of point *(d)*, the algorithm proposed in Chapter 5 will be hugely benefited since it will be easier to calculate the amount of traffic present in different components of the topology, therefore it will be easier to distribute the components in the correct worker.

References

[1] Cisco Visual Networking. Cisco global cloud index: Forecast and methodology, 2011-2016. White Paper, 2012.

[2] Cisco Visual Networking. Cisco global cloud index: Forecast and methodology, 2013-2018. White Paper, 2014.

[3] OpenFlow Consortium. OpenFlow Switch Specification Version 1.0.0 (Wire Protocol 0x01). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`, December 2009. Accessed: 07-10-2015.

[4] O. M. E. Committee. Software-defined Networking: The New Norm for Networks. Open Networking Foundation, 2012.

[5] K. Yap, M. Kobayashi, R. Sherwood, T. Huang, M. Chan, N. Handigol, and N. McKeown. Openroads: empowering research in mobile networks. *SIGCOMM Comput. Commun. Rev.*, 40(1):125–126, January 2010.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[7] ESG Brief. IBM and NEC Bring SDN/OpenFlow to Enterprise Data Center Networks, 2012.

[8] Cisco Visual Networking. Cisco global cloud index: Forecast and methodology, 2015-2020. White Paper, 2016.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in

campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[10] X. Xipeng and L.M. Ni. Internet QoS: a big picture. *Network, IEEE*, 13(2):8–18, 1999.

[11] ITU-T. Terms and definitions related to quality of service and network performance including dependability. Recommendation E.800, International Telecommunication Union, Geneva, 2009.

[12] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009.

[13] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[14] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, 1998.

[15] L. Yang, R. Dantu, T. Anderson, and Gopal R. Forwarding and Control Element Separation (ForCES) Framework. RFC 3746 (Informational), April 2004.

[16] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an ip services protocol. RFC 3549 (Informational), July 2003.

[17] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[18] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFCs 6286, 6608, 6793.

[19] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[20] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[21] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-wide decision making: Toward a wafer-thin control plane. In *HotNets III*, 2004.

[22] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[23] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[24] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: enabling controlled networking. *SIGCOMM Comput. Commun. Rev.*, 33(1):65–70, January 2003.

[25] J. Luo, J. Pettit, M. Casado, J. Lockwood, and N. McKeown. Prototyping fast, simple, secure switches for ethane. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*, HOTI '07, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.

[26] H. Yan, D. A. Maltz, T.S. Eugene Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: a 4d network control plane. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 27–27, Berkeley, CA, USA, 2007. USENIX Association.

[27] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[28] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[29] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. S. Eugene Ng. Maestro: A clean-slate system for orchestrating network control components, 2008.

[30] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. S. Eugene Ng. The preliminary design and implementation of the maestro network control platform, 2008.

[31] A. Tootoonchian and Y. Ganjali. Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[32] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[33] M. Yu, J. Rexford, M.J. Freedman, and J. Wang. Scalable flow-based networking with difane. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[34] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating systems support for planetary-scale network services. In *NSDI*, volume 4, pages 19–19, 2004.

[35] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[36] NetFPGA: Programmable Networking Hardware. `http://netfpga.org`, 2012. Accessed: 07-10-2015.

[37] J.W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and L. Jianying. Netfpga–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, pages 160–161, 2007.

[38] G. Watson, N. McKeown, and M. Casado. Netfpga: A tool for network research and education. In *2nd workshop on Architectural Research using FPGA Platforms*. WARFP, 2006.

[39] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

[40] OpenFlow Consortium. OpenFlow Switch Specification Version 1.3.2 (Wire Protocol 0x04). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf`, April 2013. Accessed: 07-10-2015.

[41] OpenFlow Consortium. OpenFlow Switch Specification Version 1.2 (Wire Protocol 0x03). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf`, December 2011. Accessed: 07-10-2015.

[42] OpenFlow Consortium. OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`, June 2012. Accessed: 07-10-2015.

[43] OpenFlow Consortium. OpenFlow Switch Specification Version 1.4.0 (Wire Protocol 0x05). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, October 2013. Accessed: 07-10-2015.

[44] Open Networking Foundation Extensibility Working Group. https://www.opennetworking.org/images/stories/downloads/working-groups/charter-extensibility.pdf. Accessed: 07-10-2015.

[45] H. Berkowitz, E. Davies, S. Hares, P. Krishnaswamy, and M. Lepp. Terminology for Benchmarking BGP Device Convergence in the Control Plane. RFC 4098 (Informational), June 2005.

[46] G. Trotter. Terminology for Forwarding Information Base (FIB) based Router Performance. RFC 3222 (Informational), December 2001.

[47] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[48] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013.

[49] B. Pfaff and B. Davie. The open vswitch database management protocol. RFC 7047, RFC Editor, December 2013. http://www.rfc-editor.org/rfc/rfc7047.txt.

[50] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[51] M. Suñé, V. Alvarez, T. Jungel, U. Toseef, and K. Pentikousis. An openflow implementation for network processors. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 123–124. IEEE, 2014.

[52] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher. Opflex control protocol. RFC 7047, Internet Draft, April 2014. http://tools.ietf.org/html/draft-smith-opflex-00.

[53] Trema. http://trema.github.com/trema. Accessed: 07-10-2015.

[54] Beacon. https://openflow.stanford.edu/display/Beacon/Home. Accessed: 07-10-2015.

[55] SNAC. http://openflow.org/wp/snac. Accessed: 07-10-2015.

[56] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[57] Floodlight OpenFlow Controller. http://www.projectfloodlight.org, 2015. Accessed: 07-10-2015.

[58] Ryu. https://osrg.github.io/ryu. Accessed: 07-10-2015.

[59] Nodeflow controller. https://github.com/gaberger/NodeFLow, 2015. Accessed: 07-10-2015.

[60] H. Shimonishi, S. Ishii, Y. Chiba, T. Koide, M. Takahashi, Y. Takamiya, and L. Sun. Helios: Fully distributed openflow controller platform. In *Proceedings of the 9th GENI engineering conference (GEC9) Demo*, 2010.

[61] BigSwitch OpenFlow Controller. http://www.bigswitch.com/products/SDN-Controller, 2015. Accessed: 07-10-2015.

[62] B. Lee, S. Park, J. Shin, and S. Yang. Iris: the openflow-based recursive sdn controller. In *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, pages 1227–1231. IEEE, 2014.

[63] K. Phemius, M. Bouet, and J. Leguay. DISCO: Distributed Multi-domain SDN Controllers. *ArXiv e-prints*, August 2013.

[64] HP. Hp SDN controller architecture. Technical report, Hewlett-Packard Development Company, L.P., September 2013.

[65] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[66] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and W. Guohui. Meridian: an SDN platform for cloud network services. *Communications Magazine, IEEE*, 51(2):120–127, 2013.

[67] S. Dipjyoti. MuL OpenFlow controller. `http://sourceforge.net/projects/mul/`, 2013. Accessed: 07-10-2015.

[68] Juniper Networks. Opencontrail. `http://opencontrail.org/`, 2013. Accessed: 07-10-2015.

[69] NEC. Award-winning Software-defined Networking NEC ProgrammableFlow Networking Suite. `http://www.necam.com/docs/?id=67c33426-0a2b-4b87-9a7a-d3cecc14d26a`, September 2013. Accessed: 07-10-2015.

[70] F. Botelho, A. Bessani, F. Ramos, and P. Ferreira. On the design of practical fault-tolerant SDN controllers. In *Third European Workshop on Software Defined Networks*, pages –, 2014.

[71] M. Monaco, O. Michel, and E. Keller. Applying Operating System Principles to SDN Controller Design. In *Twelfth ACM Workshop on Hot Topics in Networks (HotNets-XII)*, College Park, MD, November 2013.

[72] S. Matsumoto, S. Hitz, and A. Perrig. Fleet: Defending SDNs from malicious administrators. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 103–108, New York, NY, USA, 2014. ACM.

[73] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network

virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, April 2014.

[74] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: an API for application control of SDNs. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, pages 327–338, New York, NY, USA, 2013. ACM.

[75] S. Seungwon, S. Yongjoo, L. Taekyung, L. Sangho, C. Jaewoong, P. Phillip, Y. Vinod, N. Jisung, and B. K. Brent. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014. *To appear.*

[76] K. Pentikousis, Y. Wang, and W. Hu. Mobileflow: Toward software-defined mobile networks. *IEEE Communications magazine*, 51(7):44–53, 2013.

[77] S. Racherla, D. Cain, S. Irwin, P. Ljungstrom, P. Patil, and A. Tarenzio. *Implementing IBM Software Defined Network for Virtual Environments.* IBM RedBooks, May 2014.

[78] S. Wang, C. Chou, and C. Yang. EstiNet OpenFlow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, 2013.

[79] D. Klein and M. Jarschel. An OpenFlow extension for the OMNeT++ INET framework. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, pages 322–329. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[80] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[81] A. Varga, R. Hornig, B. Seregi, L. Meszaros, and Z. Bojthe. INET Framework. `https://inet.omnetpp.org`. Accessed: 07-10-2015.

[82] 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*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[83] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Mininet Performance Fidelity Benchmarks. `http://hci.stanford.edu/cstr/reports/2012-02.pdf`. Accessed: 07-10-2015.

[84] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Comput. Commun. Rev.*, 39(4):39–50, August 2009.

[85] B. Heller, D. Erickson, N. McKeown, R. Griffith, I. Ganichev, S. Whyte, K. Zarifis, D. Moon, S. Shenker, and S. Stuart. Ripcord: a modular platform for data center networking. *SIGCOMM Comput. Commun. Rev.*, 40(4):457–458, August 2010.

[86] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying nox to the datacenter. In *In 8th ACM Workshop on Hot Topics in Networking (Hotnets)*, New York City, NY, USA, October 2009.

[87] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 8. ACM, 2011.

[88] A. Das, C. Lumezanu, Y. Zhang, V. K. Singh, G. Jiang, and C. Yu. Transparent and flexible network management for big data processing in the cloud. In *HotCloud*, 2013.

[89] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Jo-

hari, N. McKeown, and G. Parulkar. Carving research slices out of your production networks with openflow. *SIGCOMM Comput. Commun. Rev.*, 40(1):129–130, January 2010.

[90] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *Technical Report Openflow-tr-2009-1*, 2009.

[91] N. Bastin and A. Al-Shabibi. Flowvisor. `http://https://openflow.stanford.edu/display/DOCS/Flowvisor`, December 2012. Accessed: 07-10-2015.

[92] A. Mohammad, R. Sivasankar, R. Barath, H. Nelson, and V. Amin. Hedera: Dynamic flow scheduling for data center networks. In *In Proc. of Networked Systems Design and Implementation (NSDI) Symposium*, 2010.

[93] J. Naous, R. Stutsman, D. Mazieres, N. McKeown, and N. Zeldovich. Delegating network security with more information. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, Barcelona, Spain, August 2009. ACM Press.

[94] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks. In *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*, pages 1–8. IEEE, 2012.

[95] M. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. Policycop: An autonomic qos policy enforcement framework for software defined networks. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*, pages 1–7. IEEE, 2013.

[96] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. Opentm: traffic matrix estimator for openflow networks. In *Proceedings of the 11th international conference on Passive and active measurement*, PAM'10, pages 201–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[97] K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown. The stanford openroads deployment. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, WINTECH '09, pages 59–66, New York, NY, USA, 2009. ACM.

[98] K. Yap, T. Huang, M. Kobayashi, M. Chan, R. Sherwood, G. Parulkar, and N. McKeown. Lossless handover with n-casting between wifi-wimax on openroads. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, MobiCom 2009, Beijing, China, September 2009. ACM, Sigmobile.

[99] L. Haizhuo, S. Lujing, F. Yuntao, and G. Suiming. Apply embedded openflow mpls technology on wireless openflow, openroads. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 916–919, 2012.

[100] K. Yap, R. Sherwood, M. Kobayashi, T. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, VISA '10, pages 25–32, New York, NY, USA, 2010. ACM.

[101] J. Schulz-Zander, N. Sarrar, and S. Schmid. Aeroflux: A near-sighted controller architecture for software-defined wireless networks. In *ONS*, 2014.

[102] J. Schulz-Zander, N. Sarrar, and S. Schmid. Towards a scalable and near-sighted control plane architecture for wifi sdns. In *HotSDN*, pages 217–218, 2014.

[103] J. Schulz-Zander, P. L. Suresh, N. Sarrar, A. Feldmann, T. Hühn, and R. Merz. Programmatic orchestration of wifi networks. In *USENIX Annual Technical Conference*, pages 347–358, 2014.

[104] M. Yang, Y. Li, D. Jin, L. Su, S. Ma, and L. Zeng. Openran: a software-defined ran architecture via virtualization. *ACM SIGCOMM computer communication review*, 43(4):549–550, 2013.

[105] J. Kempf and P. Yegani. Openran: A new architecture for mobile wireless internet radio access networks. *IEEE Communications Magazine*, 40(5):118–123, 2002.

[106] A. Gudipati, D. Perry, L. E. Li, and S. Katti. Softran: Software defined radio access network. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 25–30. ACM, 2013.

[107] A.K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pages 11–18, New York, NY, USA, 2009. ACM.

[108] J. R. Ballard, I. Rae, and A. Akella. Extensible and scalable network monitoring using opensafe. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[109] J. Matias, J. Garay, A. Mendiola, N. Toledo, and E. Jacob. Flownac: Flow-based network access control. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 79–84. IEEE, 2014.

[110] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 121–126. ACM, 2012.

[111] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS*, 2013.

[112] U. Hoelzle. Keynote speech: Openflow @ Google. In *Open Networking Summit Conference 2012*, April 2012.

[113] Z. Min, M. Dusi, W. John, and C. Changjia. Analysis of udp traffic usage on internet backbone links. In *Applications and the Internet, 2009. SAINT '09. Ninth Annual International Symposium on*, pages 280–281, July 2009.

[114] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[115] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. *ACM SIGCOMM Computer Communication Review*, 41(4):254–265, 2011.

[116] POX Controller. `http://www.noxrepo.org/pox/about-pox/`. Accessed: 13-10-2014.

[117] Open VSwitch. `http://www.openvswitch.org`. Accessed: 07-10-2015.

[118] NEC ProgrammableFlow UNIVERGE PF5820. `http://www.necam.com/Docs/?id=ba0dadc4-f253-4a8a-b27a-a791378f9acf`, 2014. Accessed: 07-10-2015.

[119] Brocade MLX 24-Port 10 GBE Switch. `http://www.brocade.com/en/backend-content/pdf-page.html?/content/dam/common/documents/content-types/datasheet/brocade-mlx-24x10gbe-ds.pdf`, 2014. Accessed: 07-10-2015.

[120] Brocade MLX 20-Port 10 GBE Switch. `http://www.brocade.com/en/backend-content/pdf-page.html?/content/dam/common/documents/content-types/datasheet/brocade-mlx-20x10gbe-ds.pdf`, 2014. Accessed: 07-10-2015.

[121] NEC ProgrammableFlow PF5240 Switch. `http://www.necam.com/sdn/doc.cfm?t=PFlowPF5240Switch`, 2014. Accessed: 07-10-2015.

[122] Arista 7050 Series Switch. `http://www.arista.com/en/products/7050-series`, 2014. Accessed: 07-10-2015.

[123] Arista 7150 Series Switch. `http://www.arista.com/en/products/7150-series`, 2014. Accessed: 07-10-2015.

[124] Arista 7300 Series Switch. http://www.arista.com/en/products/7300-series, 2014. Accessed: 07-10-2015.

[125] Arista 7500 Series Switch. http://www.arista.com/en/products/7500-series, 2014. Accessed: 07-10-2015.

[126] A. Azzouni, N. Trang, R. Boutaba, and G. Pujolle. Limitations of openflow topology discovery protocol. *arXiv preprint arXiv:1705.00706*, 2017.

[127] HP Procurve 3500 Switch. https://h20195.www2.hpe.com/v2/getpdf.aspx/c04123356.pdf?ver=5. Accessed: 07-10-2015.

[128] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool. http://sourceforge.net/projects/iperf. Accessed: 07-10-2015.

[129] T. Benson, A. Akella, and D. A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[130] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208. ACM, 2009.

[131] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[132] S. Shirali-Shahreza and Y. Ganjali. Empowering software defined network controller with packet-level information. In *Communications Workshops (ICC), 2013 IEEE International Conference on*, pages 1335–1339. IEEE, 2013.

[133] A. Roy, M. F. Bari, M. F. Zhani, R. Ahmed, and R. Boutaba. DOT: distributed OpenFlow testbed. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 367–368. ACM, 2014.

[134] B. Lantz and B. O'Connor. A mininet-based virtual testbed for distributed sdn development. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 365–366. ACM, 2015.

[135] V. Antonenko and R. Smelyanskiy. Global network modelling based on mininet approach. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 145–146. ACM, 2013.

[136] J. Teixeira, G. Antichi, D. Adami, A. Del Chiaro, S. Giordano, and A. Santos. Datacenter in a box: Test your sdn cloud-datacenter controller at home. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 99–104. IEEE, 2013.

[137] P. Wette, M. Dräxler, and A. Schwabe. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[138] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.

[139] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[140] K. C. Webb, A. C. Snoeren, and K. Yocum. Topology switching for data center networks. *Hot-ICE*, 11, 2011.

[141] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[142] Ostinato network traffic generator and analyzer. `http://ostinato.org`. Accessed: 26-06-2017.

[143] M. Gupta, J. Sommers, and P. Barford. Fast, accurate simulation for SDN prototyping. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 31–36. ACM, 2013.

[144] T. Henderson, M. Lacage, G. Riley, M. Watrous, G. Carneiro, T. Pecorella, and others. Network Simulator 3. `https://www.nsnam.org`. Accessed: 07-10-2015.

[145] EstiNet Technologies Inc. EstiNet. `http://www.estinet.com`. Accessed: 07-10-2015.

[146] Pantou. `http://www.openflow.org/wk/index.php/Pantou`. Accessed: 07-10-2015.

[147] OpenWrt. `http://openwrt.org`. Accessed: 07-10-2015.

[148] Indigo Virtual Switch IVS. `https://github.com/floodlight/ivs`. Accessed: 07-10-2015.

[149] Broadcom. `http://www.broadcom.com`. Accessed: 07-10-2015.

[150] TP-LINK TL-WR1043ND. `http://uk.tp-link.com/products/details/TL-WR1043ND.html`. Accessed: 07-10-2015.

[151] Atheros. `http://www.atheros.com`. Accessed: 07-10-2015.

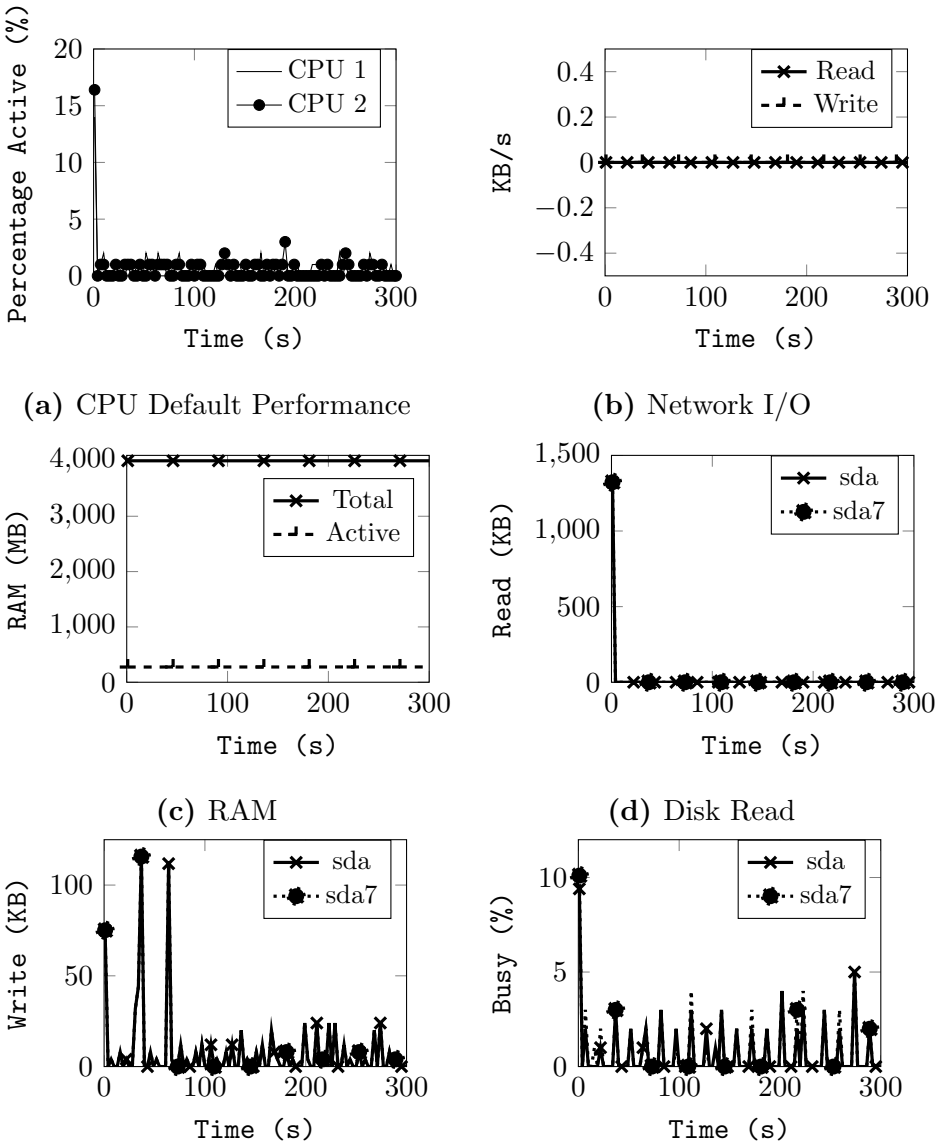
[152] Spirent TestCenter. `https://www.spirent.com/Products/TestCenter`. Accessed: 07-10-2015.

Appendices

Appendix A

OpenFlow Software & Hardware Performance Evaluation Figures

A.1 Mininet System Default Performance



(a) CPU Default Performance

(b) Network I/O

(c) RAM

(d) Disk Read

(e) Disk Write

(f) Disk Busy Percentage

Figure A.1: System Default Performance

A.2 Scenario 1.a - Bandwidth

A.2.1 TP-Link OpenWrt

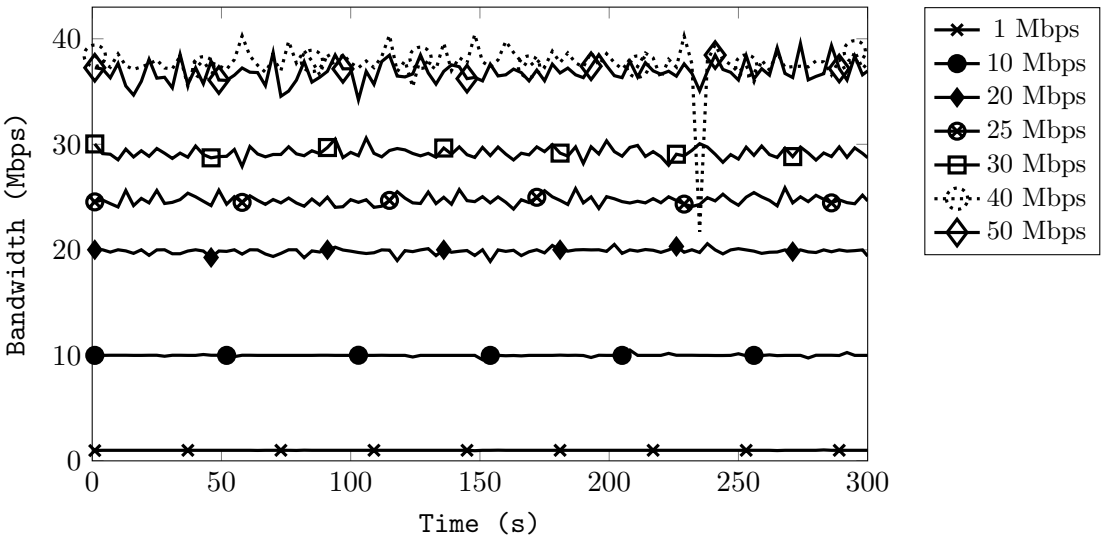
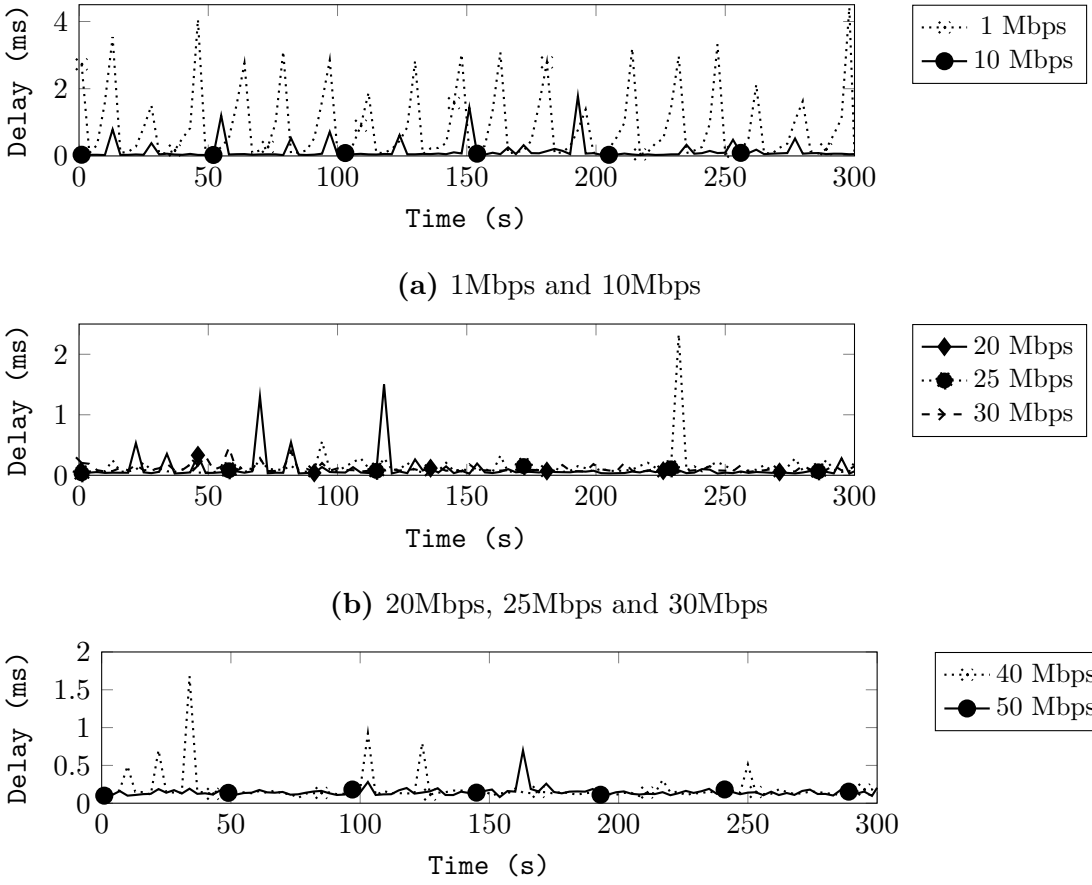


Figure A.2: TP-Link Scenario 1.a - Bandwidth

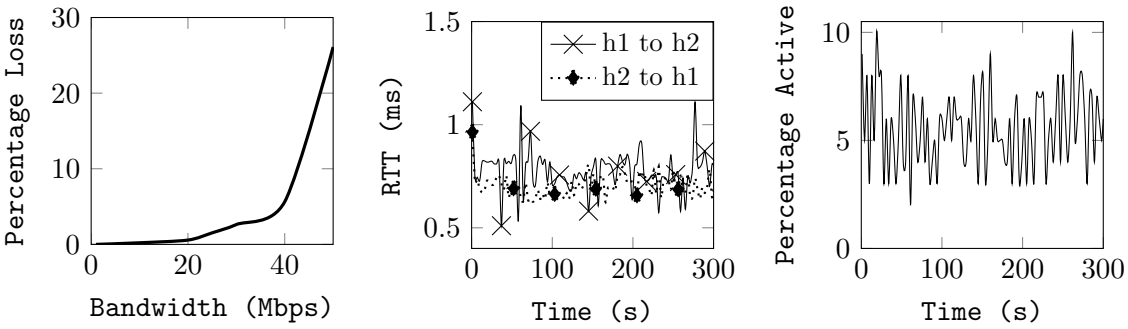


(a) 1Mbps and 10Mbps

(b) 20Mbps, 25Mbps and 30Mbps

(c) 40Mbps and 50Mbps

Figure A.3: TP-Link Scenario 1.a - Delay

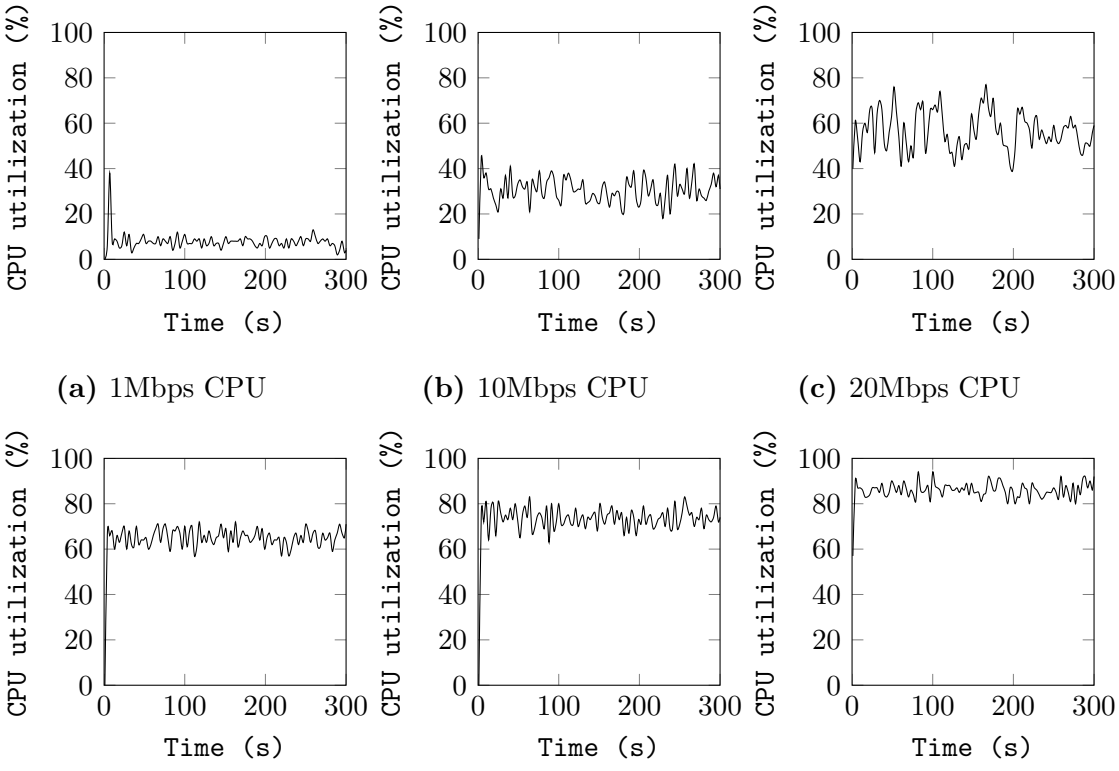


(a) Packet Loss %

(b) Latency-RTT

(c) CPU

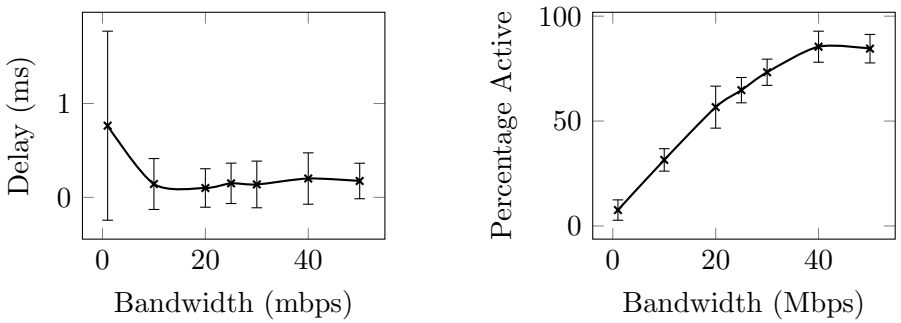
Figure A.4: TP-Link Scenario 1.a - Performance



(a) 1Mbps CPU **(b)** 10Mbps CPU **(c)** 20Mbps CPU

(d) 25Mbps CPU **(e)** 30Mbps CPU **(f)** 40Mbps CPU

Figure A.5: TP-Link Scenario 1.a - CPU Performance (% Active)



(a) Bandwidth Vs Delay **(b)** Bandwidth Vs CPU

Figure A.6: TP-Link Scenario 1.a - Comparisons

A.2.2 Mininet

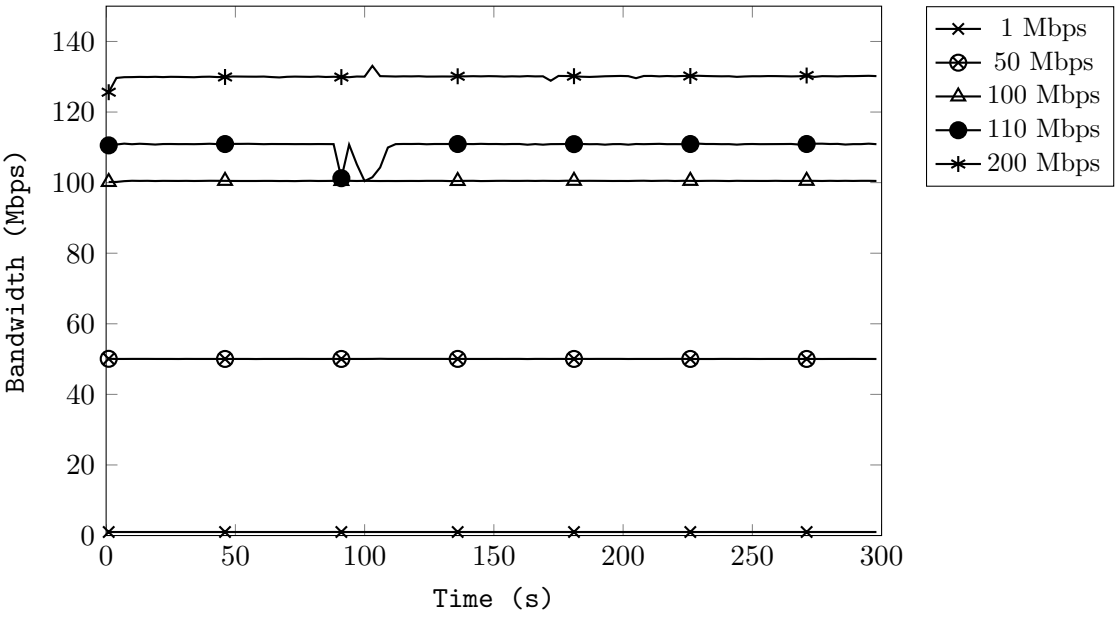


Figure A.7: Mininet Scenario 1.a - Bandwidth

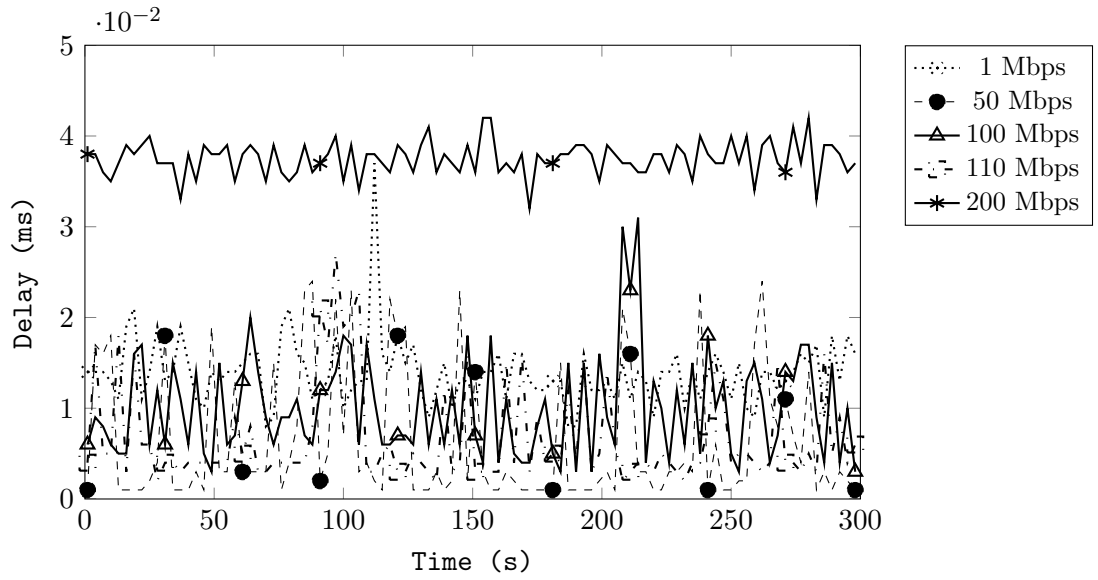
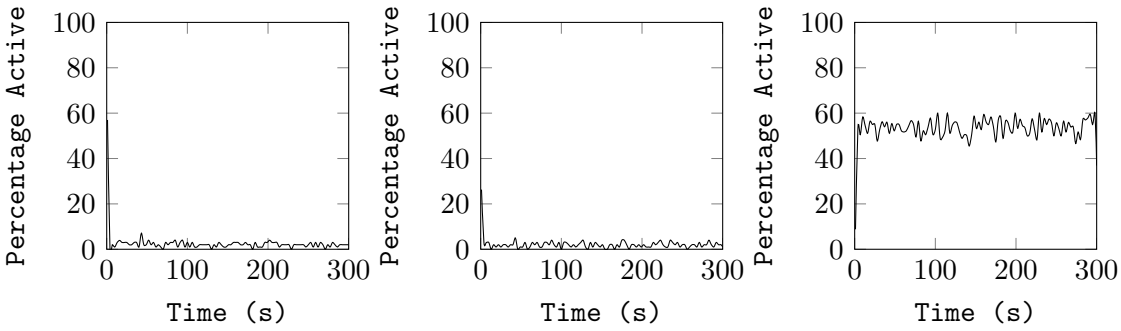


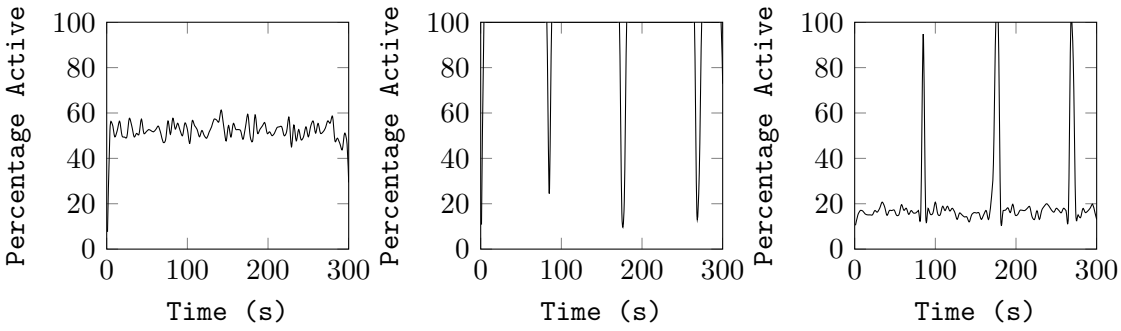
Figure A.8: Mininet Scenario 1.a - Delay



(a) 1Mbps First CPU Core

(b) 1Mbps Second CPU Core

(c) 100Mbps First CPU Core

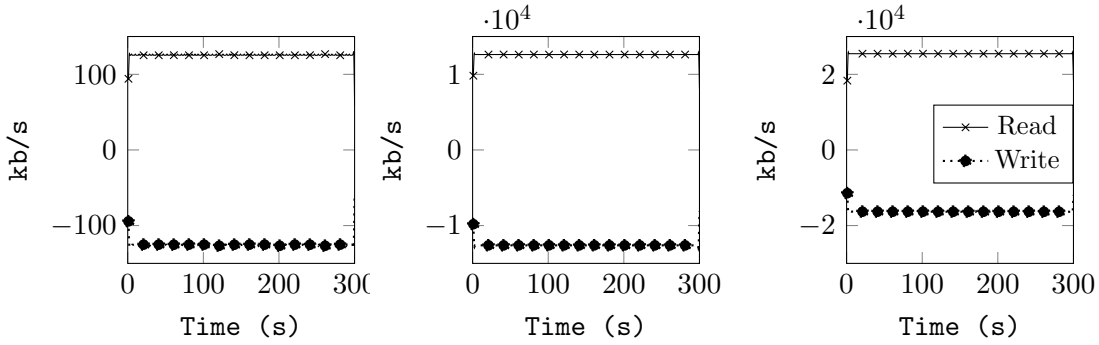


(d) 100Mbps Second CPU Core

(e) 200Mbps First CPU Core

(f) 200Mbps Second CPU Core

Figure A.9: Mininet Scenario 1.a - CPU Performance

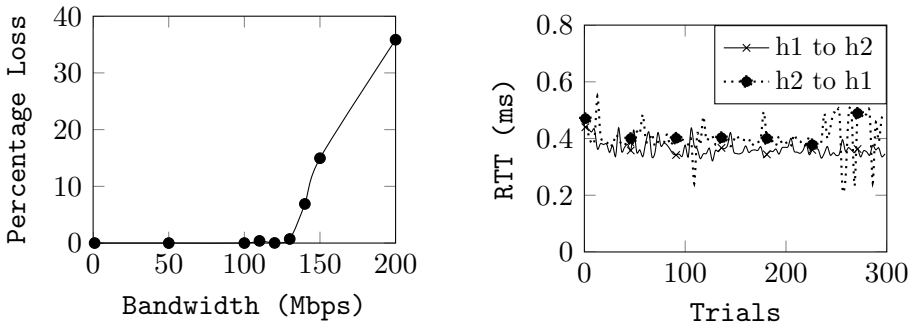


(a) 1Mbps

(b) 100Mbps

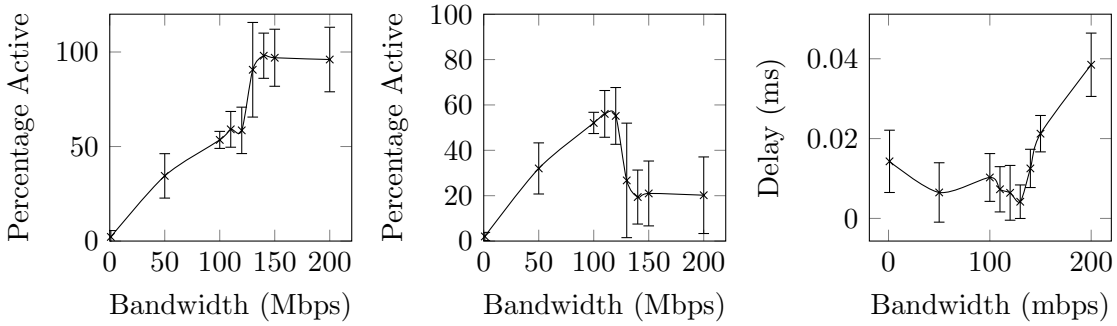
(c) 200Mbps

Figure A.10: Mininet Scenario 1.a - Network I/O



(a) Packet Loss Percentage **(b)** Latency-RTT

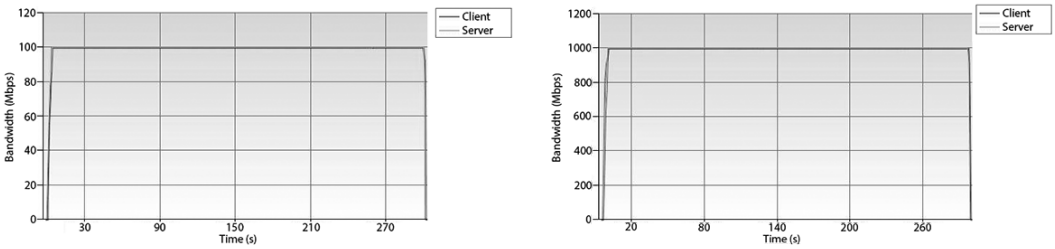
Figure A.11: Mininet Scenario 1.a - Performance



(a) Bandwidth Vs Core 1 **(b)** Bandwidth Vs Core 2 **(c)** Bandwidth Vs Delay

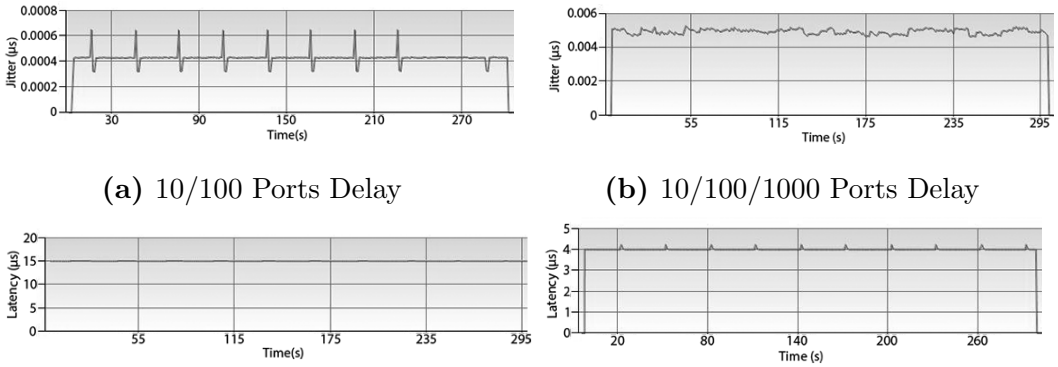
Figure A.12: Mininet Scenario 1.a - Comparisons

A.2.3 HP-Procurve



(a) 10/100 Ports Bandwidth **(b)** 10/100/1000 Ports Bandwidth

Figure A.13: HP Procurve Scenario 1 - Bandwidth



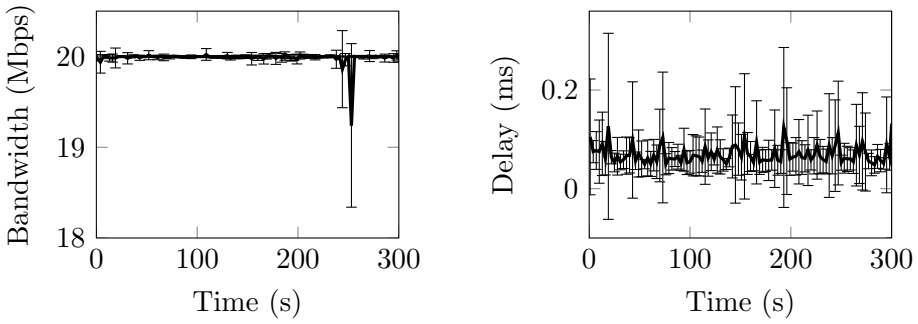
(a) 10/100 Ports Delay **(b)** 10/100/1000 Ports Delay

(c) 10/100 Ports Latency **(d)** 10/100/1000 Ports Latency

Figure A.14: HP Procurve Scenario 1 - Delay & Latency

A.3 Scenario 1.b - Bandwidth Stability

A.3.1 TP-Link OpenWrt



(a) Bandwidth **(b)** Delay

Figure A.15: TP-Link Scenario 1.b - Bandwidth & Delay

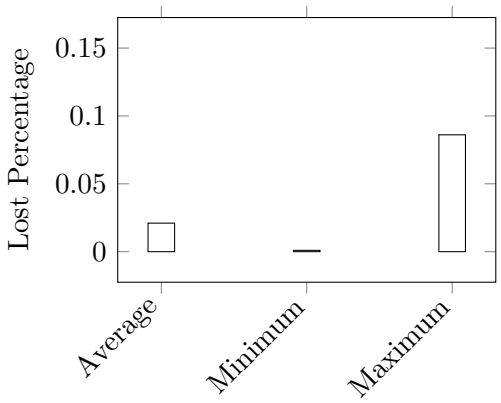
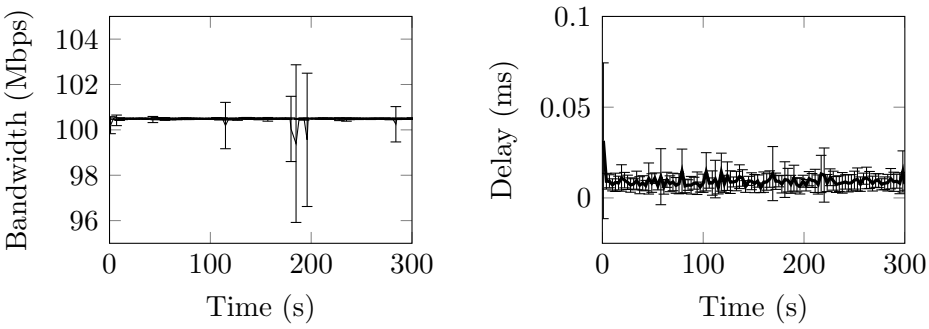


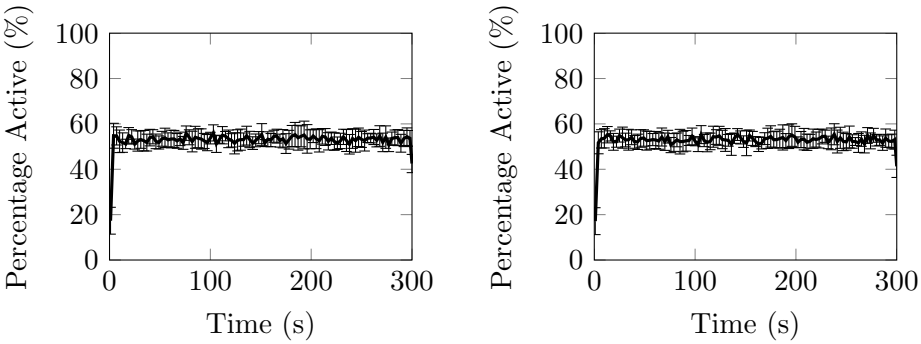
Figure A.16: TP-Link Scenario 1.b - Packet Loss

A.3.2 Mininet



(a) Bandwidth **(b)** Delay

Figure A.17: Mininet Scenario 1.b - Bandwidth & Delay



(a) Core 1 **(b)** Core 2

Figure A.18: Mininet Scenario 1.b - Cores Performance

A.4 Scenario 2 - Multiple Streams

A.4.1 TP-Link OpenWrt

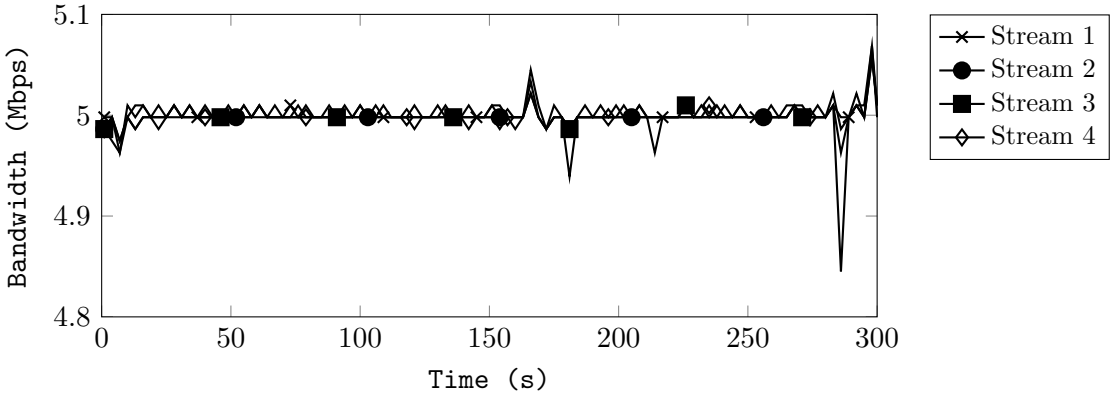


Figure A.19: TP-Link Scenario 2 - Bandwidth

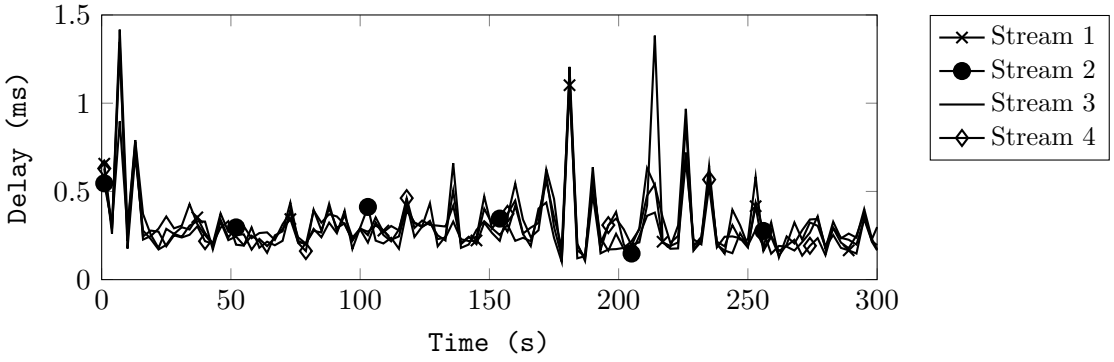
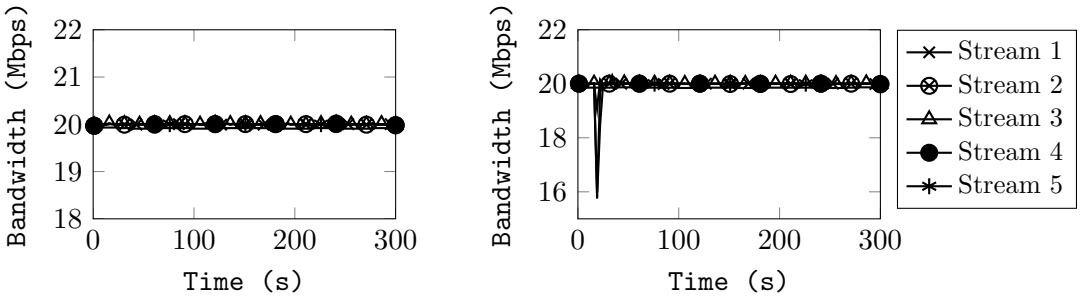


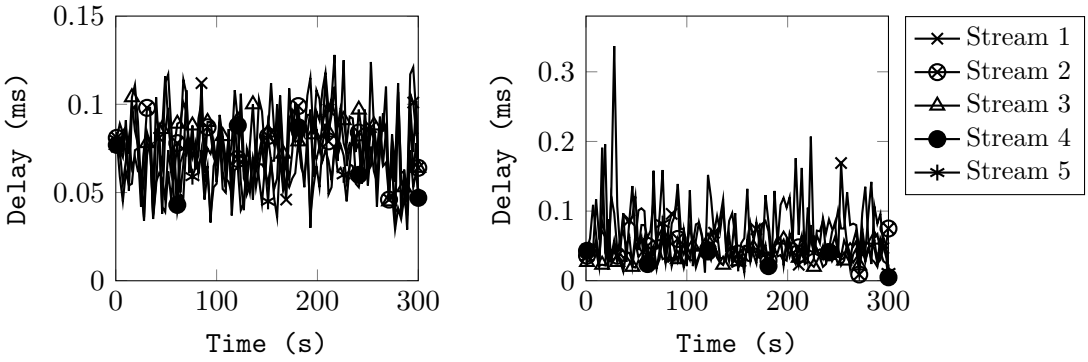
Figure A.20: TP-Link Scenario 2 - Delay

A.4.2 Mininet



(a) 2 Hosts **(b)** 6 Hosts

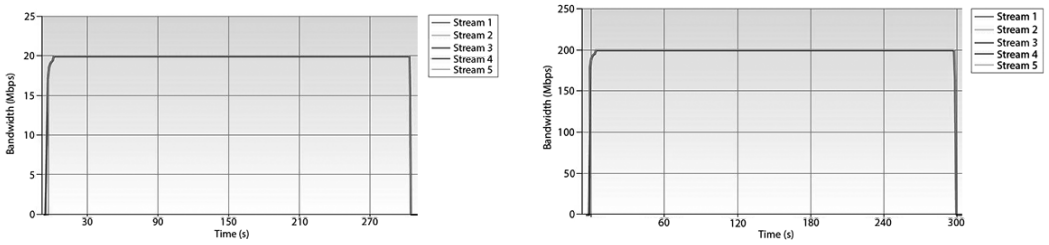
Figure A.21: Mininet Scenario 2 - Bandwidth



(a) 2 Hosts **(b)** 6 Hosts

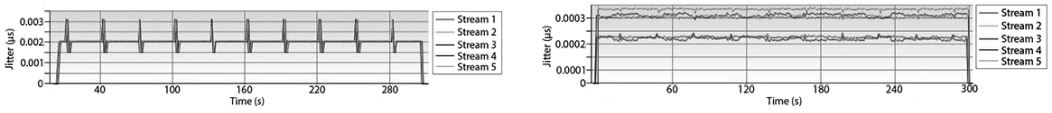
Figure A.22: Mininet Scenario 2 - Delay

A.4.3 HP-Procurve



(a) 100Mbps Bandwidth **(b)** 1000Mbps Bandwidth

Figure A.23: HP Procurve Scenario 2 - Bandwidth

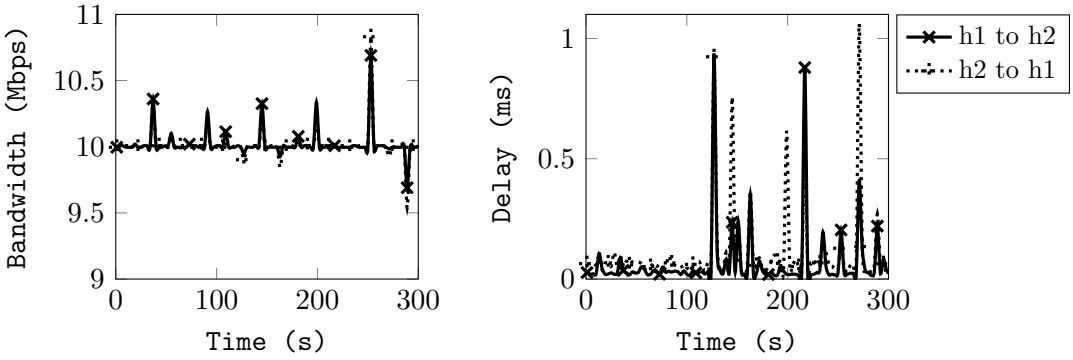


(a) 100Mbps Delay **(b)** 1000Mbps Delay

Figure A.24: HP Procurve Scenario 2 - Delay

A.5 Scenario 3 - Bidirectional Traffic

A.5.1 TP-Link OpenWrt



(a) Bandwidth **(b)** Delay

Figure A.25: TP-Link Scenario 3 - Bandwidth and Delay

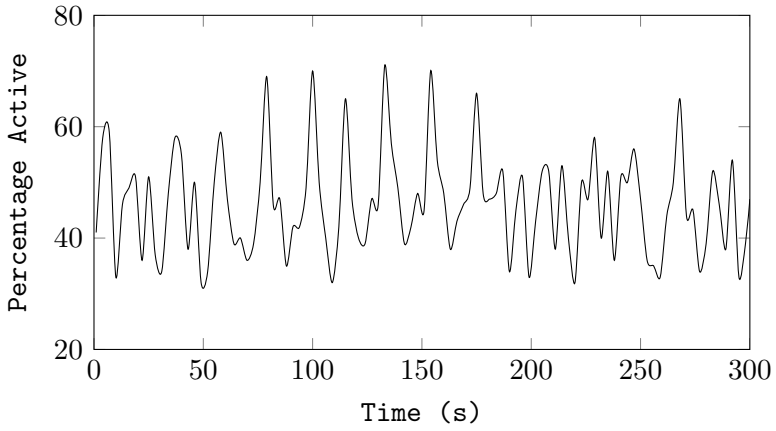
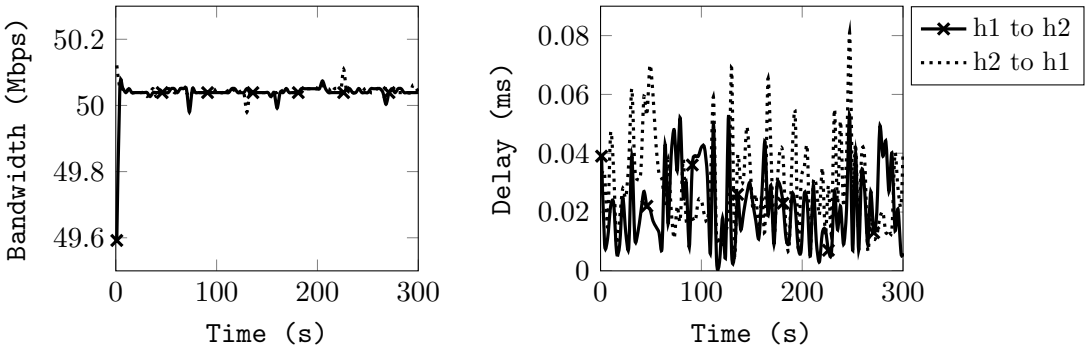


Figure A.26: TP-Link Scenario 3 - CPU Performance

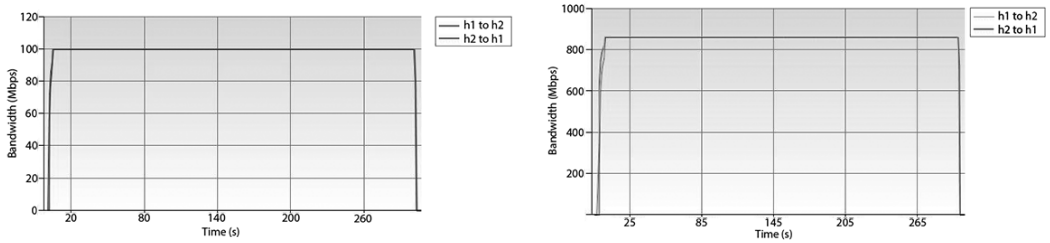
A.5.2 Mininet



(a) Bandwidth **(b)** Delay

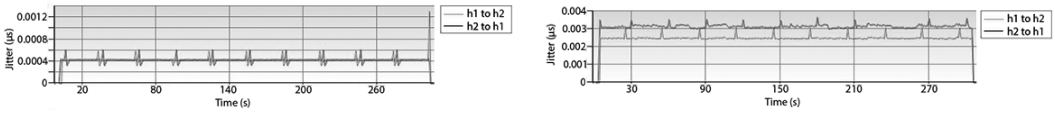
Figure A.27: Mininet Scenario 3 - Bandwidth & Delay

A.5.3 HP-Procurve



(a) 100Mbps Bandwidth **(b)** 1000Mbps Bandwidth

Figure A.28: HP Procurve Scenario 3 - Bandwidth

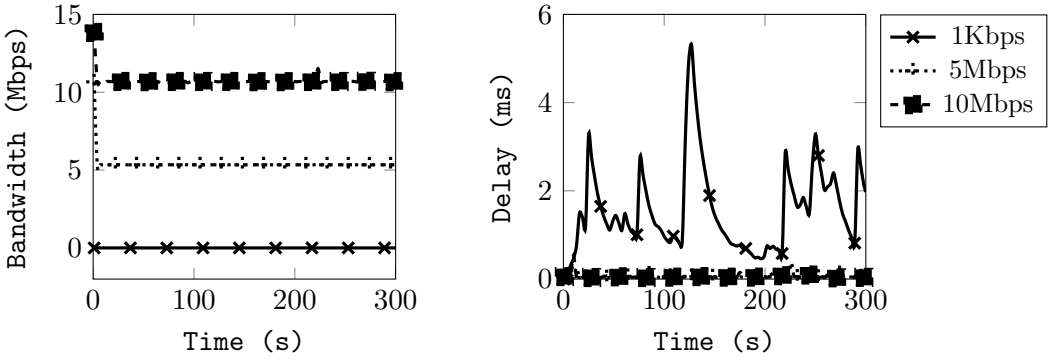


(a) 100Mbps Delay **(b)** 1000Mbps Delay

Figure A.29: HP Procurve Scenario 3 - Delay

A.6 Scenario 4 - Rate Limiting

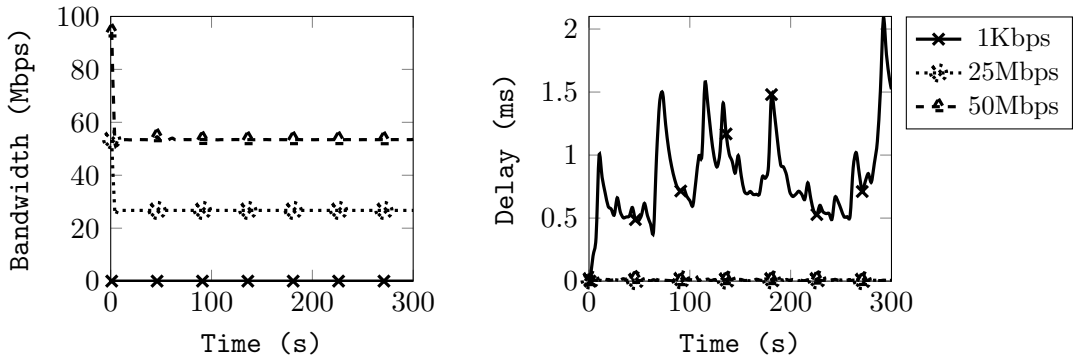
A.6.1 TP-Link OpenWrt



(a) Bandwidth **(b)** Delay

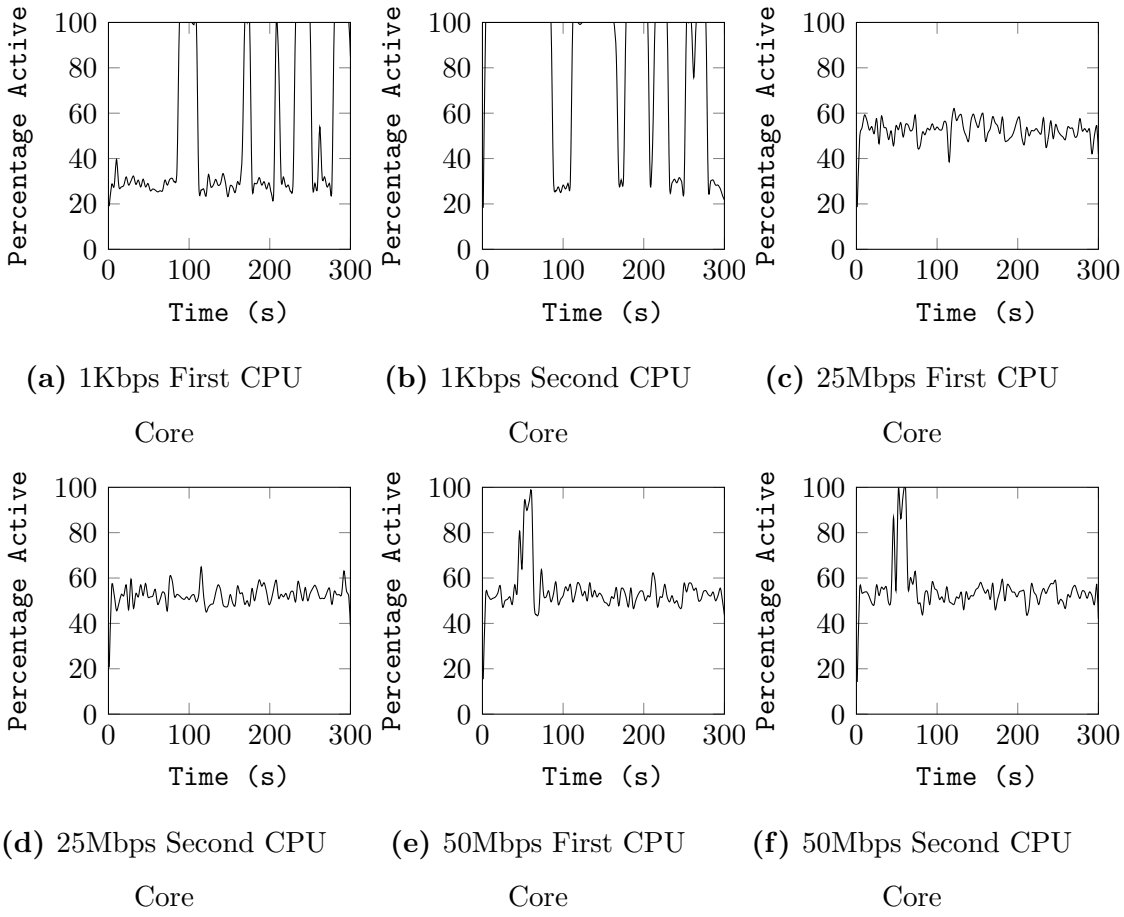
Figure A.30: TP-Link Scenario 4 - Bandwidth & Delay

A.6.2 Mininet



(a) Bandwidth **(b)** Delay

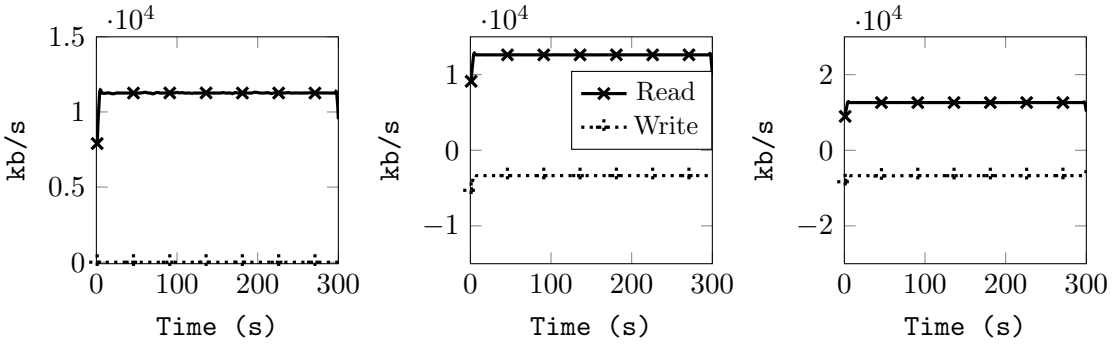
Figure A.31: Mininet Scenario 4 - Bandwidth & Delay



(a) 1Kbps First CPU Core **(b)** 1Kbps Second CPU Core **(c)** 25Mbps First CPU Core

(d) 25Mbps Second CPU Core **(e)** 50Mbps First CPU Core **(f)** 50Mbps Second CPU Core

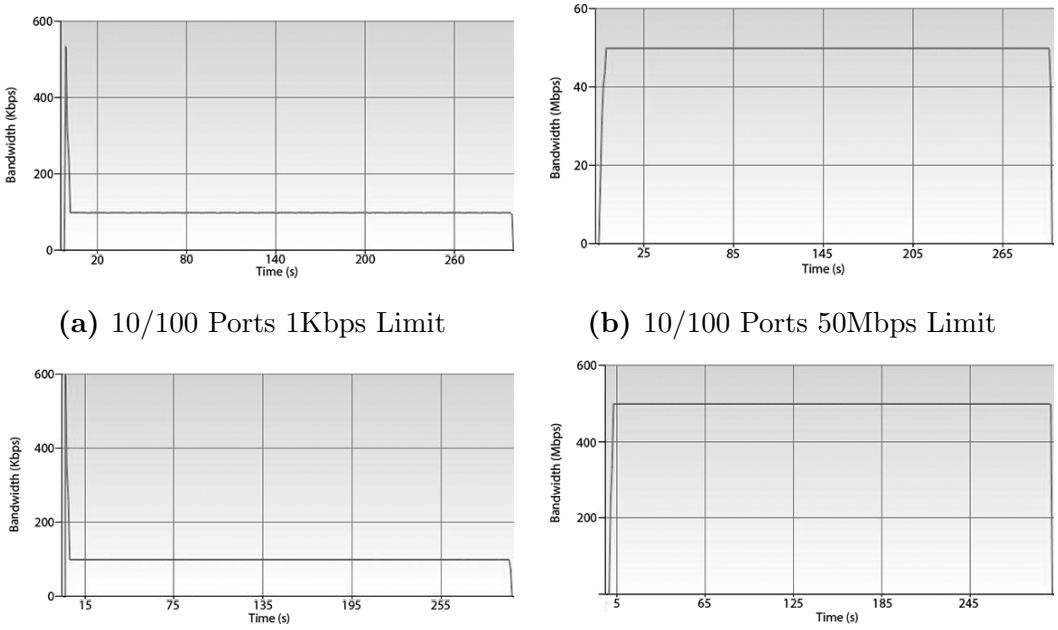
Figure A.32: Mininet Scenario 4 - CPU Performance



(a) 1Kbps **(b)** 25Mbps **(c)** 50Mbps

Figure A.33: Mininet Scenario 4 - Network I/O

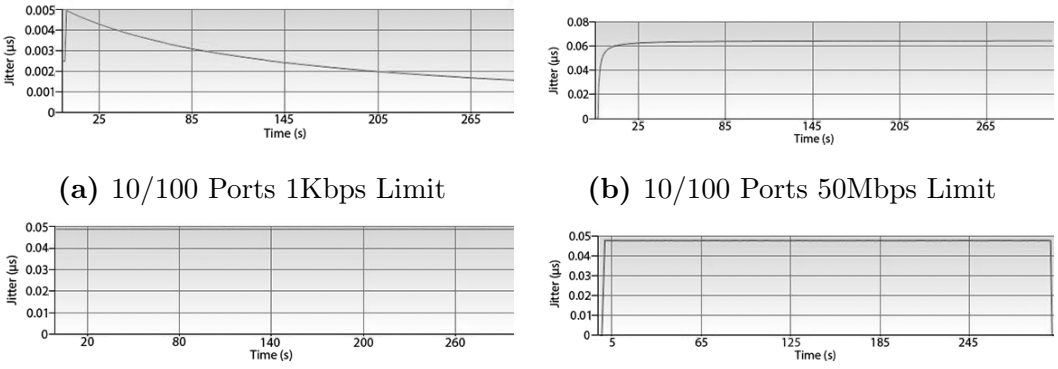
A.6.3 HP-Procurve



(a) 10/100 Ports 1Kbps Limit **(b)** 10/100 Ports 50Mbps Limit

(c) 10/100/1000 Ports 1Kbps Limit **(d)** 10/100/1000 Ports 500Mbps Limit

Figure A.34: HP Procurve Scenario 4 - Bandwidth



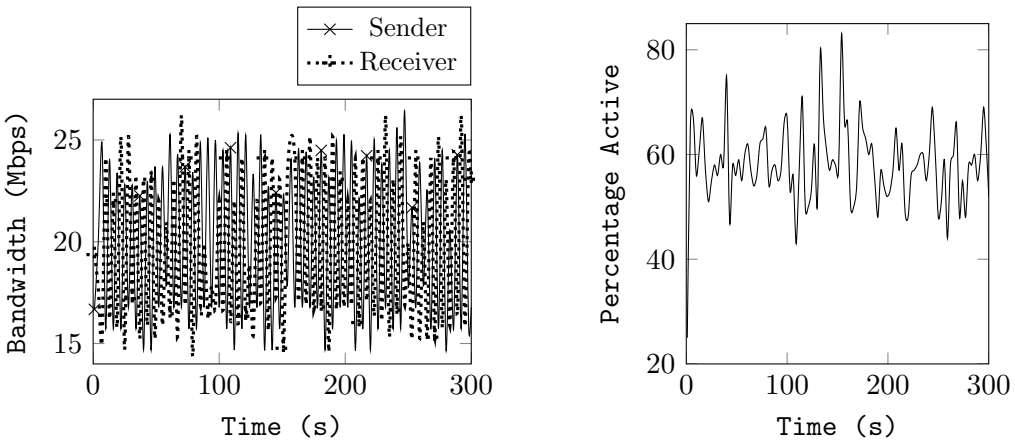
(a) 10/100 Ports 1Kbps Limit **(b)** 10/100 Ports 50Mbps Limit

(c) 10/100/100 Ports 1Kbps Limit **(d)** 10/100/100 Ports 500Mbps Limit

Figure A.35: HP Procurve Scenario 4 - Delay

A.7 Scenario 5 - TCp Bandwidth

A.7.1 TP-Link OpenWrt



(a) Bandwidth **(b)** CPU Performance

Figure A.36: TP-Link Scenario 5 - Bandwidth & CPU Performance

A.7.2 Mininet

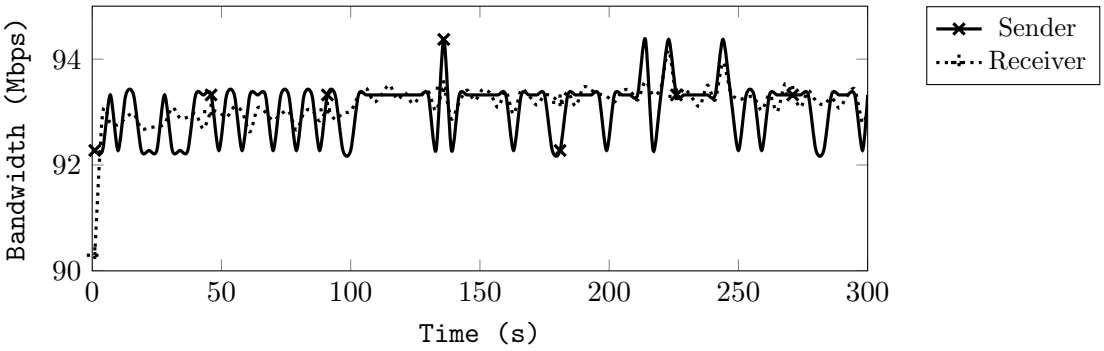
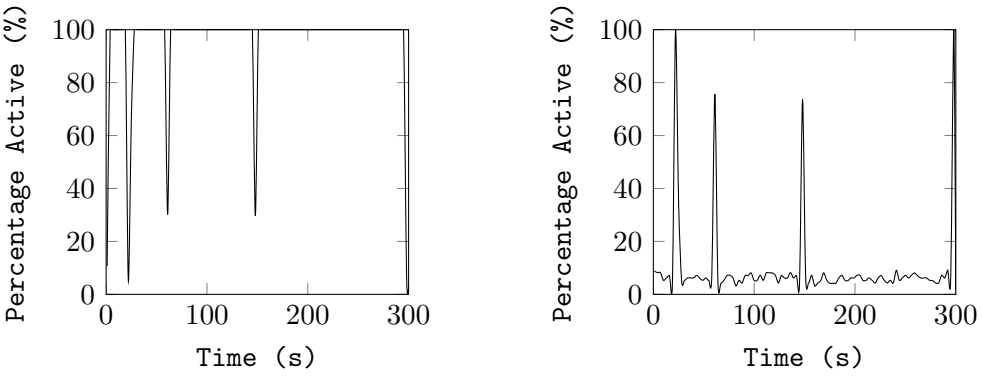
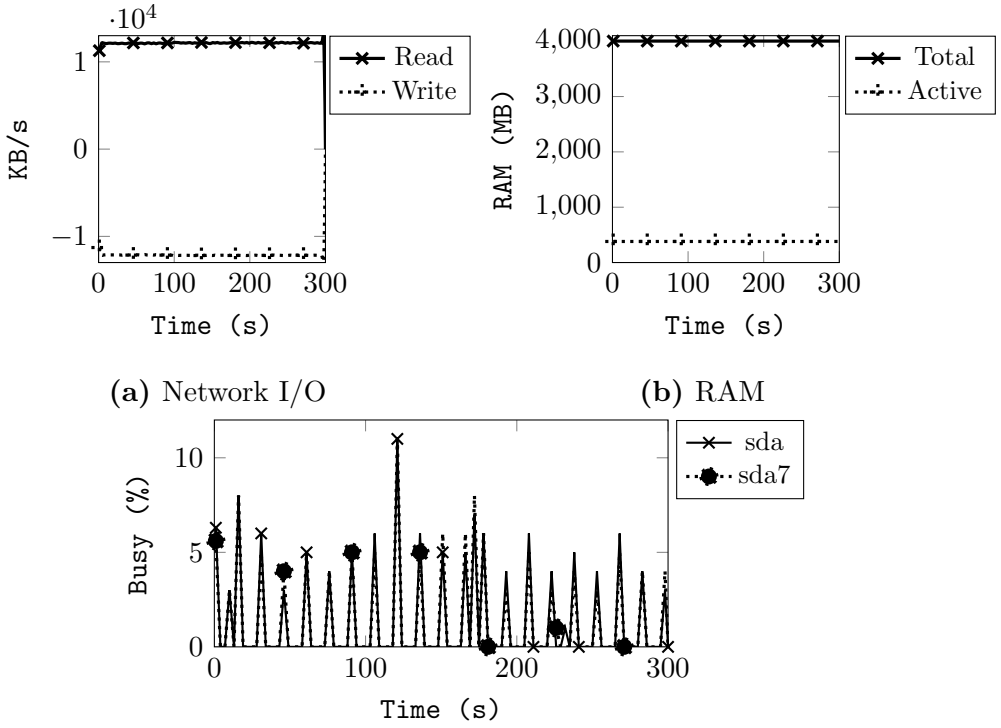


Figure A.37: Mininet Scenario 5 - Bandwidth



(a) First CPU Core **(b)** Second CPU Core

Figure A.38: Mininet Scenario 5 - CPU Performance

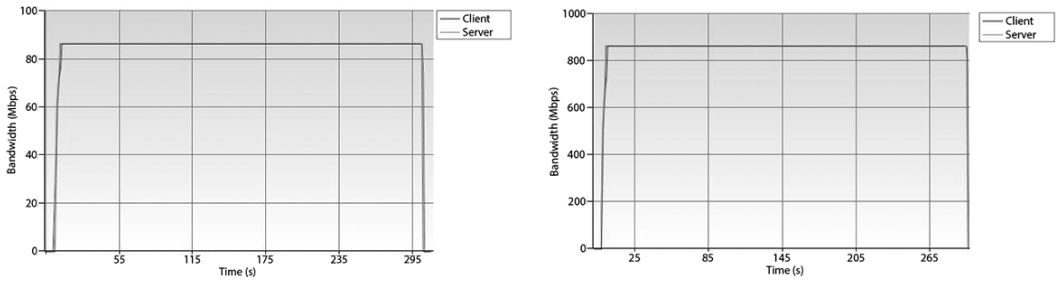


(a) Network I/O **(b)** RAM

(c) Disk Busy Percentage

Figure A.39: Mininet Scenario 5 - Network I/O, RAM & Disk Busy Performance

A.7.3 HP-Procurve



(a) 100Mbps **(b)** 1000Mbps

Figure A.40: HP Procurve Scenario 5 - Bandwidth

A.8 Scenario 6 - TCP and UDP Bandwidth

A.8.1 TP-Link OpenWrt

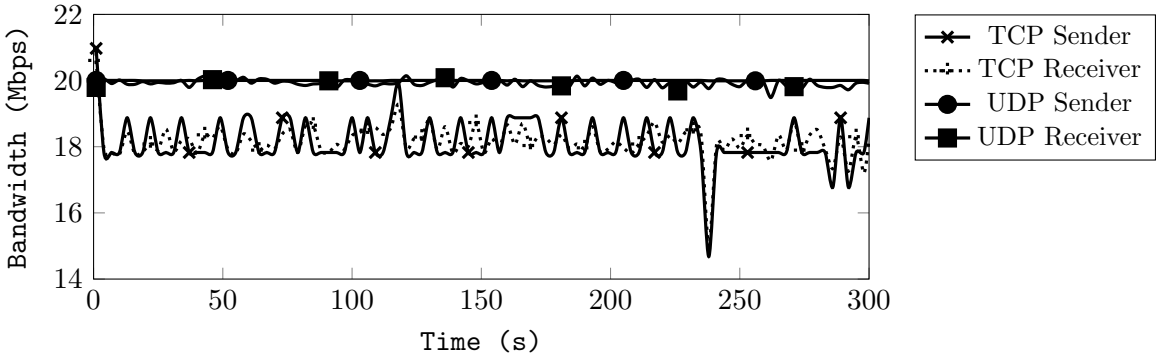
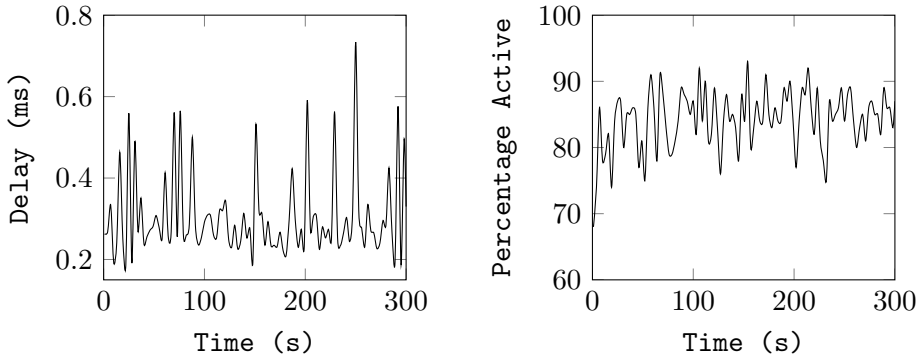


Figure A.41: TP-Link Scenario 6 - Bandwidth



(a) Delay **(b)** CPU Performance

Figure A.42: TP-Link Scenario 6 - Delay & CPU Performance

A.8.2 Mininet

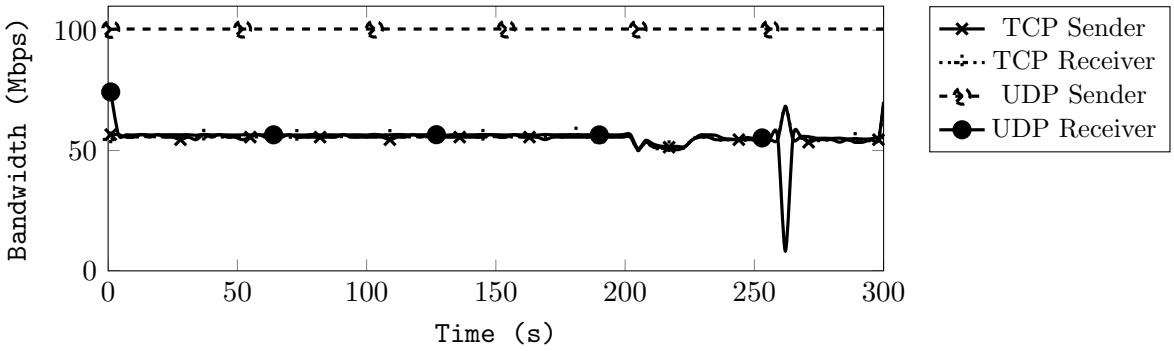


Figure A.43: Mininet Scenario 6 - Bandwidth

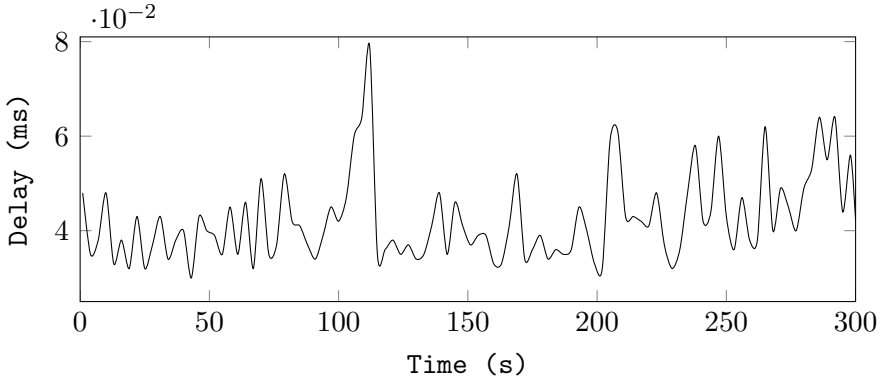
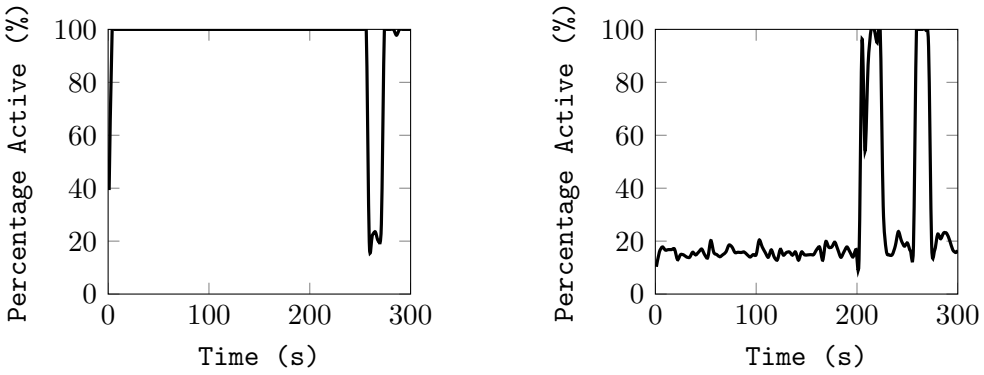
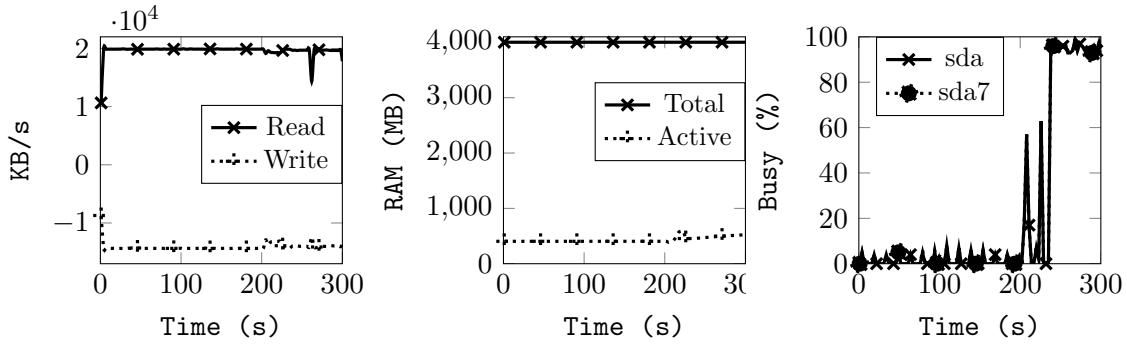


Figure A.44: Mininet Scenario 6 - Delay



(a) First CPU Core **(b)** Second CPU Core

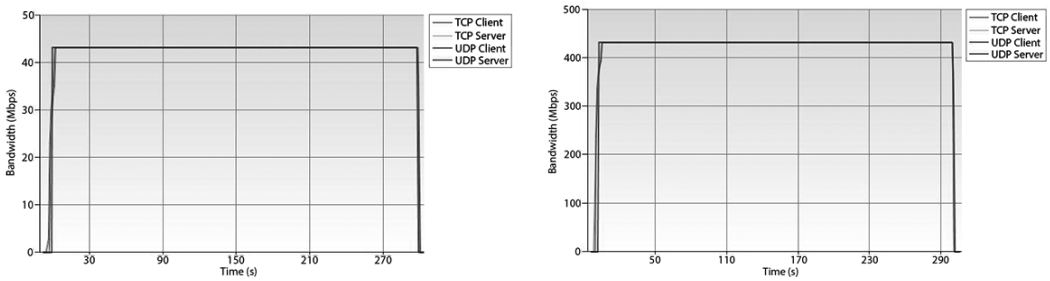
Figure A.45: Mininet Scenario 6 - CPU Performance



(a) Network I/O **(b)** RAM **(c)** Disk Busy Percentage

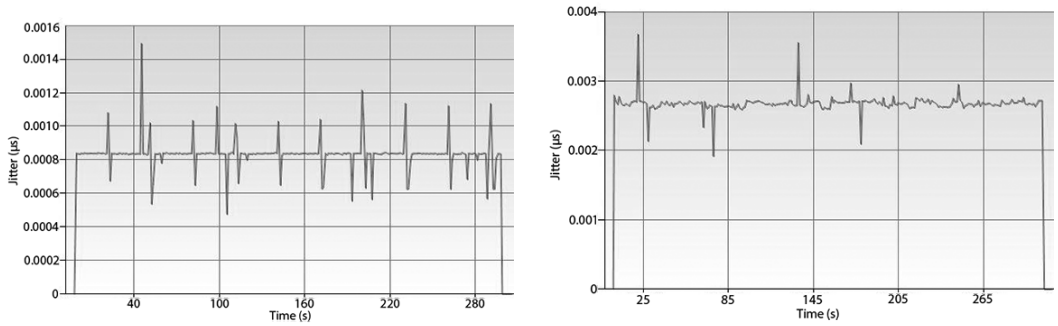
Figure A.46: Mininet Scenario 6 - Network I/O, Ram & Disk Performance

A.8.3 HP-Procurve



(a) 100Mbps Bandwidth **(b)** 1000Mbps Bandwidth

Figure A.47: HP Procurve Scenario 6 - Bandwidth



(a) 100Mbps Delay **(b)** 1000Mbps Delay

Figure A.48: HP Procurve Scenario 6 - Delay