

POLITECNICO DI TORINO

III Facoltà di Ingegneria dell'Informazione
Corso di Laurea in Telecommunication Engineering

Tesi di Laurea Magistrale

OpenFlow Switching Performance



Relatore:

prof. Andrea BIANCO

Manuel PALACIN MATEO

Luglio 2009

Abstract

The aim of this Master Thesis is to analyse the OpenFlow switching technology and to deploy a test-bed to compare OpenFlow performance with layer-2 Ethernet Switching technology and with layer-3 IP Routing technology. OpenFlow is a forwarding technology developed to isolate flows and to expand the possibilities of existing switching and routing.

Nowadays, computer networks are in constant evolution and need to be flexible, scalable and programmable to be ready to future innovations of next generation networks. OpenFlow is intended to solve the problem of assigning resources to users in an easy-way giving them a "slice" of the network without disturbing existant traffic flows. Furthermore, OpenFlow isolates traffic flows enabling researchers to run experimental protocols in campus networks and provides an architecture that allows traffic engineering according to the network status and the behaviour of elements and users that form it.

In classical routers or switches, the fast packet forwarding (data path) and the high level routing decisions (control path) occur on the same device. An OpenFlow Switch separates these two functions. The data path function still resides on the switch (server with OpenFlow software or hardward-based router/switch), while high-level routing decisions are moved to a separate device called Controller, typically a standard server. The OpenFlow Switch and Controller communicate via the OpenFlow protocol, which defines messages, such as packet-received, send-packet-out, modify-forwarding-table, and get-stats.

This thesis is structured in several chapters to discuss different Openflow issues. This report can be divided in two main parts. The first one is a descriptive part in which is explained OpenFlow technology, its architecture and its innovative features in front of software Ethernet switching and IP routing. The second one is an experimental part in which are described the tests that have been conducted and are shown the obtained results. Finally, each test is analysed to draw conclusions.

We have designed a test bench of a system formed by a Synthetic Traffic Generator and a Linux PC that acts as an OpenFlow switch, as a software Ethernet bridge or as a software IP router. In this configuration the Traffic Generator sends synthetic traffic to the PC, it process the packets according to the installed technology and then it forwards the recieved packets to the Traffic Generator where packets are captured to be analysed and to obtain statistics.

Tests have been done to analyse OpenFlow performance in terms of forwarding throughput and packet latency in normal and overload conditions and with different traffic patterns. Moreover, it has also been studied the scalability of the system

using different forwarding-table sizes and the impact in the distribution of resources when is used more than one flow (fairness test).

Finally, if we analyse the obtained results of the experiments we can conclude that OpenFlow can do the same functions as switching and routing with the same and, in some situations, better performance. OpenFlow code has been implemented optimally to maximize the forwarding performance due to its flow-table (forwarding-table) configuration. This high-performance processing and forwarding, coupled with its endless possibilities of configuration, traffic isolation and its distributed schema do it a very attractive technology for network designing and managing.

Acknowledgments

Thanks to my parents Maribel and Antonio for encourage me to study hard, to my lovely girlfriend Maria without her my life has no sense and to my grandparents, uncles, cousins and friends for their unconditional support. Also I want to thanks Pr. Andrea Bianco for make me possible to work in this project and to Robert Birke and Luca Giraudo for help me doing my experimental task during these months. This thesis is in memory of my cousin Pedro and my grandfather Manolo.

Agradecimientos a mis padres Maribel y Antonio por animarme a trabajar duro cada día, a mi querida María sin la cual mi vida no tiene sentido y a mis abuelos, tios, primos y amigos por su apoyo incondicional. También quería agradecer al Profesor Andrea Bianco el privilegio de poder realizar este proyecto junto a él y a Robert Birke y Luca Giraudo por su ayuda en el desarrollo de mis tareas experimentales durante estos meses. Esta tesis va dedicada en memoria de mi primo Pedro y mi abuelo Manolo.

Contents

Acknowledgments	III
1 Introduction	4
1.1 Motivations	4
1.2 State of the Art	5
1.3 Main objectives: What is the specific purpose of the study?	6
2 Experimental setup	7
2.1 What materials were used?	7
2.2 Methodology: How the study has been done and how materials are used?	8
2.3 Where and when was the work done?	8
2.4 Environmental impact	8
3 Technology	10
3.1 Switching and Routing	10
3.1.1 Layer-2 switching	10
3.1.2 Layer-3 routing	11
3.2 OpenFlow technology	11
3.2.1 OpenFlow switch	12
3.2.2 Controller	15
3.2.3 OpenFlow Secure Channel	15
3.2.4 OpenFlow protocol	16
3.2.5 OpenFlow Use Cases	16
4 Experiment description and Discussion of results: Performance of the system	18
4.1 Test 1: Performance of system according to packet length and load	19
4.1.1 Devices	19
4.1.2 Test configuration	19
4.1.3 Results	20

4.1.4	Discussion of results	21
4.2	Test 2: Performance of system according to forwarding table size . . .	22
4.2.1	Devices	22
4.2.2	Test configuration	22
4.2.3	Results	24
4.2.4	Discussion of results	29
4.3	Test 3: Performance of OpenFlow according to forwarding table configurations	29
4.3.1	Devices	30
4.3.2	Test configuration	30
4.3.3	Results	32
4.3.4	Discussion of results	34
4.4	Test 4: Performance of OpenFlow balancing traffic between table types	34
4.4.1	Devices	34
4.4.2	Test configuration	35
4.4.3	Results	38
4.4.4	Discussion of results	38
4.5	Test 5: Performance of system subject to a fairness test	39
4.5.1	Devices	39
4.5.2	Test configuration	40
4.5.3	Results	43
4.5.4	Discussion of results	44
4.6	Test 6: OpenFlow packet latency according to table size, table type and packet length	45
4.6.1	Devices	45
4.6.2	Test configuration	46
4.6.3	Results	47
4.6.4	Discussion of results	47
5	Conclusions	49
5.1	Future Lines	50
	Bibliography	51
A	How to install OpenFlow and NOX Controller	52
A.1	OpenFlow	52
A.2	NOX Controller	52
B	Technologies configuration	53
B.1	Layer 2: Bridging configuration	53
B.2	Layer 3: Routing configuration	53

B.3	OpenFlow configuration	53
C	Scripts	55
C.1	Switching	55
C.2	Routing	55
C.3	OpenFlow	56
C.3.1	Flow-table: 100 Linear entries	56
C.3.2	Flow-table: 64K Hash entries	56
C.3.3	Flow-table: 128K Hash entries	57
C.3.4	Flow-table: 100 Linear and 96K Hash entries	57
C.3.5	Flow-table: 100 Linear and 128K Hash entries	58
D	Scripts for fairness test	60
D.1	Switching	60
D.2	Routing	60
D.2.1	Routing-table: 1024 entries	60
D.2.2	Routing-table: 64K entries	61
D.3	OpenFlow	61
D.3.1	Flow-table: 1024 entries	61
D.3.2	Flow-table: 64K entries	62

List of Tables

4.1	Example of fairness ration between two flows	44
4.2	OpenFlow packet latency for different table types and sizes	47

List of Figures

3.1	OpenFlow network	13
3.2	OpenFlow Switch architecture	14
3.3	Open Flow Table fields	14
4.1	Experimental environment	18
4.2	64 byte - Throughput and Average Latency vs Load	20
4.3	1024 byte - Throughput and Average Latency vs Load	21
4.4	Throughput at 75 and 99 % of Load vs Packet Length	21
4.5	Load at 10% with table size 1: Throughput and Average Latency vs Packet Size	24
4.6	Load at 10% with table size 1024: Throughput and Average Latency vs Packet Size	24
4.7	Load at 10% with table size 8192: Throughput and Average Latency vs Packet Size	25
4.8	Load at 10% with table size 64K: Throughput and Average Latency vs Packet Size	25
4.9	Load at 10% with table size 128K: Throughput and Average Latency vs Packet Size	25
4.10	Load at 40% with table size 1: Throughput and Average Latency vs Packet Size	26
4.11	Load at 40% with table size 1024: Throughput and Average Latency vs Packet Size	26
4.12	Load at 40% with table size 8192: Throughput and Average Latency vs Packet Size	26
4.13	Load at 40% with table size 64K: Throughput and Average Latency vs Packet Size	27
4.14	Load at 40% with table size 128K: Throughput and Average Latency vs Packet Size	27
4.15	Load at 99% with table size 1: Throughput and Average Latency vs Packet Size	27

4.16	Load at 99% with table size 1024: Throughput and Average Latency vs Packet Size	28
4.17	Load at 99% with table size 8192: Throughput and Average Latency vs Packet Size	28
4.18	Load at 99% with table size 64K: Throughput and Average Latency vs Packet Size	28
4.19	Load at 99% with table size 128K: Throughput and Average Latency vs Packet Size	29
4.20	OpenFlow flow-tables: Load 10% - Throughput and Average Latency vs Packet Length	32
4.21	OpenFlow flow-tables: Load 40% - Throughput and Average Latency vs Packet Length	32
4.22	OpenFlow flow-tables: Load 80% - Throughput and Average Latency vs Packet Length	33
4.23	OpenFlow flow-tables: Load 90% - Throughput and Average Latency vs Packet Length	33
4.24	OpenFlow flow-tables: Load 99% - Throughput and Average Latency vs Packet Length	33
4.25	OpenFlow tables: Throughput vs Packet Length balancing traffic to one table	38
4.26	OpenFlow tables: 96 Byte - Throughput vs Load balancing to both tables	38
4.27	Fairness test configuration for 2 inputs and 1 output interfaces	40
4.28	Fairness test (1 input - 1 output): Percentage difference Tx/Rx between fixed-variable flows	43
4.29	Fairness test (2 inputs - 1 output): Percentage difference Tx/Rx between fixed-variable flows	43
4.30	Packet transmission between Agilent tester and OpenFlow PC	45

Chapter 1

Introduction

This Master Thesis wants to reflect the results obtained after the analysis of the OpenFlow switching protocol and the subsequent launching in an experimental stage. OpenFlow switching technology is a protocol that lets researchers to do experiments and to test new protocols in production networks without interfere existing traffic. OpenFlow allows switches to segment traffic into flows using rules configured in a particular switching table called flow table.

This document consists of different chapters: this first chapter called introduction exposes the existing problematic and the specific hypotheses. The second chapter describes the methodology and the working environment used to develop the project. The third chapter describes software Switching, Routing and OpenFlow technology. The fourth chapter collects both analytical and experimental results about the development and discusses these obtained results. And finally the fifth chapter outlines the conclusions learned after this project.

1.1 Motivations

Networks have become a critical part of business and institutions. A failure in a network could become a failure in business processes and the consequent money lost. Therefore network administrators have to ensure a perfect network running close to 100 percent of the time. But many times researchers need real environments in which they can test experimental network protocols and usually encounter opposition from network administrators who forbid them to test their experiments in production networks. Here is where it appears the term programmable networks and where OpenFlow technology can help to solve this problematic.

OpenFlow technology allows network administrators to segment telecommunication networks programming the devices involved in the system. OpenFlow devices identify different traffic flows following rules pre-configured by network managers.

This technology virtualizes network into flows in a way that there are no interferences between traffics. Furthermore, once the virtualization is done the network administrator can delegate the management of network segment/s to the researchers as if it was a new network.

Summarizing programming networks with OpenFlow technology take following advantages:

- Network virtualization: experimental tests can be deployed into production networks without disturbing existant isolated traffic
- Network managers can delegate virtual segments to be managed by researchers
- Low cost devices: OpenFlow switches can be deployed into UNIX/Linux platforms
- OpenFlow protocol can be exploited in modern Ethernet switches/routers from different vendors as an extra functionality using TCAM

Therefore the need for programmable networks that allow network managers to virtualize networks in an easy-way providing researchers a real infrastructure in which they can test their experiments make that this study takes sense. And in this context is where we want to test the performance of OpenFlow technology againts the traditional layer-2 switching and layer-3 routing.

1.2 State of the Art

We can separate OpenFlow technology into two parts. The first one is referring to the switching function in which OpenFlow only follows a pre-defined rules and forwards packet flows to the correct destination. Therefore we can see this feature as a filter like a firewall or a NAT. The second one is referring to the virtualizing function in which OpenFlow using a pre-defined configuration can isolate traffic flows. In this second field we can find other existing technologies like VPN (Virtual Private Network) in which we can find Ethernet VLANs (IEEE 802.1Q) that works at Layer-2 level, MPLS tunneling technologies that works at Layer-2/Layer-3 level or Pseudo-wired circuits of obsolet technologies like Frame Relay or ATM. But the problem is that they do not have an easy management or they are not so flexible in a multi-protocol sense. Furthermore there are some other experiments like GENI project [1] that is developing a programmable network for research in which a researcher can obtain a slice of resources (network, CPU, RAM, disk...) to do his experiment. But this approach is very ambitious and it will take years to be deployed and it will be very costly. Hence we have to try to find a tehcnology that combines the following features without being a utopia:

- High performance
- High stability
- Easy configurable and easy manageable
- Re-use existing hardware (network elements and infrastructure)
- Low-cost (can be deployed in UNIX/Linux platforms)

And in this study we are going to see if OpenFlow technology is capable to offer all of these key-points becoming an authentic silver bullet.

1.3 Main objectives: What is the specific purpose of the study?

This Master Thesis will be done into 3 phases. The first phase is documentation about OpenFlow technology and other network technologies. The second phase is to define an experimental environment and deploy a real scenario to test OpenFlow performance against other existing technologies. And the third phase is to evaluate the obtained results to take conclusions.

Once the basic phases of the Master Thesis are defined, we can describe the main objectives of this study as the following:

- Understanding network virtualization
- Understanding OpenFlow technology
- Configuring OpenFlow protocol in a Linux-based platform
- Designing a real environment to deploy a test-bed
- Testing the experiment using synthetic traffic running classic Layer-2 switching and obtain results
- Testing the experiment using synthetic traffic running classic Layer-3 routing and obtain results
- Testing the experiment using synthetic traffic running OpenFlow protocol and obtain results
- Compare and discuss obtained results
- Extracting conclusions about the work done
- Future lines of the project (providing new improvements or new features)

Chapter 2

Experimental setup

This chapter will explain what resources are available and the methodology used to realize the different experimental tests of the project. The different tests will be developed at the telecommunication department of the Mario Boella Institute in the main campus of the Politecnico di Torino.

2.1 What materials were used?

At the telecommunication department we have several devices in which we can test the experiments. For the completion of the work we are going to use one desktop PCs, one laptop and one traffic generator. Following will be described each device:

- PC:
 - CPU: Intel Core2 Duo E6750 2.66 Ghz
 - RAM: 8 GB 1066MHz
 - HARD DISK:40 GB
 - NIC: 2 x Intel PRO/1000 PT dual port 1 Gbps PCI-Express
 - Operating System: Linux Ubuntu 8.10 64 bit. Kernel 2.6.27

- Laptop:
 - CPU: Intel Core2 Duo P8400 2,6 Ghz
 - RAM: 3 GB 800MHz
 - HARD DISK: 320 GB
 - NIC: Realtek RTL8168 Gigabit Ethernet
 - Operating System: Linux Ubuntu 8.10 64 bit. Kernel 2.6.27

- Agilent N2X traffic generator:
 - Modular chassis: up to 4 modules
 - Used modules: E7919A 1000Base-X GBIC-RJ45 and E7919A 1000Base-X
 - N2X Packets 6.4 System Release traffic analyzer software

2.2 Methodology: How the study has been done and how materials are used?

Before project begins we provide the different steps that we are going to follow for conducting the experiments. The first step was a previous in depth study about the OpenFlow protocol, trying to understand what are the different elements, how it works and its complexity. The second step was the installation of the OpenFlow switching technology into a PC for a first test contact. The third one was to design the different test that we are going to perform comparing the OpenFlow technology against classic software switching and routing techniques. The fourth step was perform the different tests using a PC as a SUT (System Under Test) and the Agilent Traffic Generator to generate and collect data. The fifth step was collect and process all the data provided by the Agilent Traffic Generator. And finally the last step was discuss the results to draw conclusions that are presented in this report.

2.3 Where and when was the work done?

The different experiments have been developed at telecommunication department of the Mario Boella Institute, a research center associated with the University Politecnico di Torino. The experimental work has been done during 5 months, 5 days per week.

2.4 Environmental impact

The completion of this work has tried to be as respectful as possible to the environment. Among the initiatives taken to minimize the environmental impact we can highlight the following:

- We have tried to work using natural light from the sun.

- We have minimized the use of paper using PCs and digital media devices and when it has been necessary to write, we have used recycled paper sheets.
- We have tried to use new brand processors that reduce the current consumption and we have taken care of switch off the devices when they are not used.

Chapter 3

Technology

This chapter will explain the different technologies that will be used for testing. First it will be briefly explained the classic software switching/routing technologies describing how they work. Then it will be described the OpenFlow technology which is the aim of the project. It is important to note that this project evaluates software based technologies that run over UNIX/Linux architectures.

3.1 Switching and Routing

Switching and Routing are two methods to forward packets. The difference between them is the OSI level at which they perform the forwarding. Switching or Bridging (we will use both terms indistinctly) are performed at layer 2 while Routing is performed at layer 3. In this project when we refer to switching, we are talking about the Ethernet technology and when we refer to routing we are talking about IP forwarding. For further information about switching and routing go to CISCO Internetworking reference book [2].

3.1.1 Layer-2 switching

Ethernet switching is performed at Data Link Layer of the OSI stack. This means that link layer controls data flow, handles transmission errors, provides physical (as opposed to logical) addressing, and manages access to the physical medium. Ethernet differentiates the OSI link layer into two separate sublayers: the Media Access Control (MAC) sublayer and the Logical Link Control (LLC) sublayer. The MAC sublayer permits and orchestrates media access, while the LLC sublayer deals with framing, flow control, error control, and MAC sublayer addressing.

Linux software switching is a code that implements a subset of the ANSI/IEEE 802.1d standard [3]. The original Linux bridging was first done in Linux kernel 2.2,

then rewritten by Lennert Buytenhek. The code for bridging has been integrated into 2.4 and 2.6 kernel series (see appendix B.1 to know how to configure software switching bridge-utils).

3.1.2 Layer-3 routing

IP routing is performed at Network Layer of the OSI stack. This means that the Network Layer is responsible for end-to-end (source to destination) packet delivery including routing through intermediate hosts, whereas the Data Link Layer is responsible for node-to-node (hop-to-hop) frame delivery on the same link. The Network Layer provides the functional and procedural means of transferring variable length data sequences from a source to a destination host via one or more networks while maintaining the quality of service and error control functions. Some of the functions that performs Network Layer are Logical Addressing, Routing, Datagram Encapsulation, Fragmentation and Reassembly, Error Handling and Diagnostics.

Linux software routing is performed by the module IP forwarding of the Linux Kernel. This allow us to convert a simple Linux into a router that forwards IP packets (see appendix B.2 to know how to configure software routing).

3.2 OpenFlow technology

This section will talk about the OpenFlow technology which is the aim of the project. OpenFlow provides network administrators a set of elements that allows them to define flows and to define the path that they will follow without disturbing the existing traffic. It also provides methods to define policies to find automatically paths that accomplish certain characteristics, like having higher band width, suffering less latency, reducing the number of hops and reducing the required energy that needs traffic to reach its destination. Therefore we can see OpenFlow as a tool that allows network managers to perform traffic engineering in their systems.

In a classical router or switch, the fast packet forwarding (data path) and the high level routing decisions (control path) occur on the same device. An OpenFlow Switch separates these two functions. The data path portion still resides on the switch, while high-level routing decisions are moved to a separate controller, typically a standard server. The OpenFlow Switch and Controller communicate via the OpenFlow protocol, which defines messages, such as packet-received, send-packet-out, modify-forwarding-table, and get-stats.

The data path of an OpenFlow Switch presents a clean flow table in which each flow entry contains a set of packet fields to match and a defined action (such as send-out-port, modify-field, or drop). When an OpenFlow Switch receives a packet it has never seen before, for which it has no matching flow entries, it sends this

packet to the controller. The controller then makes a decision on how to handle this packet. It can drop the packet, or it can add a flow entry directing the switch on how to forward similar packets in the future.

OpenFlow technology is a set of networks elements (hardware and software based) and an own protocol to configure the behavior and the actions of a network. Following, this elements will be explained in detail based in the OpenFlow white paper [4] and the OpenFlow Specification [5].

An OpenFlow network is formed basically by one or more OpenFlow-switches which contains a Flow-table with the defined flow entries and the actions to be performed, one or more Controller which are the responsible of adding and dropping flow entries, a Secure-Channel that interconnects the switch with the controller and the OpenFlow Protocol for signalling the whole architecture (see Figure 3.1).

3.2.1 OpenFlow switch

There are 2 types of OpenFlow switch. The first type is the hardware-based commercial switches that use the TCAM and the Operating System of the switch/router to implement the Flow-table and the OpenFlow protocol. The second type is the software-based switches that use UNIX/Linux systems to implement the entire OpenFlow switch functions. In this project is going to be used the OpenFlow software-based edition (OpenFlow version v0.8.9r2) in a Linux Ubuntu 8.10 64 bit Kernel 2.6.27 (see appendix B.3 to know how to configure OpenFlow).

An OpenFlow Switch consists of at least three parts (see Figure 3.2): (1) A Flow Table, with an action associated with each flow entry, to tell the switch how to process the flow, (2) A Secure Channel that connects the switch to a remote control process (called the controller), allowing commands and packets to be sent between a controller and the switch using (3) The OpenFlow Protocol, which provides an open and standard way for a controller to communicate with a switch. By specifying a standard interface (the OpenFlow Protocol) through which entries in the Flow Table can be defined externally, the OpenFlow Switch avoids the need for researchers to program the switch.

Flow-table

The Flow-table of the OpenFlow switch consist of a three field table (see Figure 3.3). These three fields are (1) a packet header that defines the flow, (2) the action, which defines how the packets should be processed, and (3) statistics, which keep track of the number of packets and bytes for each flow, and the time since the last packet matched the flow (to help with the removal of inactive flows).

Internally inside the OpenFlow table there are 2 tables where are defined the flows. The first table is a linear table that uses wildcards to match defined flows. It

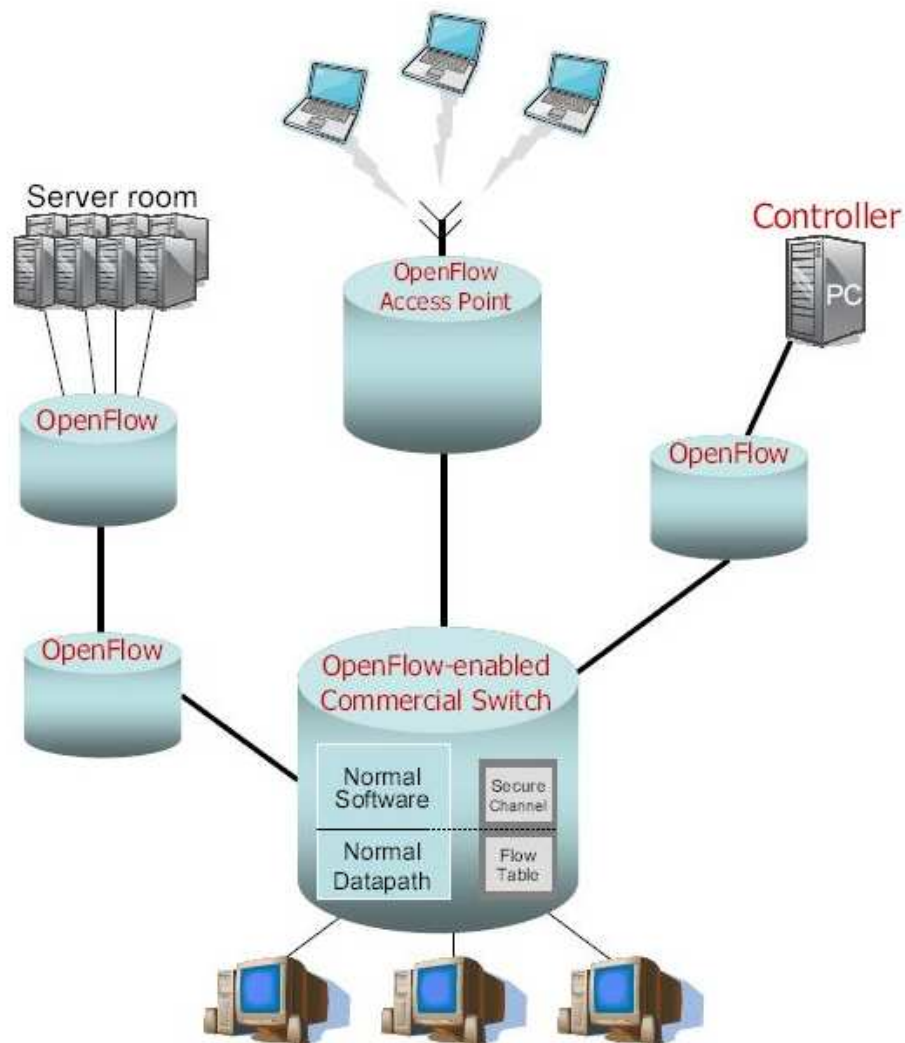


Figure 3.1. OpenFlow network

has a size of 100 wildcard flow-entries. A wildcard-entry is an entry that only defines some fields of a flow (MAC address, IP address, TCP port,...), and the OpenFlow switch will match the flows that match this fields of the linear table. The second table is an exact match table that use a HASH algorithm to store and search the entries. It has a size of 131072 exact-match flow-entries. An exact-match entry is an entry that defines all possible values of a flow (Input port,MAC source/destination address, Ethernet protocol type, IP MAC source/destination address, network protocol, source/destination port,...), and the OpenFlow switch will match flows only if they are exact as the HASH table entries.

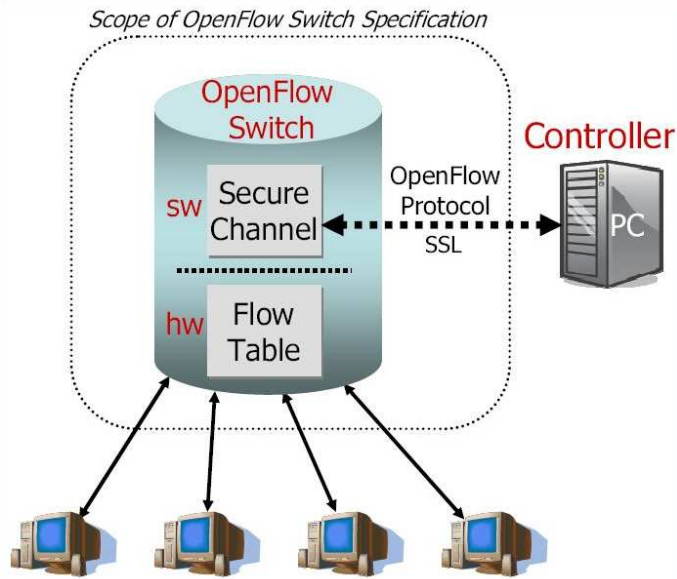


Figure 3.2. OpenFlow Switch architecture

Header Fields		Counters	Actions						
In Port	VLAN ID	Ethernet			IP			TCP	
		SA	DA	Type	SA	DA	Proto	Src	Dst

Figure 3.3. Open Flow Table fields

An example of linear entry is following:

```
dpctl add-flow nl:0 dl_type=0x0800, nw_dst=192.0.0.1, \
idle_timeout=0, actions=output:1
```

An example of exact-match is following:

```
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
dl_src=00:00:C0:03:00:02, dl_dst=00:00:C0:04:00:02, \
dl_type=0x0800, nw_src=192.3.0.2, nw_dst=192.4.0.2, \
nw_proto=17, tp_src=20, tp_dst=80, idle_timeout=0, \
actions=output:1
```

OpenFlow Switch actions

The OpenFlow switch can perform three different actions associated to the flow entries of the Flow-table and a fourth if it is hardware based device:

1. Forward this flow's packets to a given port (or ports). This allows packets to be routed through the network. In most switches this is expected to take place at line-rate.
2. Encapsulate and forward this flow's packets to a controller. Packet is delivered to Secure Channel, where it is encapsulated and sent to a controller. Typically used for the first packet in a new flow, so a controller can decide if the flow should be added to the Flow-table. Or in some experiments, it could be used to forward all packets to a controller for processing.
3. Drop this flow's packets. Can be used for security, to curb denial of service attacks, or to reduce spurious broadcast discovery traffic from end-hosts.
4. For hardware based devices: Forward this flow's packets through the switch's normal processing pipeline.

3.2.2 Controller

The Controller is the OpenFlow network element that is the responsible for managing the OpenFlow switches. The controller can be a device that only adds and removes flow-entries statically or a sophisticated device that can dynamically add and remove flow-entries depending on different pre-configured conditions.

The OpenFlow project provides in the source code a simple controller to manage switches but sometimes it could be more interesting to use a more sophisticated controller like NOX controller [6] which is compatible with the OpenFlow protocol.

3.2.3 OpenFlow Secure Channel

The secure channel is the interface that connects each OpenFlow switch to a controller. Through this interface, the controller configures and manages the switch, receives events from the switch, and send packets out the switch.

Between the datapath and the secure channel, the interface is implementation-specific, however all secure channel messages must be formatted according to the OpenFlow protocol. Secure Channel is encrypted using SSL.

3.2.4 OpenFlow protocol

The OpenFlow protocol is the language that use OpenFlow devices to communicate between them. The OpenFlow protocol supports three message types, controller-to-switch, asynchronous, and symmetric, each with multiple sub-types. Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation. For further information and to see the details of the message types revise the OpenFlow specification document [5].

3.2.5 OpenFlow Use Cases

Below it will be described different examples of use cases from OpenFlow white paper [4]:

- Example 1: Network Management and Access Control. We can use OpenFlow as a generalization of Ethane [7] to allow network managers to define a network-wide policy in the central controller, which is enforced directly by making admission control decisions for each new flow. A controller checks a new flow against a set of rules, such as "Guests can communicate using HTTP, but only via a web proxy" or "VoIP phones are not allowed to communicate with laptops". A controller associates packets with their senders by managing all the bindings between names and addresses, it essentially takes over DNS, DHCP and authenticates all users when they join, keeping track of which switch port (or access point) they are connected to. One could envisage an extension in which a policy dictates that particular flows are sent to a user's process in a controller, hence allowing researcher-specific processing to be performed in the network.
- Example 2: VLANs. OpenFlow can easily provide users with their own isolated network, just as VLANs do. The simplest approach is to statically declare a set of flows which specify the ports accessible by traffic on a given VLAN ID. Traffic identified as coming from a single user (for example, originating from specific switch ports or MAC addresses) is tagged by the switches (via an action) with the appropriate VLAN ID. A more dynamic approach might use a controller to manage authentication of users and use the knowledge of the users' locations for tagging traffic at runtime.
- Example 3: Mobile wireless VOIP clients. For this example consider an experiment of a new call-handoff mechanism for WiFi-enabled phones. In the experiment VOIP clients establish a new connection over the OpenFlow-enabled

network. A controller is implemented to track the location of clients, re-routing connections (by reprogramming the Flow Tables) as users move through the network, allowing seamless handoff from one access point to another.

- Example 4: A non-IP network. So far, our examples have assumed an IP network, but OpenFlow doesn't require packets to be of any one format so long as the Flow Table is able to match on the packet header. This would allow experiments using new naming, addressing and routing schemes. There are several ways an OpenFlow-enabled switch can support non-IP traffic. For example, flows could be identified using their Ethernet header (MAC src and dst addresses), a new EtherType value, or at the IP level, by a new IP Version number. More generally, we hope that future switches will allow a controller to create a generic mask (offset + value + mask), allowing packets to be processed in a researcher-specified way.
- Example 5: Processing packets rather than flows. The examples above are for experiments involving flows where a controller makes decisions when the flow starts. There are, of course, interesting experiments to be performed that require every packet to be processed. For example, an intrusion detection system that inspects every packet, an explicit congestion control mechanism, or when modifying the contents of packets, such as when converting packets from one protocol format to another. There are two basic ways to process packets in an OpenFlow-enabled network. First, and simplest, is to force all of a flow's packets to pass through a controller. To do this, a controller doesn't add a new flow entry into the Flow Switch it just allows the switch to default to forwarding every packet to a controller. This has the advantage of flexibility, at the cost of performance. It might provide a useful way to test the functionality of a new protocol, but is unlikely to be of much interest for deployment in a large network.

The second way to process packets is to route them to a programmable switch that does packet processing for example, a NetFPGA-based programmable router. The advantage is that the packets can be processed at line-rate in a user-definable way; Figure 3 shows an example of how this could be done, in which the OpenFlow-enabled switch operates essentially as a patch-panel to allow the packets to reach the NetFPGA. In some cases, the NetFPGA board (a PCI board that plugs into a Linux PC) might be placed in the wiring closet alongside the OpenFlow-enabled switch, or (more likely) in a laboratory.

Chapter 4

Experiment description and Discussion of results: Performance of the system

This chapter will show how the different tests have been done and then it will be discussed the obtained results. Although the main theme of the project is to evaluate OpenFlow technology, it has also been compared with other classic technologies such as software Switching and Routing. Each experiment will have a detailed description of how it has been realized, some graphics with results and finally a discussion explaining and comparing these obtained results.

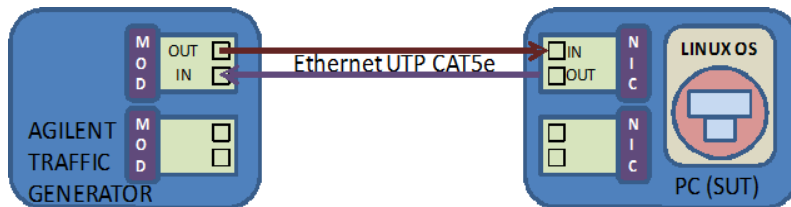


Figure 4.1. Experimental environment

In this project have been made 6 experiments using a synthetic traffic generator device connected to a PC with a Linux Operating System that can act as switch, router or OpenFlow switch (see Figure 4.1). First experiment is a load test in which the 3 technologies have been compared for different packet size and different loads. Second is similar to the first one but introducing different forwarding table sizes. Third is focused in OpenFlow switching analyzing the behaviour of this technology with different combinations of table configurations. Fourth test tries to view the differences when traffic is balanced between OpenFlow forwarding tables. Fifth test compares the 3 technologies in a fairness test in which there are 2 input flows and

only 1 output flow. Finally the sixth test is again focused in OpenFlow technology comparing packet latency according to table size, table type and packet length.

4.1 Test 1: Performance of system according to packet length and load

This test shows the different behaviour in terms of throughput and latency that Switching, Routing and OpenFlow technologies adopt when the systems are subjected to a load test.

4.1.1 Devices

For this test is going to be used the 2 ports of 1 module of the Agilent N2X Traffic Generator and a linux PC with a dual port NIC (see Figure 4.1). One port of the Traffic Generator will be used as a traffic output attached to one input port of the dual NIC through a UTP Ethernet cable. The second port of the dual NIC will be used as output attached to the other input port of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic to the PC, the PC will forward the packet according to the selected technology (switching, routing, OpenFlow) and finally the PC will return the packets to the Agilent tester where the packets will be captured to be analysed.

4.1.2 Test configuration

In this section will be explained how the test has been configured. Due to we are going to test 3 different technologies we need to set 3 different configurations in the linux PC and in the Agilent tester. Following will be described the 3 different configurations:

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 64, 96, 128, 192, 224, 256, 320, 512, 1024, 1120, 1500 Bytes
 - Link Speed: 1 Gbps Ethernet
 - Load: 10, 25, 50, 75, 90, 99 %
 - Time: 60 seconds (each packet size with each load)
- Switching: in the linux PC will be used Bridge-tools to set the layer-2 forwarding of the Kernel. Using Bridge-tools we will configure the 2 ports of the

dual NIC to act as a bridge and to bring up a pseudo-interface for bridging. In the Agilent tester we have only to set the destination MAC address of the packets to be forwarded (see appendix B.1).

- Routing: in the linux PC will be enabled the ip_forwarding feature to set the layer-3 of the Kernel. Using ifconfig we will configure the 2 ports of the dual NIC bringing up these interfaces to act as a router. In the Agilent tester we have to assign IP addresses to the Agilent module ethernet interfaces and to set the destination IP address of the generated packets with the IP of the input port of the module to be forwarded (see appendix B.2).
- OpenFlow: in the linux PC will be enabled the OpenFlow module (OpenFlow version v0.8.9r2). Using the OpenFlow tool DataPathControl (dpctl) we will configure the 2 ports of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add a simple rule in the flow table to forward input packets with a certain destination or source IP address to the output port (interface). In the Agilent tester we have only to send packets with an IP that matches with the entry of the flow table (see appendix B.3).

4.1.3 Results

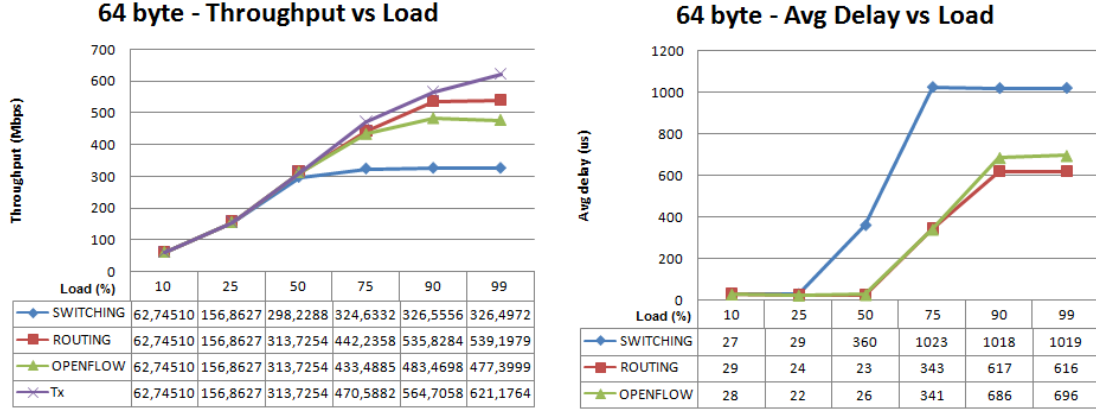


Figure 4.2. 64 byte - Throughput and Average Latency vs Load

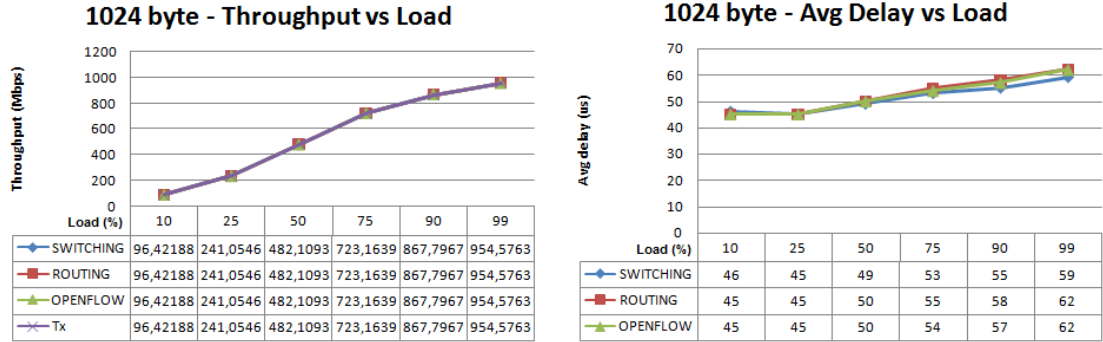


Figure 4.3. 1024 byte - Throughput and Average Latency vs Load

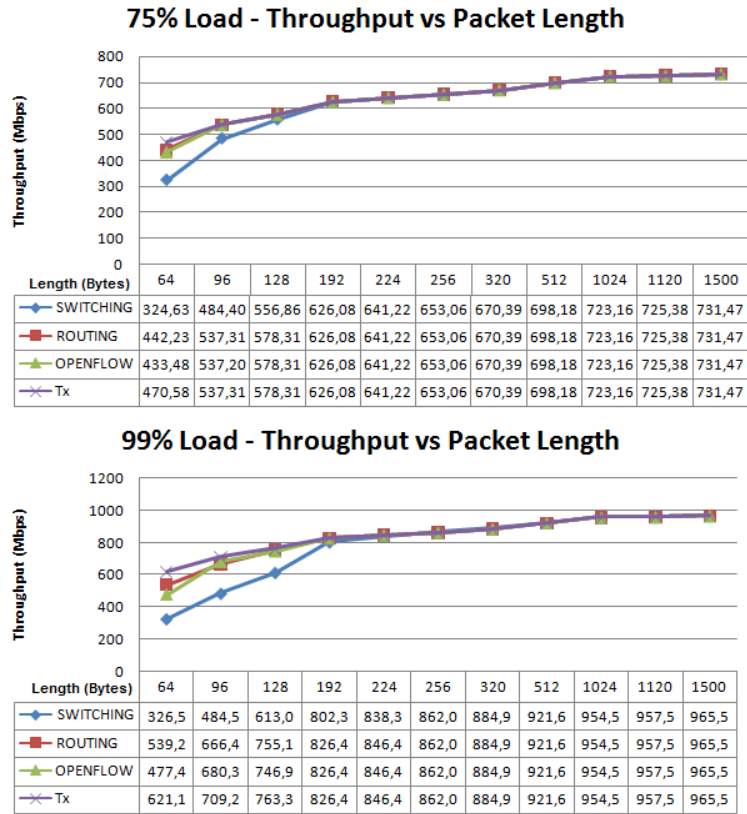


Figure 4.4. Throughput at 75 and 99 % of Load vs Packet Length

4.1.4 Discussion of results

Doing this test we have selected the most significant information so we have selected the results for small (64 byte packets, Figure 4.2) and big packets (1024 byte packets, Figure 4.3) to compare them. We have also selected the evolution of throughput

depending on the packet size at 75% and 99% of load (Figure 4.4). We can observe that sending small packets like in Figure 4.2, Switching technology has worse performance in terms of Throughput than Routing and OpenFlow. This could be due to software Switching implementation and its forwarding method is worse implemented than the other technologies. If we see Average Latency of Figure 4.2 we can also see that Switching has worse performance and as we expected with high loads the average latency increase due to with high rates we have lots of small packets with their headers that have to be processed. If we observe Figure 4.4 we continue observing the bad performance of Switching technology with small packets but when the packets begin to increase their length the results are identical. Furthermore we can see that with higher loads the graphic lines increase depending on the packet size without falls trying to obtain the maximum throughput possible in each case (wire-speed).

4.2 Test 2: Performance of system according to forwarding table size

This test shows the different behaviour in terms of throughput and latency that Switching, Routing and OpenFlow technologies adopt with different forwarding tables size when the systems are subjected to a 10, 40 and 99% load tests.

4.2.1 Devices

For this test is going to be used the 2 ports of 1 module of the Agilent N2X Traffic Generator and a linux PC with a dual port NIC (see Figure 4.1). One port of the Traffic Generator will be used as a traffic output attached to one input port of the dual NIC through a UTP Ethernet cable. The second port of the dual NIC will be used as output attached to the other input port of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic to the PC, the PC will forward the packet according to the selected technology (switching, routing, OpenFlow) and finally the PC will return the packets to the Agilent tester where packets will be captured to be analysed.

4.2.2 Test configuration

In this section will be explained how the test has been configured. Due to we are going to test 3 different technologies we need to set 3 different configurations in the linux PC and in the Agilent tester. Following will be described the 3 different configurations:

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 128, 256, 512, 1024, 1500 Bytes
 - Forwarding table size: 1, 1024, 8192, 64K, 128K
 - Link Speed: 1 Gbps Ethernet
 - Load: 10, 40, 99 %
 - Time: 60 seconds (each packet size with each load)
- Switching: in the linux PC will be used Bridge-tools to set the layer-2 forwarding of the Kernel. Using Bridge-tools we will configure the 2 ports of the dual NIC to act as a bridge and to bring up a pseudo-interface for bridging. Then, before the test is started we have to fill the switching forwarding table sending synthetic traffic from the destination Agilent port. To do this we have to send as many different packets with different destination MAC address as number of entries (size) we need in the forwarding table. Once table filling is done, we just need to configure the Agilent tester to send packets with random source MAC address inside the range of the forwarding entries of the table. For example if we have filled the 64K table, we need to generate 64K random source MAC address packets inside this range (see appendix C.1).
- Routing: in the linux PC will be enabled the ip_forwarding feature to set the layer-3 of the Kernel. Using ifconfig we will configure the 2 ports of the dual NIC bringing up these interfaces to act as a router. Then, before the test is started we have to fill the routing table using the "add route" command defining an output gateway for packets that match with the table entries. In the Agilent tester we have to assign IP addresses to the Agilent module ethernet interfaces and to set randomly the destination IP address of the generated packets with the range of IP networks of the routing table. For example if we have filled the 64K table, we need to generate 64K random destination IP address packets inside this range (see appendix C.2).
- OpenFlow: in the linux PC will be enabled the OpenFlow module (OpenFlow version v0.8.9r2). Using the OpenFlow tool DataPathControl (dpctl) we will configure the 2 ports of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow. To create up to 128K entries we change the UDP source and destination ports combining them in some cases. In the Agilent tester we have to send packets changing randomly

source and destination UDP ports that match with the entries of the flow table (see appendix C.3.3).

4.2.3 Results

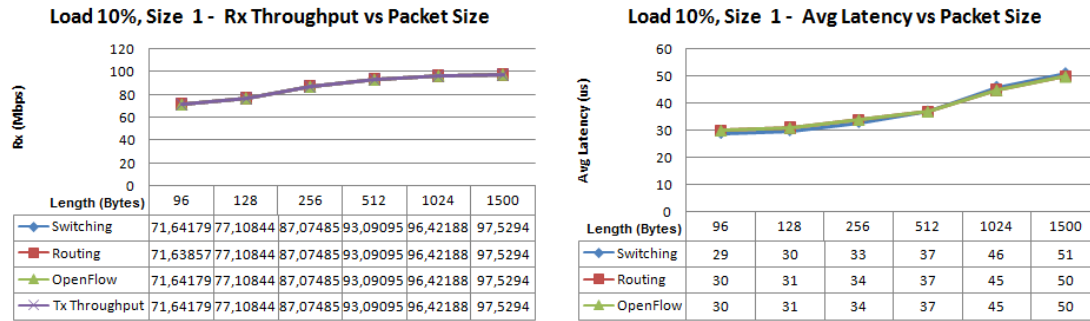


Figure 4.5. Load at 10% with table size 1: Throughput and Average Latency vs Packet Size

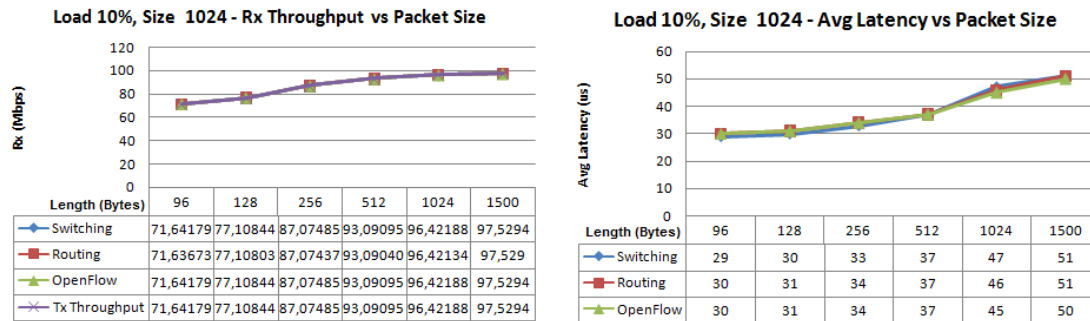


Figure 4.6. Load at 10% with table size 1024: Throughput and Average Latency vs Packet Size

4.2 – Test 2: Performance of system according to forwarding table size

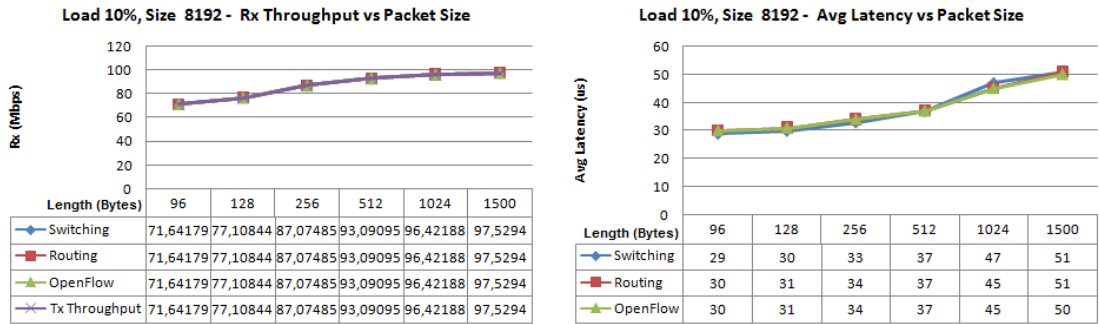


Figure 4.7. Load at 10% with table size 8192: Throughput and Average Latency vs Packet Size

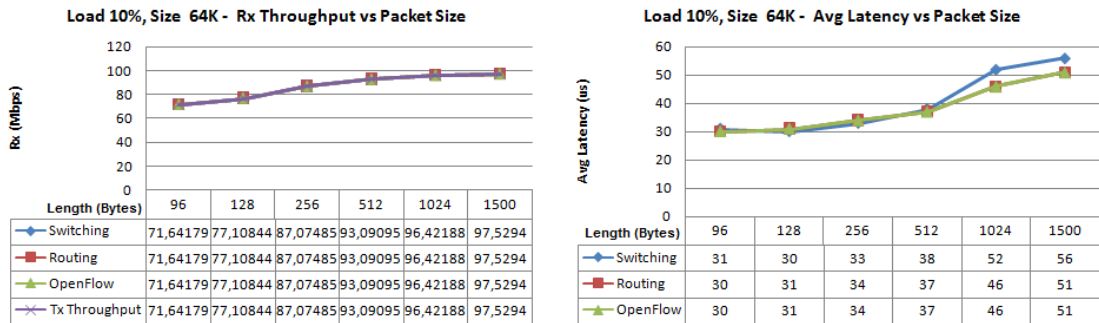


Figure 4.8. Load at 10% with table size 64K: Throughput and Average Latency vs Packet Size

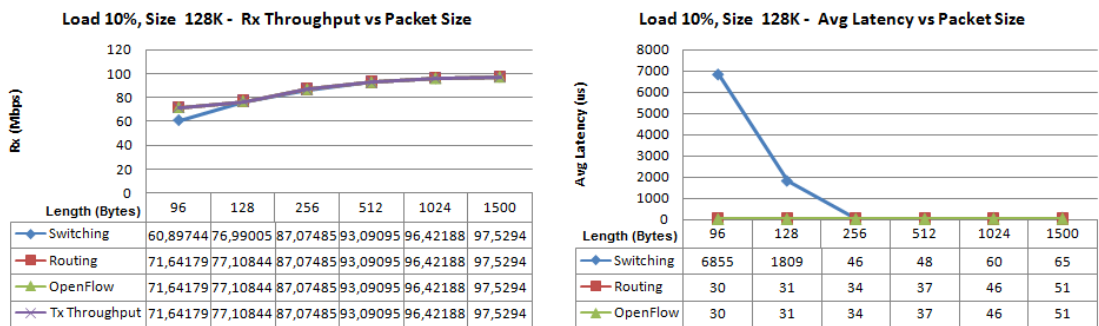


Figure 4.9. Load at 10% with table size 128K: Throughput and Average Latency vs Packet Size

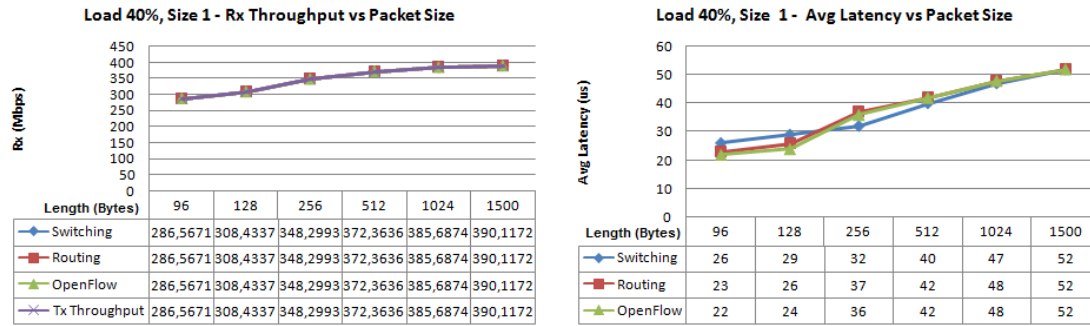


Figure 4.10. Load at 40% with table size 1: Throughput and Average Latency vs Packet Size

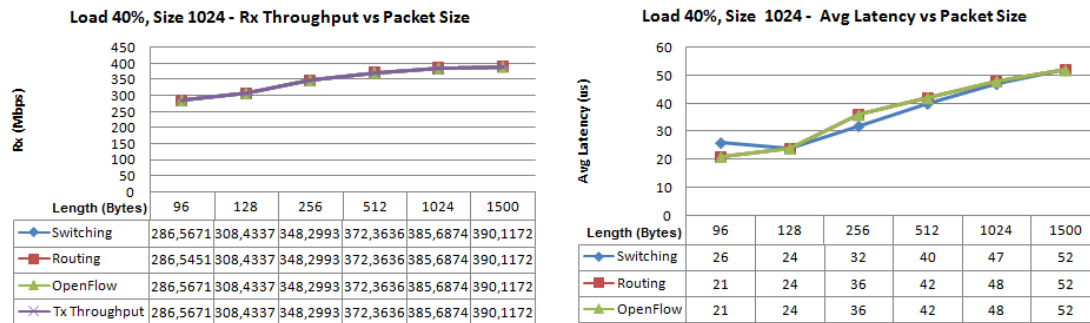


Figure 4.11. Load at 40% with table size 1024: Throughput and Average Latency vs Packet Size

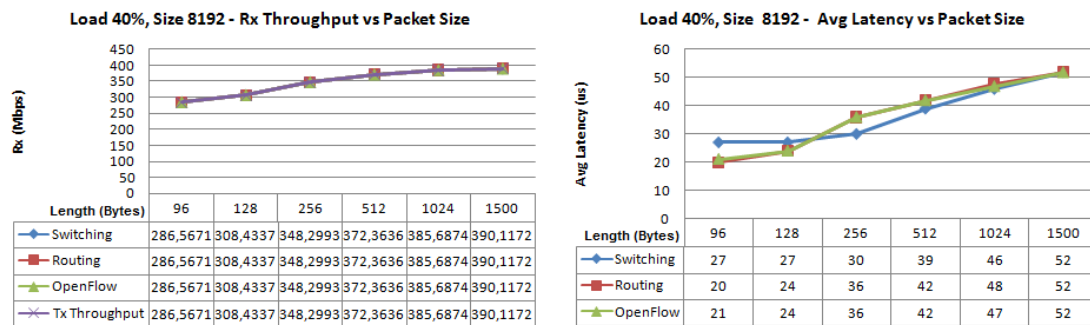


Figure 4.12. Load at 40% with table size 8192: Throughput and Average Latency vs Packet Size

4.2 – Test 2: Performance of system according to forwarding table size

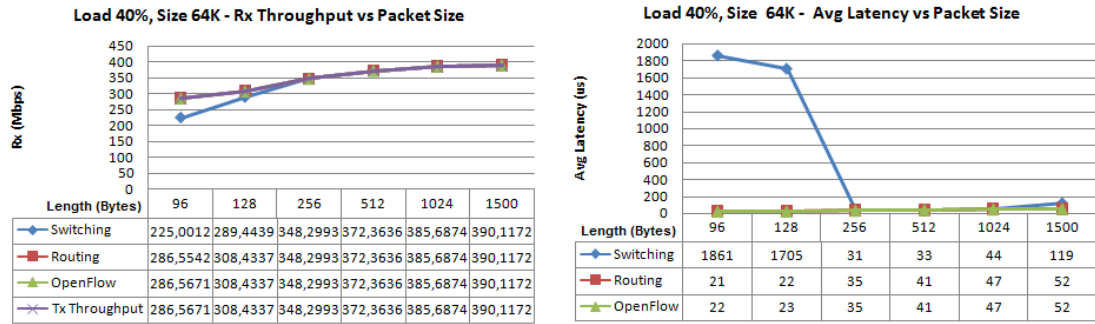


Figure 4.13. Load at 40% with table size 64K: Throughput and Average Latency vs Packet Size

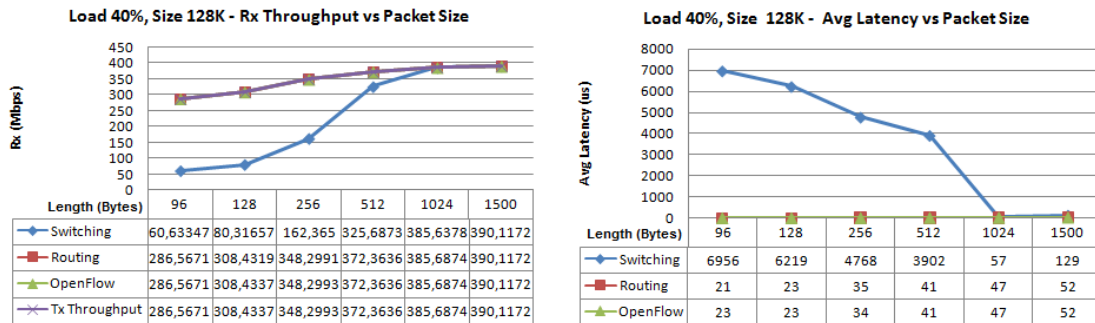


Figure 4.14. Load at 40% with table size 128K: Throughput and Average Latency vs Packet Size

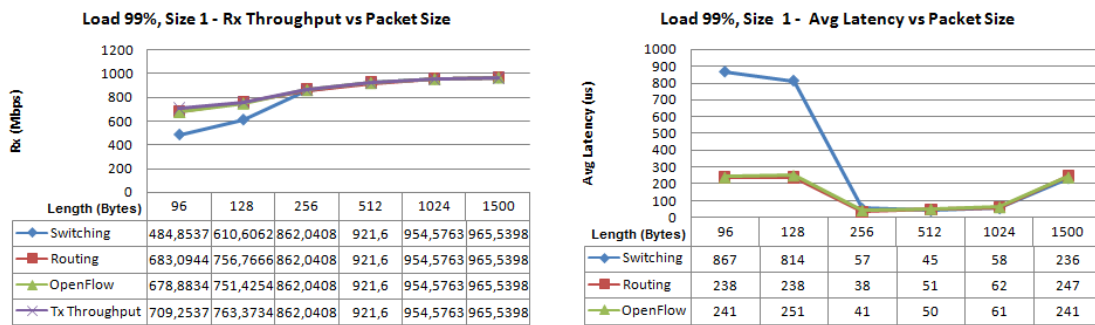


Figure 4.15. Load at 99% with table size 1: Throughput and Average Latency vs Packet Size

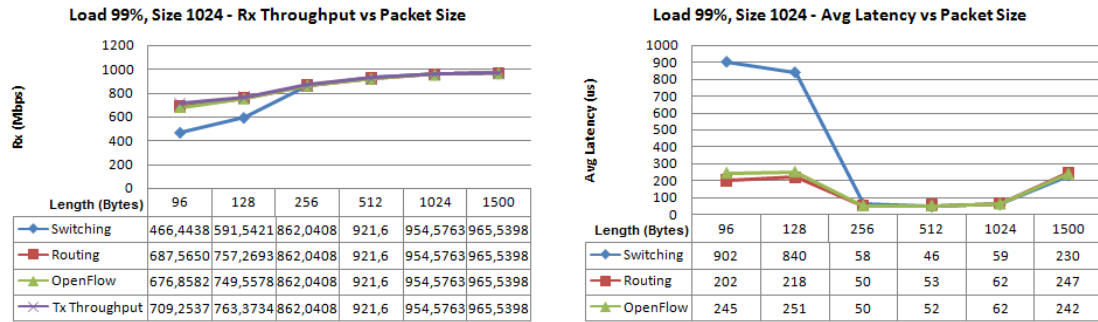


Figure 4.16. Load at 99% with table size 1024: Throughput and Average Latency vs Packet Size

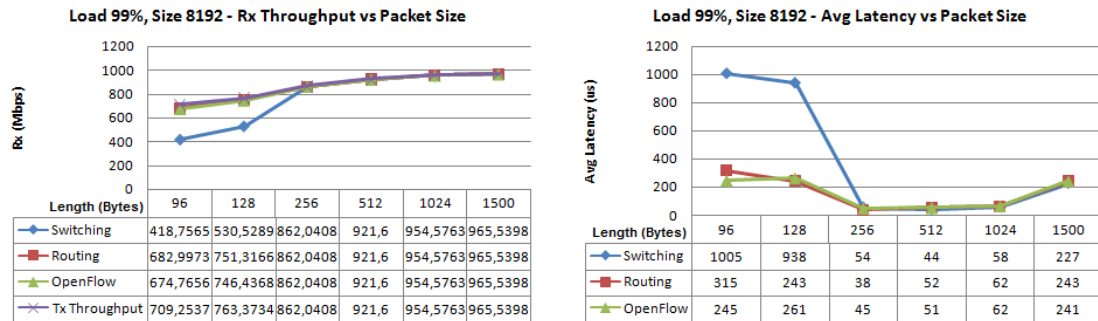


Figure 4.17. Load at 99% with table size 8192: Throughput and Average Latency vs Packet Size

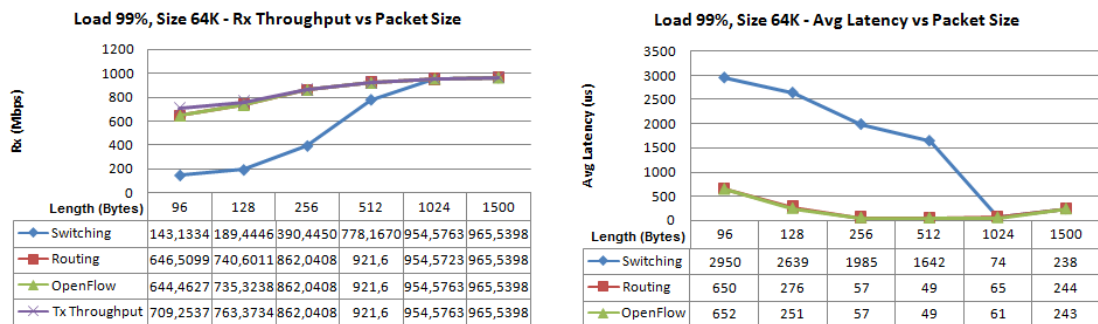


Figure 4.18. Load at 99% with table size 64K: Throughput and Average Latency vs Packet Size

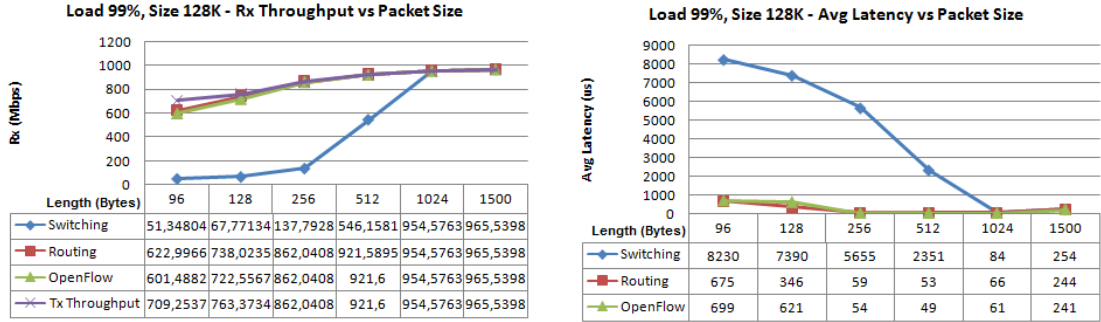


Figure 4.19. Load at 99% with table size 128K: Throughput and Average Latency vs Packet Size

4.2.4 Discussion of results

In this test we have selected graphics for the loads at 10, 40 and 99% and with forwarding table sizes of 1, 1024, 8192, 64K and 128K for Switching, Routing and OpenFlow technologies. In Figures 4.5, 4.6, 4.7, 4.8 and 4.9 we can see that the performance with load at 10% of the different technologies are stable except Switching technology when the forwarding table size is 128K that have worse performance in terms of throughput and average latency. This phenomenon occurs because software switching is not not well developed for large table size. Furthermore both Routing and OpenFlow have an optimal performance for any table size at 10% of load. In Figures 4.10, 4.11, 4.12, 4.13 and 4.14 we can see that the performance with load at 40% of Routing and OpenFlow technologies are stable and without packet loss. But Switching technology when the forwarding tables are too big begins to obtain bad results with small packets. Therefore we can conclude that software Switching is not optimized for big tables and small packets for this intermediate load. In Figures 4.15, 4.16, 4.17, 4.18 and 4.19 we can see the performance with load at 99% and as in the previous tests Switching technology obtain the worst results but now this behaviour starts increasing from small table sizes. This is due to the system is in overload and the code is not optimized for small packets and large forwarding tables. If we observe the performance of Routing and OpenFlow we can see that both technologies are in overload but their results are similar.

4.3 Test 3: Performance of OpenFlow according to forwarding table configurations

This test shows the different behaviours in terms of throughput and latency that OpenFlow technology adopts with different sizes and types of flow table when the

systems are subjected to a 10, 40 80, 90 and 99% load tests. In this test are used the 2 types of internal flowtables: the linear table and the exact-match table (hash-table).

4.3.1 Devices

For this test is going to be used the 2 ports of 1 module of the Agilent N2X Traffic Generator and a linux PC with a dual port NIC (see Figure 4.1). One port of the Traffic Generator will be used as a traffic output attached to one input port of the dual NIC through a UTP Ethernet cable. The second port of the dual NIC will be used as output attached to the other input port of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic to the PC, the PC will forward the packet according to the OpenFlow flow-table technology and finally the PC will return the packets to the Agilent tester where packets will be captured to be analysed.

4.3.2 Test configuration

In this section will be explained how the test has been configured. We are going to test only OpenFlow technology with different table size and type configurations in the linux PC. Is going to be tested how OpenFlow responds with only linear table, with only hash table and when combines linear and hash tables. Following it will be described how to configure this flow-tables:

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 128, 256, 512, 1024, 1500 Bytes
 - Forwarding table size: Linear: 100 / Hash: 128K /
Combining: Hash96K_Linear100, Hash128K_Linear100
 - Link Speed: 1 Gbps Ethernet
 - Load: 10, 40, 80, 90, 99 %
 - Time: 60 seconds (each packet size with each load)
- OpenFlow: in the linux PC will be enabled the OpenFlow module (OpenFlow version v0.8.9r2). Using the OpenFlow tool DataPathControl (dpctl) we will configure the 2 ports of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow and fills the hash

table or wildcard entries that fills the linear table. Following will be described how to create the 4 table configurations considered for this test:

- Linear 100 entries: to create up to 100 entries in the linear table of the linux PC we have only to define some field of the flow definition and change it 100 times. In our case we have defined 100 different IP addresses. In the Agilent tester we have to send packets changing the destination IP address that match with the entries of the linear flow table (see appendix C.3.1).
- Hash 128K entries: to create up to 128K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them. In the Agilent tester we have to send packets changing randomly source and destination UDP ports that match with the entries of the hash flow table (see appendix C.3.3).
- Hash 96K and Linear 100 entries: to create up to 96K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them to generate 96K flow definitions. To create up to 100 entries in the linear table we have only to define some field (destination IP address) of the flow definition and change it 100 times. We have also to be care that the 100 IP addresses of the linear table are different to the fixed IP address of the hash entries (remember that in this hash table does not change the IP, but the UDP port). In the Agilent tester we have to send packets changing randomly the destination IP address within the range of the 100 destination IP address defined in the linear table and the fixed destination IP address of the hash table. Also we have to change randomly source and destination UDP ports that match with the entries of the hash flow table (the linear table is not affected due to it filters flows only by destination IP). With this configuration traffic goes 1/101 to the Hash Table and 100/101 to the Linear Table, hence traffic is balanced 99% to Linear Table but is enough to see the influence of the 2 tables working together (see appendix C.3.4).
- Hash 128K and Linear 100 entries: to create up to 128K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them to generate 128K flow definitions. To create up to 100 entries in the linear table we have only to define some field (destination IP address) of the flow definition and change it 100 times. We have also to be care that the 100 IP addresses of the linear table are different to the fixed IP address of the hash entries (remember that in this hash table does not change the

IP, but the UDP port). In the Agilent tester we have to send packets changing randomly the destination IP address within the range of the 100 destination IP address defined in the linear table and the fixed destination IP address of the hash table. Also we have to change randomly source and destination UDP ports that match with the entries of the hash flow table (the linear table is not affected due to it filters flows only by destination IP). With this configuration traffic goes 1/101 to the Hash Table and 100/101 to the Linear Table, hence traffic is balanced 99% to Linear Table but is enough to see the influence of the 2 tables working together (see appendix C.3.5).

4.3.3 Results

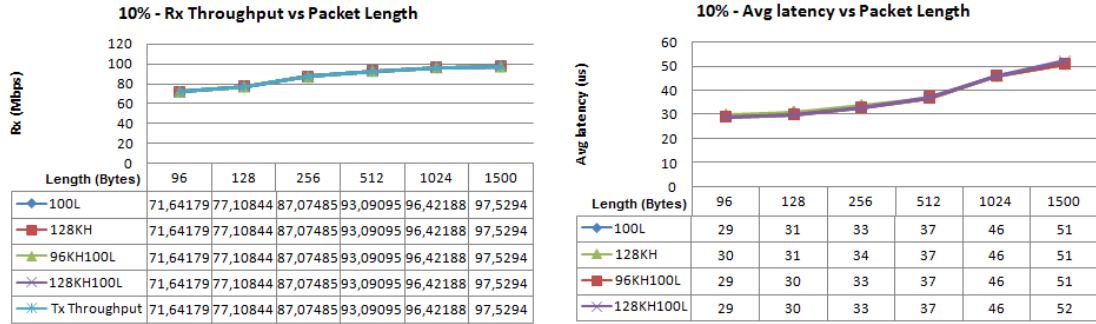


Figure 4.20. OpenFlow flow-tables: Load 10% - Throughput and Average Latency vs Packet Length

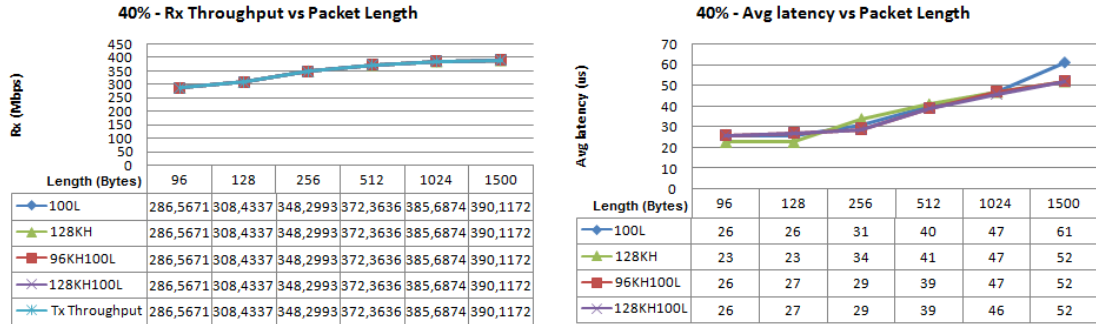


Figure 4.21. OpenFlow flow-tables: Load 40% - Throughput and Average Latency vs Packet Length

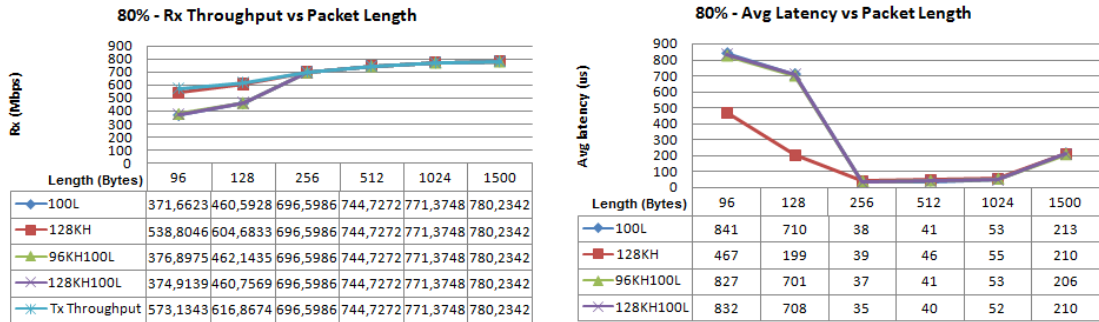


Figure 4.22. OpenFlow flow-tables: Load 80% - Throughput and Average Latency vs Packet Length

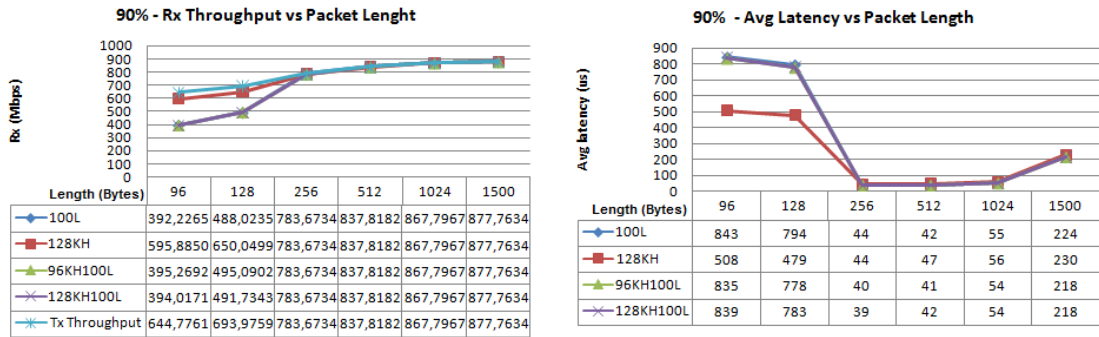


Figure 4.23. OpenFlow flow-tables: Load 90% - Throughput and Average Latency vs Packet Length

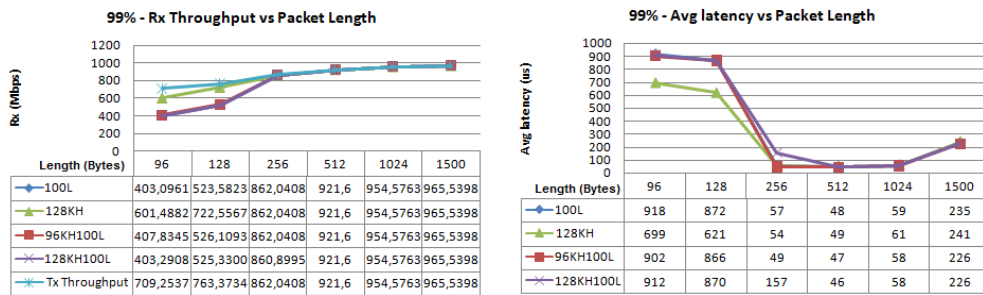


Figure 4.24. OpenFlow flow-tables: Load 99% - Throughput and Average Latency vs Packet Length

4.3.4 Discussion of results

In this test for OpenFlow technology we have selected graphics for the loads at 10, 40, 80, 90 and 99% and with different types and sizes of flow table (linear with 100 entries, hash with 128K entries and combination of linear with 100 and hash with 96K entries and combination of linear with 100 and hash with 128K entries). Seeing the different configurations for small loads at 10 and 40% (Figures 4.20 and 4.21) we note that results are similar due to the system is not in overload and can process all packet leghts. But when we increase the load at 80, 90 and 99% (Figures 4.22, 4.23 and 4.24) we can see as we expected that only hash table configuration has better results than only linear and combining configurations. With this high loads we can see that when packets are larger than 128 Bytes the system begins to perform at line-rate, so the PC is not the bottleneck. This is because search matchings in hash table is faster due to the HASH algorithm. Also we can see sensibly better results for combining configuration than only linear configuration.

4.4 Test 4: Performance of OpenFlow balancing traffic between table types

This test uses OpenFlow technology to check the performance of the system when is used both flow-tables linear and hash and we balance traffic to achieve matchings in them. The test is divided in two parts. The first part is a test that balances traffic only to one of the two tables to compare the performance in terms of throughput. The second part balances traffic in different percentages to both tables.

4.4.1 Devices

For this test is going to be used the 2 ports of 1 module of the Agilent N2X Traffic Generator and a linux PC with a dual port NIC (see Figure 4.1). One port of the Traffic Generator will be used as a traffic output attached to one input port of the dual NIC through a UTP Ethernet cable. The second port of the dual NIC will be used as output attached to the other input port of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic to the PC, the PC will forward the packet according to the OpenFlow flow-table technology and finally the PC will return the packets to the Agilent tester where packets will be captured to be analysed.

4.4.2 Test configuration

In this section will be explained how the test has been configured. We are going to test only OpenFlow technology with different table size and type configurations in the linux PC. Is going to be tested how OpenFlow responds when combines linear and hash tables. As we have said in the introduction of this test, we are going to divide the test in two parts, so there will be two different configurations. Following it will be described how to configure these environments.

Configuration: OpenFlow - balancing traffic only to one table type

In this configuration it will be set both linear and hash tables but traffic will be sended only to one. Also, we are going to show the results when traffic is balanced randomly to both tables (values from Test 3) to compare with the obtained results.

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 128, 256, 512, 1024, 1500 Bytes
 - Forwarding table size: Combining: Hash96K_Linear100, Hash128K_Linear100
 - Link Speed: 1 Gbps Ethernet
 - Load: 10, 40, 99
 - Time: 60 seconds (each packet size with each load)
- OpenFlow: in the linux PC will be enabled the OpenFlow module (OpenFlow version v0.8.9r2). Using the OpenFlow tool DataPathControl (dpctl) we will configure the 2 ports of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow and fills the hash table or wildcard entries that fills the linear table. Following will be described how to create the 2 table configurations considered for this test:
 - Hash 96K and Linear 100 entries: to create up to 96K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them to generate 96K flow definitions. To create up to 100 entries in the linear table we have only to define some field (destination IP address) of the flow definition and change it 100 times. We have also to be care that the 100 IP

addresses of the linear table are different to the fixed IP address of the hash entries (remember that in this hash table does not change the IP, but the UDP port). In the Agilent tester we have to send packets only to one table, so we need to send random traffic that matches with the linear table (changing destination IP address within the range set in the flow definition) or to send random traffic that matches with the hash table (sending exact flows changing only destination and source UDP port within the range set in the flow definition) see appendix C.3.4).

- Hash 128K and Linear 100 entries: to create up to 128K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them to generate 128K flow definitions. To create up to 100 entries in the linear table we have only to define some field (destination IP address) of the flow definition and change it 100 times. We have also to be care that the 100 IP addresses of the linear table are different to the fixed IP address of the hash entries (remember that in this hash table does not change the IP, but the UDP port). In the Agilent tester we have to send packets only to one table, so we need to send random traffic that matches with the linear table (changing destination IP address within the range set in the flow definition) or to send random traffic that matches with the hash table (sending exact flows changing only destination and source UDP port within the range set in the flow definition) (see appendix C.3.5).

Configuration: OpenFlow - balancing traffic to both table types

In this configuration it will be set both linear and hash tables and traffic will be sended balanced in different percentages to both tables.

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic with the following pattern using 2 streams to send traffic to each table (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 1024 Bytes
 - Forwarding table size: Combining: Hash96K_Linear100, Hash128K_Linear100
 - Link Speed: 1 Gbps Ethernet
 - Load Balancing (% Linear-Hash): (Total=99%) 0-99, 10-89, 30-69, 49'5-49'5, 69-30, 89-10, 99-0 %
 - Time: 60 seconds (each packet size with each load)

- OpenFlow: in the linux PC will be enabled the OpenFlow module. Using the OpenFlow tool DataPathControl (dpctl) we will configure the 2 ports of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow and fills the hash table or wildcard entries that fills the linear table. Following will be described how to create the 2 table configurations considered for this test:
 - Hash 96K and Linear 100 entries: to create up to 96K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them to generate 96K flow definitions. To create up to 100 entries in the linear table we have only to define some field (destination IP address) of the flow definition and change it 100 times. We have also to be care that the 100 IP addresses of the linear table are different to the fixed IP address of the hash entries (remember that in this hash table does not change the IP, but the UDP port). In the Agilent tester we have to send packets to both tables creating two stream flows at different loads, so we need a stream flow to send random traffic that matches with the linear table (changing destination IP address within the range set in the flow definition) and another stream flow to send random traffic that matches with the hash table (sending exact flows changing only destination and source UDP port within the range set in the flow definition) (see appendix C.3.4).
 - Hash 128K and Linear 100 entries: to create up to 128K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them to generate 128K flow definitions. To create up to 100 entries in the linear table we have only to define some field (destination IP address) of the flow definition and change it 100 times. We have also to be care that the 100 IP addresses of the linear table are different to the fixed IP address of the hash entries (remember that in this hash table does not change the IP, but the UDP port). In the Agilent tester we have to send packets to both tables creating two stream flows at different loads, so we need a stream flow to send random traffic that matches with the linear table (changing destination IP address within the range set in the flow definition) and another stream flow to send random traffic that matches with the hash table (sending exact flows changing only destination and source UDP port within the range set in the flow definition) (see appendix C.3.5).

4.4.3 Results

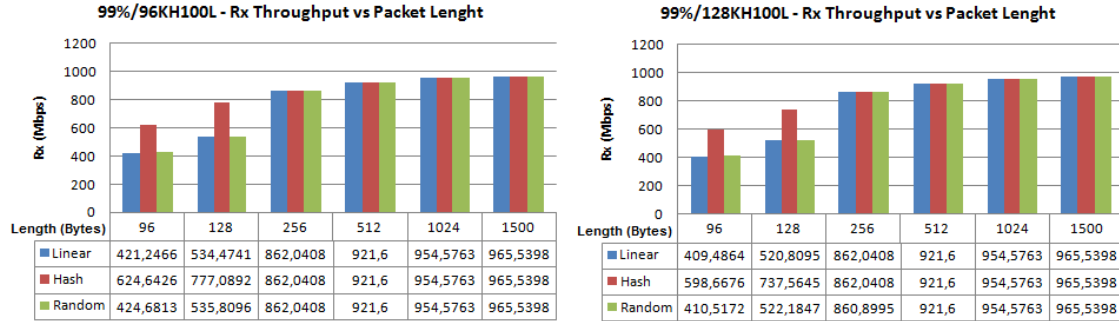


Figure 4.25. OpenFlow tables: Throughput vs Packet Length balancing traffic to one table

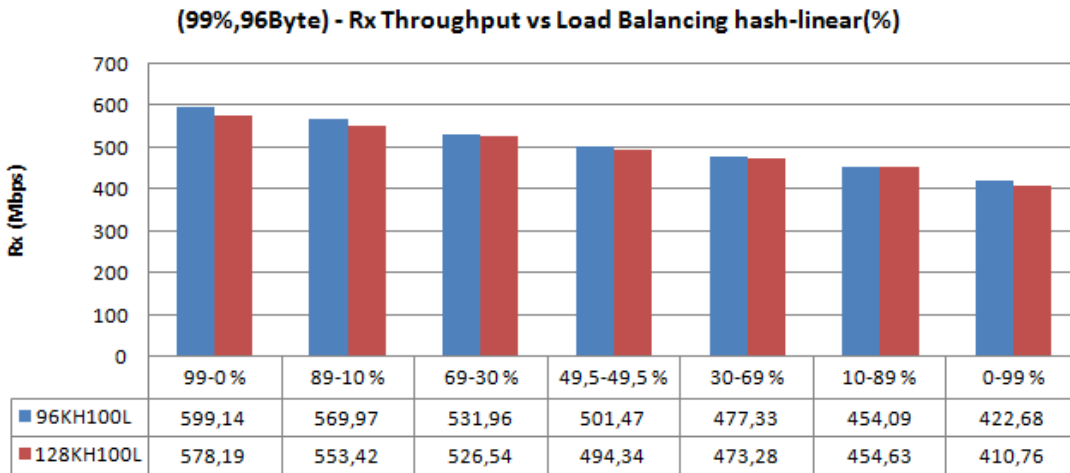


Figure 4.26. OpenFlow tables: 96 Byte - Throughput vs Load balancing to both tables

4.4.4 Discussion of results

As we have explained before this test is separated in 2 parts. Figure 4.25 corresponds to part 1 and we can see the performance of the system when we balance traffic to one of the 2 types of table. We only consider relevant when the system is overloaded at 99% because with other loads (10 and 40 %) the results are similar for all configurations. In Figure 4.25 we can see that with small packets sending traffic only to hash table has better performance that sending only to linear table.

This is due to searching hash algorithm is faster than linear methods. Also we can observe that comparing the two graphics, with small packets the configuration with the small hash table (96K entries) is a bit faster than the big one (128K entries). If we compare with the random configuration we can see that results are similar to linear due to load balancing percentage as we have explained in test 3 (1/101 to hash, 100/101 to linear).

Figure 4.26 corresponds to part 2 and we can see the performance of the system when we balance traffic to both table types. We only consider relevant when is sended 96 Byte packets. In Figure 4.26 we can see as we expected that performance is better when we balance most of the traffic to the hash table. This is due to hash algorithm is more efficient than linear one. If we observe the shape of the graphic we can see a decrease of near 30% between balancing traffic all to Hash (first column) or balancing all to linear (last column).

4.5 Test 5: Performance of system subject to a fairness test

The fairness test is a good way to see which is the behaviour of a system when more than one flow (in this test 2 flows) are injected to forward in the network element. We are going to do the fairness test in 2 configurations for all three technologies Switching, Routing and OpenFlow. The first configuration is sending 2 flows to the same input NIC port, process them in the linux PC and then return them through the output interface to the Agilent tester. The second configuration is to create 2 flows from independent ports of the Agilent tester and send them through separate cables to 2 different input NIC ports in the Linux PC, process them and finally return both flows through only one NIC port to the Agilent tester.

4.5.1 Devices

Configuration 1: 1 input - 1 output

For this configuration is going to be used the 2 ports of 1 module of the Agilent N2X Traffic Generator and a linux PC with a dual port NIC (see Figure 4.1). One port of the Traffic Generator will be used as a traffic output attached to one input port of the dual NIC through a UTP Ethernet cable. The second port of the dual NIC will be used as output attached to the other input port of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic to the PC, the PC will forward the packet according to the selected technology (switching, routing, OpenFlow) and finally the PC will return the packets to the Agilent tester where packets will be captured to be analysed.

Configuration 2: 2 inputs - 1 output

For this configuration is going to be used the 2 ports of 1 module for sending traffic and 1 port of another module for receiving traffic of the Agilent N2X Traffic Generator and a linux PC with 2 dual port NIC (see Figure 4.27). Two ports of one module of the Traffic Generator will be used as a traffic output generators attached to two input ports of the dual NIC through two UTP Ethernet cables. A port of the other dual NIC will be used as output attached to the other input port of the second module of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic with 2 output ports to the PC, the PC will process and forward the packets according to the selected technology (switching, routing, OpenFlow) and finally the PC will return the packets to the Agilent tester where packets will be captured to be analysed.

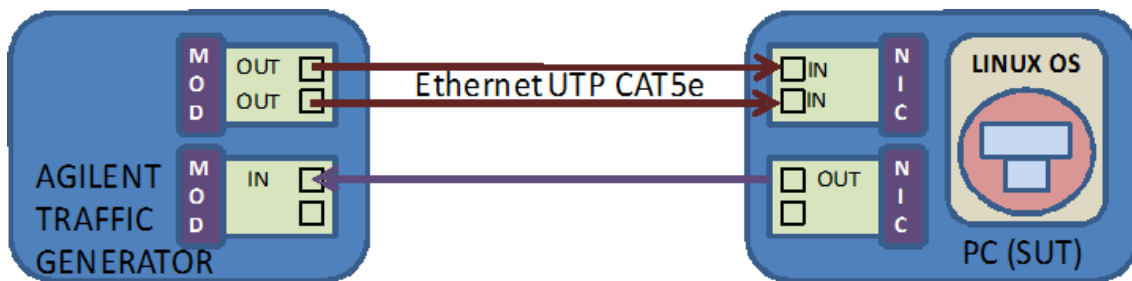


Figure 4.27. Fairness test configuration for 2 inputs and 1 output interfaces

4.5.2 Test configuration

The fairness test will be done using 2 separated but identical flows. One flow will have fixed line rate and the other one will be variable increased until the system is overloaded. As we have explained previously we are going to test 2 configurations, hence the parameters of the test will be different.

Configuration 1: 1 input - 1 output

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic through 1 output port with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 1024Bytes
 - Forwarding table size: 1024, 64K
 - Link Speed: 1 Gbps Ethernet

- Number of Agilent modules and ports: 1 module -> 2 ports (input - output)
 - Load (Fixed-Variable Mbps): 96 Byte-> 100-400, 100-500, 100-600, 100-827 Mbps / 1024 Byte -> 300-500, 300-600, 300-700, 300-980 Mbps
 - Time: 60 seconds (each packet size with each load)
-
- Switching: in the linux PC will be used Bridge-tools to set the layer-2 forwarding of the Kernel. Using Bridge-tools we will configure the 2 ports (1 input, 1 output) of the dual NIC to act as a bridge and to bring up a pseudo-interface for bridging. Then, before the test is started we have to fill the switching forwarding table sending synthetic traffic from the destination Agilent port. To do this we have to send as many different packets with different destination MAC address as number of entries (size) we need in the forwarding table. Once table filling is done, we just need to configure the Agilent tester to send packets with random source MAC address inside the range of the forwarding entries of the table. For example if we have filled the 64K table, we need to generate 64K random source MAC address packets inside this range (see appendix C.1).
 - Routing: in the linux PC will be enabled the `ip_forwarding` feature to set the layer-3 of the Kernel. Using `ifconfig` we will configure the 2 ports (1 input, 1 output) of the dual NIC bringing up these interfaces to act as a router. Then, before the test is started we have to fill the routing table using the "add route" command defining an output gateway for packets that match with the table entries. In the Agilent tester we have to assign IP addresses to the Agilent module ethernet interfaces and to set randomly the destination IP address of the generated packets with the range of IP networks of the routing table. For example if we have filled the 64K table, we need to generate 64K random destination IP address packets inside this range (see appendix C.2).
 - OpenFlow: in the linux PC will be enabled the OpenFlow module (OpenFlow version v0.8.9r2). Using the OpenFlow tool DataPathControl (`dpctl`) we will configure the 2 ports (1 input, 1 output) of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow. To create up to 64K entries we change the UDP source port. In the Agilent tester we have to send packets changing randomly the source UDP port that match with the entries of the flow table (see appendix C.3.2).

Configuration 2: 2 inputs - 1 output

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic through 2 output ports with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 1024Bytes
 - Forwarding table size: 1024, 64K
 - Link Speed: 1 Gbps Ethernet
 - Number of Agilent modules and ports: 2 module → 3 ports (2 input - 1 output)
 - Load (Fixed-Variable Mbps): 96 Byte → 100-400, 100-500, 100-600, 100-827 Mbps / 1024 Byte → 300-500, 300-600, 300-700, 300-980 Mbps
 - Time: 60 seconds (each packet size with each load)
- Switching: in the linux PC will be used Bridge-tools to set the layer-2 forwarding of the Kernel. Using Bridge-tools we will configure the 3 ports (2 inputs, 1 output) of the 2 dual NICs to act as a bridge and to bring up a pseudo-interface for bridging. Then, before the test is started we have to fill the switching forwarding table sending synthetic traffic from the destination Agilent port. To do this we have to send as many different packets with different destination MAC address as number of entries (size) we need in the forwarding table. Once table filling is done, we just need to configure the Agilent tester to send packets with random source MAC address inside the range of the forwarding entries of the table. For example if we have filled the 64K table, we need to generate 2 flows of 64K random source MAC address packets inside this range from the 2 output ports of the Agilent tester (see appendix D.1).
- Routing: in the linux PC will be enabled the `ip_forwarding` feature to set the layer-3 of the Kernel. Using `ifconfig` we will configure the 3 ports (2 inputs, 1 output) of the 2 dual NIC bringings up these interfaces to act as a router. Then, before the test is started we have to fill the routing table using the "add route" command defining an output gateway (through the output interface) for packets that match with the table entries. In the Agilent tester we have to assign IP addresses to the Agilent module ethernet interfaces and to set randomly the destination IP address of the generated packets with the range of IP networks of the routing table. For example if we have filled the 64K table, we need to generate 2 flows of 64K random destination IP address packets inside this range from the 2 output ports of the Agilent tester (see appendix D.2.1 and appendix D.2.2).

- OpenFlow: in the linux PC will be enabled the OpenFlow module. Using the OpenFlow tool DataPathControl (dpctl) we will configure the 3 ports (2 inputs, 1 output) of the 2 dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow. To create up to 64K entries we change the UDP source port. In the Agilent tester we have to send packets changing randomly the source UDP port that match with the entries of the flow table from the 2 output ports of the Agilent tester (see appendix D.3.1 and appendix D.3.2).

4.5.3 Results

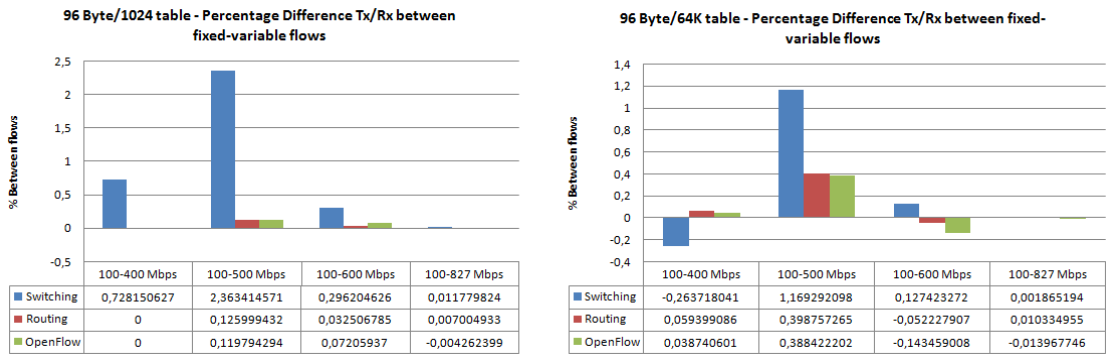


Figure 4.28. Fairness test (1 input - 1 output): Percentage difference Tx/Rx between fixed-variable flows

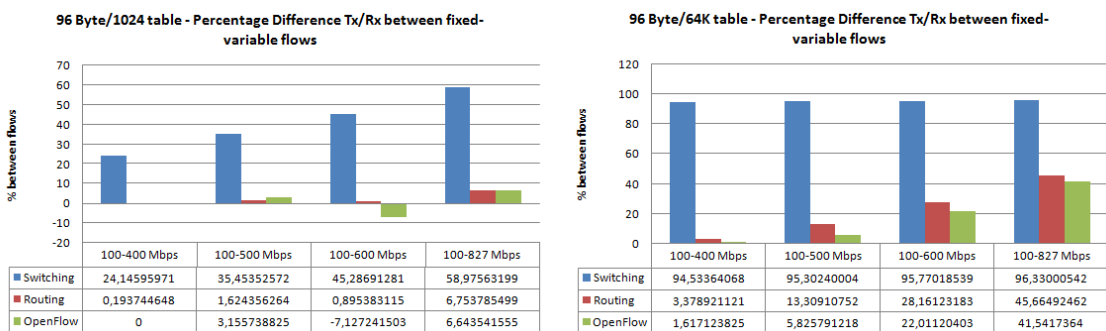


Figure 4.29. Fairness test (2 inputs - 1 output): Percentage difference Tx/Rx between fixed-variable flows

4.5.4 Discussion of results

As we have explained the fairness test has been done with 2 different hardware configurations. First of all we are going to define what is the meaning of fairness in this context. We assume that a system is fairness when the difference in Tx/Rx ratio (percentage) between 2 flows (in our case, the fixed and the variable flow) is close to 0 (see example Table 4.1). Figure 4.28 corresponds to the test with 1 input port and 1 output port in the linux PC. We have considered only interesting results for 96 byte packets. We can see that with this configuration the scheduler acts fairly for both flows. The difference in terms of percentage are close to 0 although we can see a small peak of +2'36% for 1024-entry table and +1'17% for 64K-entry table for Switching technology. So we can conclude that this configuration is fair.

	Tx Throughput (Mbps)	Rx Throughput (Mbps)	Rx/Tx ratio (%)
FLOW A	100	95	95
FLOW B	300	275	91,67
Difference Rx/Tx ratio between FlowA and FlowB (%):			3,33

Table 4.1. Example of fairness ration between two flows

Figure 4.29 corresponds to the test with 2 input ports and 1 output port in the linux PC. We have considered only interesting results for 96 byte packets. In this configuration Linux OS has to schedule the input packets from the 2 input ports and uses a Deficit Round Robin scheduler (DRR) maintaining MAX-MIN fairness between the big and small flows. Seeing the graphics we can see that Switching technology protects the small-rate flow and drops packets from the big-rate flow. Routing and OpenFlow have similiar behaviour although OpenFlow is more fair. This two technologies begin to be unfair in overload particularly with the 64K-entry tables.

4.6 Test 6: OpenFlow packet latency according to table size, table type and packet length

This test tries to measure in OpenFlow technology the influence of flow-table size and type (linear or hash) and packet length in processing latency of packets. The aim of this test is to know which is the OpenFlow processing time for matching input flows and forward them (see Figure 4.30 and Formula).

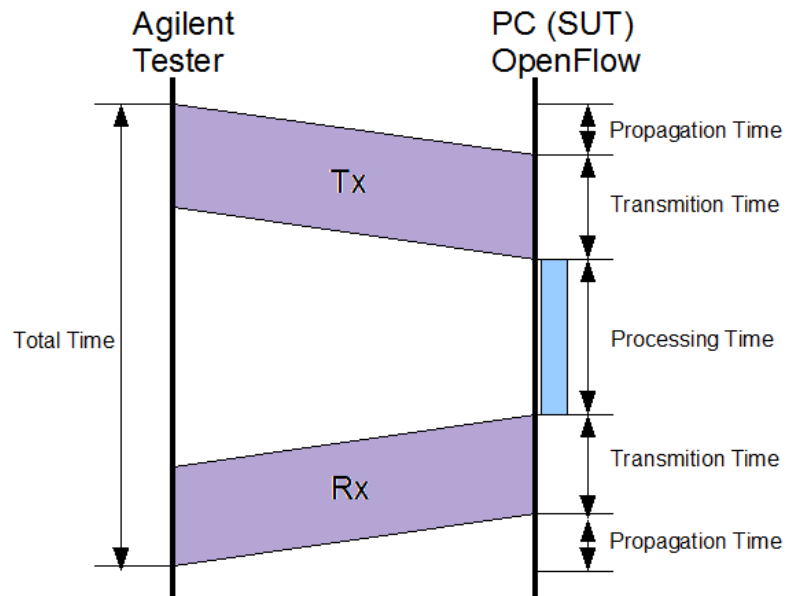


Figure 4.30. Packet transmission between Agilent tester and OpenFlow PC

$$ProcessingTime = TotalTime - (2TransmissionTime + 2PropagationTime)$$

TotalTime :obtained experimentally

$$TransmissionTime = \frac{PacketLength \times 8}{TransmissionRate}$$

TransmissionRate = 1Gbps

PropagationTime \cong 0 (negligible due to the short length of cable)

4.6.1 Devices

For this test is going to be used the 2 ports of 1 module of the Agilent N2X Traffic Generator and a linux PC with a dual port NIC (see Figure 4.1). One port of the Traffic Generator will be used as a traffic output attached to one input port of the dual NIC through a UTP Ethernet cable. The second port of the dual NIC will be

used as output attached to the other input port of the Traffic Generator with other UTP Ethernet cable. Hence, the Agilent will send synthetic traffic to the PC, the PC will forward the packet according to the OpenFlow flow-table technology and finally the PC will return the packets to the Agilent tester where packets will be captured to be analysed.

4.6.2 Test configuration

In this section will be explained how the test has been configured. We are going to test only OpenFlow technology with different table size and type configurations in the linux PC. Is going to be tested how OpenFlow responds in terms of latency with only linear table and with only hash table. Following it will be described how to configure this flow-tables:

- Agilent tester: in the Traffic Generator device we will configure a test that sends traffic with the following pattern (Note: each packet size is tested during 60 seconds at each load):
 - Packet size: 96, 128, 256, 512, 1024, 1500 Bytes
 - Forwarding table size: Linear: 1, 25, 50, 75, 100 / Hash: 1, 32K, 64K, 96, 128K
 - Link Speed: 1 Gbps Ethernet
 - Load: 10 %
 - Time: 60 seconds (each packet size with each load)
- OpenFlow: in the linux PC will be enabled the OpenFlow module (OpenFlow version v0.8.9r2). Using the OpenFlow tool DataPathControl (dpctl) we will configure the 2 ports of the dual NIC bringing up these interfaces to act as an OpenFlow switch. Also we have to add rules in the flow table to forward input packets to the output port (interface). To fill this flow table we create exact match entries that defines all possible fields of a flow and fills the hash table or wildcard entries that fills the linear table. Following will be described how to create the 4 table configurations considered for this test:
 - Linear up to 100 entries: to create up to 100 entries in the linear table of the linux PC we have only to define some field of the flow definition and change it 100 times. In our case we have defined 100 different IP addresses. In the Agilent tester we have to send packets changing the destination IP address that match with the entries of the linear flow table.

- Hash up to 128K entries: to create up to 96K or 128K entries in the hash table of the linux PC we fill all the fields of the flow definition changing the UDP source and destination ports combining them (Up to 64K we need only to change UDP source port). In the Agilent tester we have to send packets changing randomly source and destination UDP ports that match with the entries of the hash flow table.

4.6.3 Results

Packet length (Bytes)	Linear Table (us)				
	1	25	50	75	100
96	28,176	28,176	28,176	27,176	27,176
128	28,664	28,664	28,664	28,664	27,664
256	29,616	29,616	28,616	28,616	28,616
512	28,52	28,52	28,52	28,52	28,52
1024	28,328	29,328	29,328	29,328	29,328
1500	25,712	26,712	26,712	26,712	26,712
Packet length (Bytes)	Hash Table (us)				
	1	32K	64K	96K	128K
96	28,176	28,176	28,176	28,176	28,176
128	28,664	28,664	28,664	28,664	28,664
256	29,616	29,616	29,616	29,616	29,616
512	28,52	28,52	28,52	28,52	28,52
1024	28,328	28,328	28,328	29,328	29,328
1500	25,712	26,712	26,712	26,712	26,712

Table 4.2. OpenFlow packet latency for different table types and sizes

4.6.4 Discussion of results

In this test we have not obtained the results that we expected. We expected that processing time for packets with destination hash table were less (in time) than packets with destination linear table due to hash algorithm is faster and more efficient than linear searching. We also expected that hash table results were similar independently of table size and that linear table results increase linearly in time with the table entry increasing. If we observe Table 4.2 we can see that latency for the same packet size are similar for both Linear Table or Hash Table with different table sizes. We attribute these results due to the Agilent tester sends thousand of packets during the 60 seconds test and the system uses caching and knows how to process

the packets. So finally we obtain the average latency for packets that have been processed using caching methods. One solution to this problem is trying to understand how to delete the cache each time we send a packet to compute the average latency of a significant number of packets without caching.

Chapter 5

Conclusions

This chapter will explain the conclusions extracted after doing this thesis. We have achieved all the goals we set at the beginning of the project which were study and understand OpenFlow technology to then test it against other forwarding technologies. In the first part of the report we have described OpenFlow technology and traditional software forwarding technologies like Ethernet Switching and IP Routing. Following we have defined some experimental tests to check OpenFlow performance and to compare it against the other traditional technologies.

Once we have done the experimental tests we can conclude that OpenFlow switching technology is a serious alternative to software Ethernet Switching or IP Routing because it does the same layer-2 and layer-3 functions with a high performance and scalability. OpenFlow does not just do layer-2 and layer-3 forwarding, but also can do port forwarding and layer-4 forwarding, so we can consider it more flexible and configurable. This added with VLAN capabilities do it a highly recommended switching technology for isolate flows and network communications.

Furthermore OpenFlow is not only a forwarding technology, it is an architecture composed of network elements that manage and reconfigure the system depending on flow specifications and network status. This task is done by the Controller in which can be installed other applications running beside it to perform advanced functions.

If we observe the obtained results of the experimental tests, we can see that OpenFlow has similar performance results as IP Routing in terms of Throughput and processing with large forwarding tables. Switching technology has obtained no good results with small packets and large forwarding tables compared with the other two technologies. Also software Switching has bad forwarding performance when the system is overloaded due to high-rate flows. Hence we can affirm that OpenFlow code has been well-implemented and optimized for forwarding tasks.

Other important reasons why OpenFlow must be taken into account are that it can be installed on a PC or a commercial router as an added function in TCAM and that is a technology backed by major networking manufacturers such as CISCO,

Juniper, HP and NEC.

Finally we can conclude that OpenFlow although can be still considered novel or immature, it is an interesting technology to network managers that want to isolate flows separating production and experimental traffic, or to someone who wants to design a configurable network.

5.1 Future Lines

This section wants to propose several ideas to a future continuing of this project. Some of the ideas are strongly related with this project and other could be a new line of research.

- We have made OpenFlow performance tests without a Controller. It could be interesting to test the performance of the environment when the OpenFlow switch is managed by a Controller (NOX) that has to process some flows and to add new flow definitions.
- The OpenFlowSwitching community is working to add Quality Of Service (QoS) criterias to the technology. It could be interesting to create an environment in which there are different flows with different QoS priorities.
- We have tried to obtain the packet processing time in test 6 but have not obtained correct results due to caching. It could be interesting to search a way to avoid caching to see the difference in latency when packets are processed by the linear or the hash table.
- Undertand if it could be interesting to manage controllers for cooperation tasks (parallel working, process load balancing, controller restoring after a failure...).
- Try to implement an application to run over NOX controller to process some traffic with a predefined pattern.
- Try to create a test bed with a real environment with some OpenFlow switches and Controllers (interconnection with other campus).

Bibliography

- [1] GENI: Global Environment for Network Innovations. Web site: <http://geni.net>.
- [2] CISCO: Internetworking Technology Handbook . Web site: http://www.cisco.com/en/US/docs/internetworking/technology/handbook/ito_doc.html.
- [3] ANSI/IEEE 802.1d standard. Web site: <http://standards.ieee.org/getieee802/>.
- [4] OpenFlow: Enabling Innovation in Campus Networks. Web site: <http://www.openflowswitch.org//documents/openflow-wp-latest.pdf>
- [5] OpenFlow Switch Specification v0.8.9. Brandon Heller (brandonh@stanford.edu). Web site: <http://www.openflowswitch.org/documents/openflow-spec-v0.8.9.pdf/>
- [6] NOX: Towards an Operating System for Networks, Natasha Gude Teemu Koponen Justin Pettit Ben Pfaff Martín Casado Nick McKeown Scott Shenker. Web site: <http://noxrepo.org/doc/nox-ccr-final.pdf>
- [7] Ethane: Taking control of the enterprise, Martin Casado Michael J.Freedman Justin Pettit Jianying Luo Nick McKeown Scott Shenker. Web site: <http://www.stanford.edu/jpettit/papers/ethne-sigcomm07.pdf>
- [8] Welsh, Matt y Lar Kauffman. *Running Linux*, O' Reilly & Associates, Inc. Sebastopol, CA, 1995.

Appendix A

How to install OpenFlow and NOX Controller

A.1 OpenFlow

If you want to install OpenFlow switching technology in your Ubuntu PC you have to go to Official Web Page:

http://www.openflowswitch.org/wk/index.php/Ubuntu_Install

A.2 NOX Controller

If you want to install NOX Controller in your Linux PC you have to go to Official Web Page:

<http://noxrepo.org/manual/installation.html>

Appendix B

Technologies configuration

B.1 Layer 2: Bridging configuration

These are the LINUX commands to set up switching utility bridge-utils:

```
brctl addbr bridge_name
brctl addif bridge_name eth0
brctl addif bridge_name eth1
ifconfig bridge_name up
brctl show
```

bridge name	bridge id	STP enabled	interfaces
bridge_name	8000.00004c9f0bd2	no	eth0 eth1

B.2 Layer 3: Routing configuration

These are the LINUX commands to set up the IP routing:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
route add -net networkIP netmask networkMASK dev eth0
route add -net networkIP netmask networkMASK dev eth1
route add default gw gatewayIP
```

B.3 OpenFlow configuration

This are the LINUX commands to set up OpenFlow to act as a switch:

```
insmod datapath/linux-2.6/openflow_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth0
dpctl addif nl:0 eth1
dpctl show nl:0
```

Appendix C

Scripts

C.1 Switching

Here we have the switching shell script for enabling bridging in a linux machine with Bridge-utils:

```
echo 0 > /sys/devices/system/cpu/cpu1/online
brctl addbr switch
brctl addif switch eth1
brctl addif switch eth2
ifconfig switch up
brctl setageing switch 3600
brctl show
brctl showmacs switch
```

C.2 Routing

Here we have the routing shell script for enabling ip_forwarding and for creating routing tables in a linux machine (This is an example to create up to 131072 routing-table entries):

```
echo 0 > /sys/devices/system/cpu/cpu1/online
echo 0 > /proc/sys/net/ipv4/ip_forward
ifconfig eth1 192.1.0.2/24
ifconfig eth2 192.2.0.2/24
echo 1 > /proc/sys/net/ipv4/ip_forward
```

```
for ((k=193;k<195;k++))
do
for ((j=0;j<256;j++))
do
for ((i=0;i<256;i++))
do
route del -net $k.$j.$i.0 netmask 255.255.255.0 gw 192.2.0.3 dev eth2
done
done
done
```

C.3 OpenFlow

Here we have some scripts to set up and create entries in the OpenFlow flow-table:

C.3.1 Flow-table: 100 Linear entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow\_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl show nl:0
for ((k=0;k<1;k++))
do
dpctl add-flow nl:0 dl_type=0x0800, nw_dst=192.5.$k.0, \
idle_timeout=0, actions=output:1
done
dpctl dump-tables nl:0
```

C.3.2 Flow-table: 64K Hash entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl show nl:0
```

```
for ((i=0;i<65536;i++))
do
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
  dl_src=00:00:C0:03:00:02, dl_dst=00:00:C0:04:00:02, \
  dl_type=0x0800, nw_src=192.3.0.2, nw_dst=192.4.0.2, \
  nw_proto=17, tp_src=$i, tp_dst=0, idle_timeout=0, \
  actions=output:1
done
dpctl dump-tables nl:0
```

C.3.3 Flow-table: 128K Hash entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl show nl:0
for ((i=1;i<65536;i=$((i+2)))
do
for ((j=1;j<8;j=$((j+2)))
do
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
  dl_src=00:00:C0:03:00:02, dl_dst=00:00:C0:04:00:02, \
  dl_type=0x0800, nw_src=192.3.0.2, nw_dst=192.4.0.2, \
  nw_proto=17, tp_src=$i, tp_dst=$j, idle_timeout=0, \
  actions=output:1
done
done
dpctl dump-tables nl:0
```

C.3.4 Flow-table: 100 Linear and 96K Hash entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow\_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl show nl:0
```

```
for ((k=3;k<103;k++))
do
dpctl add-flow nl:0 dl_type=0x0800, nw_dst=192.4.$k.0, \
idle_timeout=0, actions=output:1
done
for ((i=1;i<65536;i=$i+2))
do
for ((j=0;j<3;j++))
do
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
dl_src=00:00:C0:03:00:02, dl_dst=00:00:C0:04:00:02, \
dl_type=0x0800, nw_src=192.3.0.2, nw_dst=192.4.0.2, \
nw_proto=17, tp_src=$i, tp_dst=$j, idle_timeout=0, \
actions=output:1
done
done
dpctl dump-tables nl:0
```

C.3.5 Flow-table: 100 Linear and 128K Hash entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow\_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl show nl:0
for ((k=3;k<103;k++))
do
dpctl add-flow nl:0 dl_type=0x0800, nw_dst=192.4.$k.0, \
idle_timeout=0, actions=output:1
done
for ((i=1;i<65536;i=$i+2))
do
for ((j=1;j<8;j=$j+2))
do
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
dl_src=00:00:C0:03:00:02, dl_dst=00:00:C0:04:00:02, \
dl_type=0x0800, nw_src=192.3.0.2, nw_dst=192.4.0.2, \
nw_proto=17, tp_src=$i, tp_dst=$j, idle_timeout=0, \
actions=output:1
```

```
done  
done  
dpctl dump-tables nl:0
```

Appendix D

Scripts for fairness test

Here we have shell scripts for enabling bridging, routing and OpenFlow in a linux machine with 3 interfaces:

D.1 Switching

```
echo 0 > /sys/devices/system/cpu/cpu1/online
brctl addbr switch
brctl addif switch eth1
brctl addif switch eth2
brctl addif switch eth3
ifconfig switch up
brctl setageing switch 3600
brctl show
brctl showmacs switch
```

D.2 Routing

D.2.1 Routing-table: 1024 entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
echo 0 > /proc/sys/net/ipv4/ip_forward
ifconfig eth1 192.1.0.2/24
ifconfig eth2 192.2.0.2/24
ifconfig eth3 192.3.0.2/24
echo 1 > /proc/sys/net/ipv4/ip_forward
for ((j=0;j<4;j++))
do
```

```
for ((i=0;i<256;i++))
do
route del -net 193.$j.$i.0 netmask 255.255.255.0 gw 192.3.0.3 dev eth3
done
done
```

D.2.2 Routing-table: 64K entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
echo 0 > /proc/sys/net/ipv4/ip_forward
ifconfig eth1 192.1.0.2/24
ifconfig eth2 192.2.0.2/24
ifconfig eth3 192.3.0.2/24
echo 1 > /proc/sys/net/ipv4/ip_forward
for ((j=0;j<256;j++))
do
for ((i=0;i<256;i++))
do
route del -net 193.$j.$i.0 netmask 255.255.255.0 gw 192.3.0.3 dev eth3
done
done
```

D.3 OpenFlow

D.3.1 Flow-table: 1024 entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl addif nl:0 eth3
dpctl show nl:0
for ((i=0;i<512;i++))
do
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
dl_src=00:00:C0:01:00:02, dl_dst=00:00:C0:03:00:02, \
dl_type=0x0800, nw_src=192.1.0.2, nw_dst=192.3.0.2, \
nw_proto=17, tp_src=$i, tp_dst=0, idle_timeout=0, \
```

```
actions=output:2
done
for ((i=0;i<512;i++))
do
dpctl add-flow nl:0 in_port=1, dl_vlan=0xffff, \
dl_src=00:00:C0:02:00:02, dl_dst=00:00:C0:03:00:02, \
dl_type=0x0800, nw_src=192.2.0.2, nw_dst=192.3.0.2, \
nw_proto=17, tp_src=$i, tp_dst=0, idle_timeout=0, \
actions=output:2
done
dpctl dump-tables nl:0
```

D.3.2 Flow-table: 64K entries

```
echo 0 > /sys/devices/system/cpu/cpu1/online
cd /home/openflow3/openflow
insmod datapath/linux-2.6/openflow_mod.ko
dpctl adddp nl:0
dpctl addif nl:0 eth1
dpctl addif nl:0 eth2
dpctl addif nl:0 eth3
dpctl show nl:0
for ((i=0;i<32768;i++))
do
dpctl add-flow nl:0 in_port=0, dl_vlan=0xffff, \
dl_src=00:00:C0:01:00:02, dl_dst=00:00:C0:03:00:02, \
dl_type=0x0800, nw_src=192.1.0.2, nw_dst=192.3.0.2, \
nw_proto=17, tp_src=$i, tp_dst=0, idle_timeout=0, \
actions=output:2
done
for ((i=32768;i<65536;i++))
do
dpctl add-flow nl:0 in_port=1, dl_vlan=0xffff, \
dl_src=00:00:C0:02:00:02, dl_dst=00:00:C0:03:00:02, \
dl_type=0x0800, nw_src=192.2.0.2, nw_dst=192.3.0.2, \
nw_proto=17, tp_src=$i, tp_dst=0, idle_timeout=0, \
actions=output:2
done
dpctl dump-tables nl:0
```