Thesis no: MSEE-2016-47

Measuring And Modeling Of Open vSwitch Performance

Implementation in Docker

HARSHINI NEKKANTI

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Masters in Electrical Engineering with emphasis on Telecommunication Systems. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Harshini Nekkanti
E-Mail: hane15@student.bth.se

University advisor:
Prof. Dr. Kurt Tutschku
Department of Communication Systems
School of Computing
BTH Karlskrona

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Network virtualization has become an important aspect of the Telecom industry. The need for efficient, scalable and reliable virtualized network functions is paramount to modern networking. Open vSwitch is such virtual switch that attempts to extend the usage of virtual switches to industry grade performance levels on heterogeneous platforms.

The aim of the thesis is to give an insight into the working of Open vSwitch. To evaluate the performance of Open vSwitch in various virtualization scenarios such as KVM (second companion thesis)[1] and Docker. To investigate different scheduling techniques offered by the Open vSwitch software and supported by the Linux kernel such as FIFO, SFQ, CODEL, FQCODEL, HTB and HFSC. To differentiate the performance of Open vSwitch in these scenarios and scheduling capacities and determine the best scenario for optimum performance.

The methodology of the thesis involved a physical model of the system used for real-time experimentation as well as quantitative analysis. Quantitative analysis of obtained results paved the way for unbiased conclusions. Experimental analysis was required to measure metrics such as throughput, latency and jitter in order to grade the performance of Open vSwitch in the particular virtualization scenario.

The results of the thesis must be considered in context with a second companion thesis[1]. Both the thesis aim at measuring the performance of Open v-Switch but the virtualization scenarios (Docker and KVM) which are chosen are different, However, this thesis outline the performance of Open v-Switch and linux bridge in docker scenario. Various scheduling techniques were measured for network performance metrics across both Docker and KVM (second companion thesis) and it was observed that Docker performed better in terms of throughput, latency and jitter. In Docker scenario amongst the scheduling algorithms measured, it has almost same throughput in all scheduling algorithms and latency shows slight variation and FIFO has least latency, as it is a simplest algorithm and consists of default qdisk. Finally jitter also shows variation on all scheduling algorithms.

The conclusion of the thesis is that the virtualization layer on which Open vSwitch operates is one of the main factors in determining the switching performance. The KVM scenario and Docker scenario each have different virtualization techniques that incur different overheads that in turn lead to different measurements. This difference occurs in different packet scheduling techniques. Docker performs better than KVM for both bridges. In the Docker scenario Linux bridge performs better than that of Open vSwitch, throughput is almost constant and FIFO has a least latency amongst all scheduling algorithms and jitter shows more variation in all scheduling algorithms.

Keywords: Docker, KVM, Open vSwitch, Scheduling

ACKNOWLEDGEMENT

I wish to express my deep sense of gratitude to my supervisor prof. Dr. Kurt Tutschku for giving his valuable suggestions in every aspect of our thesis. The suggestions are very helpful to improve the quality of our thesis. His patience, support and clarifying doubts whenever we are in need helped us a lot to give the better results of our thesis. He motivated us to do our thesis in a better way. I am so thankful to my supervisor for his endless encouragement.

I am pleased to thank all the people who helped me directly or indirectly for the completion of my thesis.

Above all I would like to thank my parents who gave me support to complete my thesis.

CONTENTS

ABBREVIATIONS

NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
SDN	Software Define Networking
OVS	Open vSwitch
VIF	Virtual Interface
PIF	Physical Interface
VETH	Virtual Ethernet Interface
FIFO	First In First Out
SFQ	Stochastic Fair Queuing
CODEL	Controlled Delay
FQ-CODEL	Fair Queuing Controlled Delay
HTB	Hierarchical Token Bucket
HFSC	Hierarchical Fair Service Curve
KVM	Kernel Virtual Machine
QDISC	Queuing Discipline
DPDK	Data Plane Development Kit
MAC	Medium Access Control
UUID	Universally Unique Identifier

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Virtualization has become an important part of the Telecom & IT industry. Many datacenters are virtualized to provide quick provisioning, spill over to the cloud, and improve availability during periods of disaster recovery. Along with virtualized operating systems, storage devices and hardware platforms there is an overpowering need for network virtualization. This need was met with the evolution of Network functions virtualization (NFV) and Software-defined networking (SDN) concepts.

NFV is a network architecture concept that uses the technologies of IT virtualization to virtualize entire classes of network node functions into building blocks that may connect, or chain together, to create communication services[3].

SDN is an approach to computer networking that allows network administrators to manage network services through abstraction of lower-level functionality. SDN is meant to address the fact that the static architecture of traditional networks doesn't support the dynamic, scalable computing and storage needs of more modern computing environments such as data centers[4].

Thus, both NFV and SDN have a strong potential to shape the future networks. The rise of server virtualization has brought with it a fundamental shift in data center networking. A new network access layer has emerged in which most network ports are virtual, not physical, which has caused the development of software/virtual switches.

A virtual switch is a software program that essentially facilitates the communication between virtual machines/containers existing on the same or different physical machines. It provides the basic functionality of a physical switch of traffic forwarding but also provides various other features such as GRE tunneling, multicast snooping and visibility into inter-VM traffic monitoring. They operate based on the control plane-data plane architecture of most SDN frameworks.

1.1 Aim, Objectives and Research Questions

The aim of the thesis is to gain an understanding of Open vSwitch design, operation and performance. To model its performance with respect to various network parameters such as throughput, latency and jitter.

To analyze the performance in various scenarios and based on experimental results, determine the best implementation scenario. To test the performance in various scheduling algorithms and analyze performance of each scheduling in these scenarios.

A brief overview of the objectives is as follows:

- Study of Open vSwitch design and operation
- Study of Open vSwitch scheduling techniques
- Identifying performance metrics for experimentation
- Implementation of Open vSwitch in Docker virtualization scenario
- Implementation of Linux Bridge in Docker virtualization scenario
- Analysis of obtained experimental results of OVS in a statistical manner
- Comparative analysis of obtained results between KVM and Docker scenarios in both Open vSwitch and Linux bridges.

Research Questions:

1. How can we measure the network performance of Open vSwitch in Docker environment?
2. How can we model the performance of an Open vSwitch with respect to scheduling technique employed?
3. How can one determine network performance of Open vSwitch in different virtualization scenarios? Does it vary with different virtualization scenarios?

1.2 Motivation

Modern networking has replaced physical hardware such as switches, bridges and routers with VNF (Virtual Network Functions) and Virtualized networks. This rise in virtualized technology has reduced capital and operational expenditure but has increased processing overhead and reduced performance. Numerous virtualization technologies have tried to reduce this tradeoff and have succeeded to some extent. There are multiple virtual switches that have flexible implementation, cost-effective resource sharing and easily portable across multiple platforms[5].

There is a need for using virtual switches in different scenarios as it helps to communicate different virtual machines or containers, which are present in two different hosts or single host. Virtual switches have many advantages like high flexibility, low cost, vendor independence and many other and these virtual switches focuses more on performance. Open vSwitch is one of the most efficient virtual switches in the current scenario. Many researchers have attempted to analyze the performance and study its internal operation. Contributors have also increased its performance using hardware such as DPDK (Data plane Development Kit). Developers have added multiple patches and plugins to maximize compatibility of OVS in most platforms[6][7].

Recent developments have included various packet-handling techniques such SFQ (Stochastic Fair Queuing), CODEL (Controller Delay) and FQCODEL (Fair queuing Controller Delay). These algorithms are provided by the Linux kernel and its implementation by the Open vSwitch is fairly new.

The thesis focuses on packet handling techniques of Open vSwitch in various virtualization platforms (KVM and Docker). The network performance has been compared to the default linux bridge performance. The latest scheduling techniques supported by Open vSwitch have been implemented and its performance has been analyzed.

1.3 Thesis Outline

Chapter 1 includes the aims and objectives, research questions and motivation behind the thesis. Chapter 2 provides the background knowledge in open v-switch technology. It explains the architecture of Open vSwitch and also gives a brief description of working of each scheduling technique supported by Open vSwitch and describes the implementation of Open vSwitch in these scheduling algorithms. Chapter 3 provides related work, which includes the contributions by other authors in the field of Open vSwitch networking on various virtualization scenarios. Chapter 4 provides

the Methodology of the approach used in this thesis. It gives a brief description of performance analysis techniques used and also explains about the metrics involved in the measurements. Chapter 5 explains the experimental set up, design and obtained results. Chapter 6 gives the analysis of the obtained results and also compares the results obtained with second companion thesis. It describes about the packet handling techniques in different virtualization scenarios. Chapter 7 gives a conclusion of the thesis. It briefly summarizes the results and also provides the future work of this thesis.

1.4 Split of Work

This section outlines the division of work between the thesis partners.

SECTION	TOPIC	CONTRIBUTOR
Chapter 1	1.1 Open v-Switch and its application	Rohit Pothuraju
	1.2 Scheduling algorithms used in this thesis	Harshini Nekkanti
	1.3 Aims, Objectives and Research questions	Harshini Nekkanti
	1.4 Motivation	Harshini Nekkanti Rohit Pothuraju
	1.5 Thesis Outline	Harshini Nekkanti
Chapter 2	2.1 Open vSwitch Technology	Rohit Pothuraju
	2.2 Scheduling Algorithms	Harshini Nekkanti
Chapter 3	Related Work	Harshini Nekkanti
Chapter 4	Methodology	Harshini Nekkanti Rohit Pothuraju
Chapter 5	5.1 Experimental Setup	Harshini Nekkanti
	5.2 Common Results	Harshini Nekkanti Rohit Pothuraju
	5.3 Individual Results	Harshini Nekkanti
Chapter 6	Analysis and Discussion	Harshini Nekkanti Rohit Pothuraju
Chapter 7	7.1 Conclusion	Harshini Nekkanti Rohit Pothuraju
	7.2 Future Work	Harshini Nekkanti

2 BACKGROUND

2.1 Open V-Switch and its Application

Open vSwitch (OVS) is one of the most efficient virtual switches. It is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation, while still supporting standard management interfaces and protocols. It can operate in many virtualization platforms such as Xen, KVM, Docker and VirtualBox. It has also been integrated into virtual management systems including OpenStack, openQRM and oVirt. The entire code is written in C and is portable[6].

Open vSwitch is targeted at multi-server virtualization deployments. These environments are often characterized by highly dynamic end-points, the maintenance of logical abstractions, and integration with or offloading to special purpose switching hardware[6].

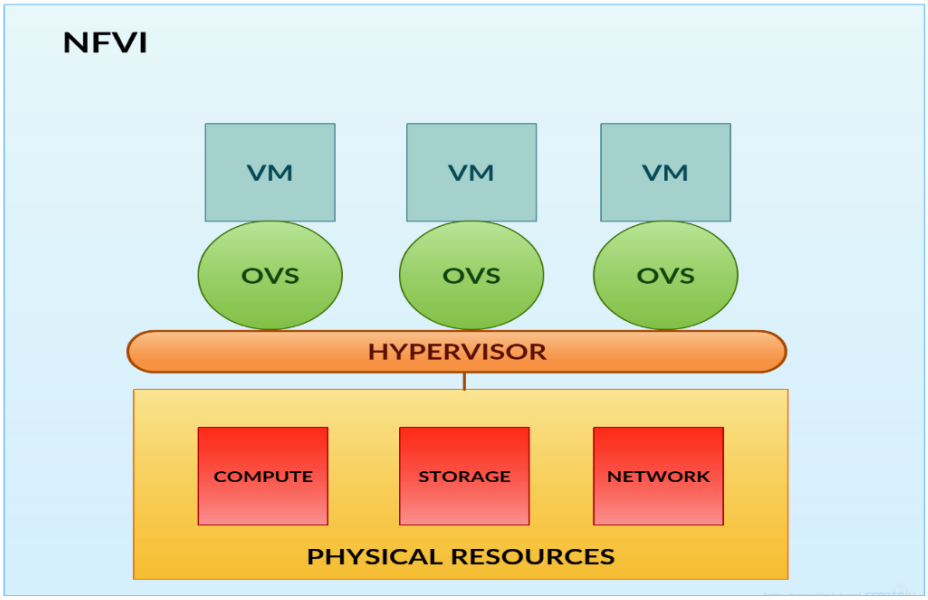

Figure 1 Open vSwitch placement in NFVI infrastructure

The focus lies on the NFVI infrastructure that depicts the relation between the hardware resource emulation into virtual resources. OVS lies on top of the virtualization layer (hypervisor) and interfaces the virtual machines.

Each VM has a virtual interface (VIF) that enables communication with other VMs in the same virtual network and forwards traffic from the VM to remote destinations. Open vSwitch enables bridging of VMs in the same virtual network and communication with other virtual or physical networks via the physical interfaces (PIF) as shown in Figure1[8][9].

It acts as a central component in the communication domain of virtual machines by interconnecting physical interfaces with the virtual interfaces.

2.1.1 Reasons for choosing Open vSwitch as a Software Switch

Hypervisors require the ability to bridge traffic between VMs and with remote destinations. On Linux-based hypervisors, this used to mean using the fast and reliable Linux Bridge (built-in L2 switch). Open vSwitch has proved to be an effective alternative to the basic Linux Bridge in terms of network performance[10].

As given by open vswitch git hub

Open vSwitch when compared to other virtual switches

- OVS enables easy configuration and migration of both slow and fast network states between instances. OVS is based on a real data-model, which leads to better automation systems.
- OVS can respond and adapt to the environmental changes through Netflow and sFlow features.
- OVS provides efficient orchestration of network processes by logically appending or manipulating tags in network packets.
- The present OVS switches are well integrated with the existing hardware, providing a single automated control mechanism for both bare-metal and virtualized hosting environments.

2.1.2 Working of Open vSwitch

During a switching operation, the first packet of a new flow that arrives at the ingress port is a miss as the packet and its flow data (source MAC and destination MAC addresses, source and destination ports as well as the action required) does not exist in the kernel IP table

The daemon carries requests from the user and implements the request on the kernel IP-forwarding table. The daemon functionality is same in most OS environments. Whereas, the kernel datapath is written specifically based on the kernel the OS supports in order to achieve highest possible performance[11].

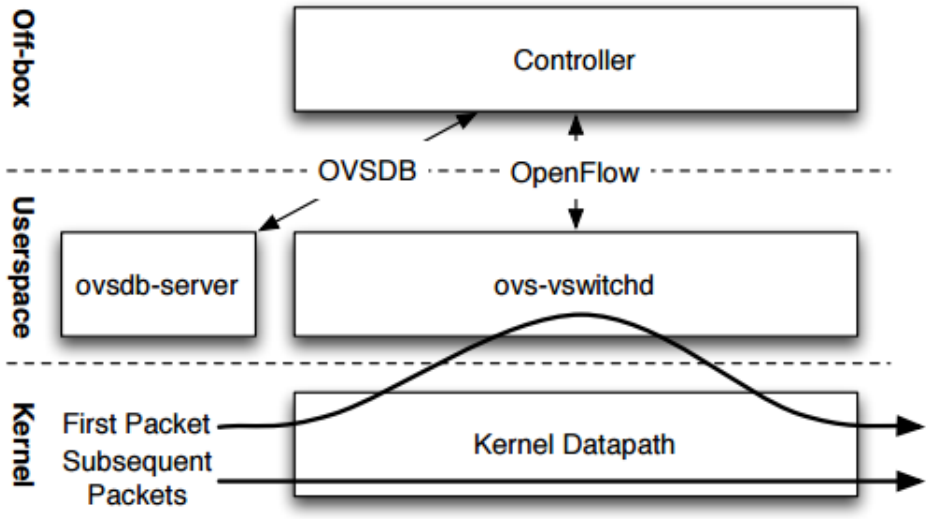

Figure 2 Open vSwitch working

The datapath passes the packet to the daemon that immediately queries the controller for actions to be performed on the packet; it then queries the ovsdb-server for forwarding decision for packets of this flow. This operation comes under slow path when userspace controls packet forwarding. The decision is then cached in the kernel so that subsequent packets follow cached action. This operation comes under fast path as only kernel is involved[11].

This procedure ensures minimum interference and overhead by the userspace by shifting majority of the working on kernel datapath and at the same time allows the user to decide the action on the arriving packets.

As the kernel datapath is performing majority of the forwarding operation, network performance depends mostly on the OS kernel and its packet handling capabilities. This is the main objective of virtual switches, to reduce its resource utilization thereby maximizing resources available to the hypervisor in order to run user workloads to the best of its abilities[11].

2.2 Architecture of Open V-Switch

OVS comprises of three main components:

- Userspace daemon (ovs-vswitchd)
- Database server (ovsdb-server)
- Kernel datapath (openvswitch.ko)

The ovs-vswitchd and the ovsdb-server come under the userspace, and the kernel datapath module comes under the kernel space. The controller (or control cluster) lies in the off-box region and functions only when requested by the userspace.

Ovs-vswitchd is the daemon program that can control any number of vSwitches on a particular local machine. It can connect to the ovsdb-server via TCP port, SSL port or a unix domain socket. The default is unix:/usr/local/var/run/openvswitch/db.sock [11].

At startup, the daemon will retrieve default configuration from the database and setup datapaths to perform switching across each OVS bridge based on the configuration data retrieved. Only a single ovs-vswitchd instance is needed to run at a time, managing as many vswitch instances as the kernel datapath can support[12].

The ovsdb-server program provides Remote Procedural Calls (RPC) interfaces to the OVS databases (OVSDBs), it supports JSON-RPC client connections over active or passive TCP/IP or Unix domain sockets[13].

The kernel module performs packet switching based on entries in the kernel IP table. Each flow is assigned an action whether to forward, drop or encapsulate the incoming packet[14].

Figure 3 gives a brief description of vSwitch operation. The daemon and the kernel module communicate by Netlink, which is a Linux kernel interface used for Inter process communication (IPC) between kernel space and user space processes. The daemon also has to communicate with the ovsdb-server using the ovsdb management protocol. The management protocol is the means by which the database responds to the queries. The daemon communicates with the controller by the Openflow protocol. Openflow is a protocol used to manage the forwarding plane of a vSwitch. Flows can be controlled using ovs-ofctl commands[11].

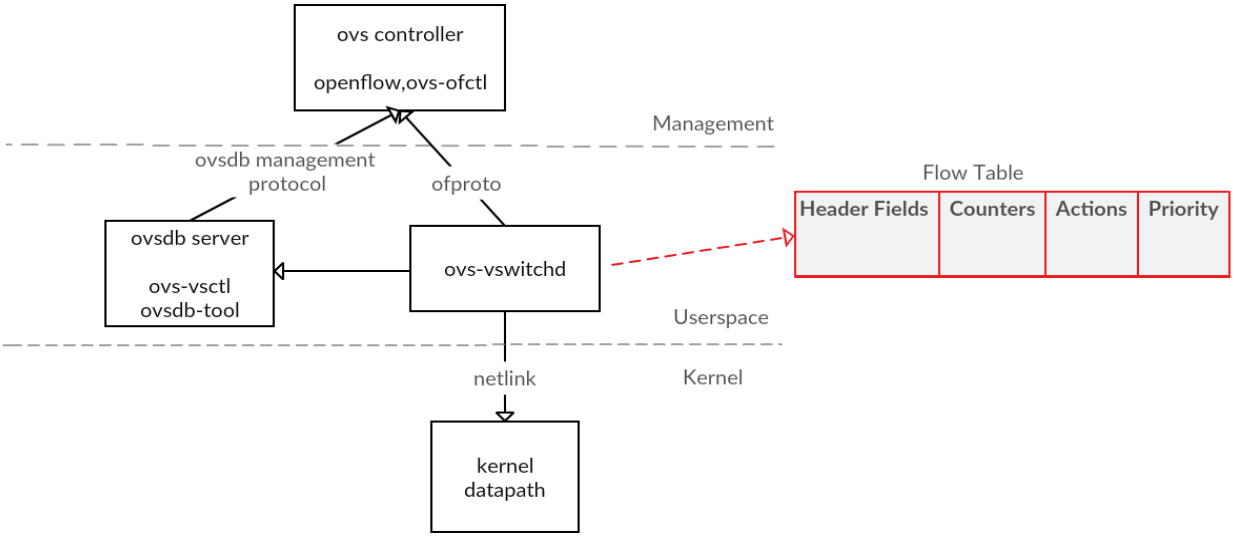

Figure 3 Architecture of Open vSwitch

2.3 Scheduling Algorithms

2.3.1 What is scheduling?

Packet scheduling is the process of managing incoming packets at the ingress port (enqueuing) and outgoing packets at the egress port (dequeuing). A packet scheduler implements QoS policies to ensure that packets of a flow are serviced in order to maximize application performance and minimize delay and packet loss[15][16].

A scheduling algorithm is a set of instructions employed at the port in order to specifically drop or reorder packets in these transmit or receive buffers. Queuing disciplines are used as attempts to ensure steady network performance and compensate for network conditions such as congestion[17].

Types of Queuing disciplines:

- Classless Scheduling: These algorithms do not impose class system on the incoming packets at the ingress buffer nor does it impose class system while dequeuing at the egress buffer. It either reschedules, drops or delays the packets. These queuing disciplines can be used to shape the traffic of an entire interface without any subdivisions. These are the fundamental scheduling techniques used in the Linux kernel[18].

- Classful Scheduling: These scheduling algorithms impose class system on the buffers and can have filters attached to them. The packets can be directed to particular classes based on TOS (Type on Service) and sub queues. The root qdisc or root queuing discipline is the primary qdisc and multiple leaf classes can come under the root qdisk[19].

Thesis focuses on testing the packet scheduling abilities of OVS in Linux environment, thereby follows some of the scheduling techniques offered by the Linux kernel. We have tested all scheduling algorithms supported by Open vSwitch, other algorithms offered by Linux kernel are not supported by Open vSwitch.

CLASSFUL	CLASSLESS
HTB (Hierarchical Token Bucket)	**FIFO (First In First Out)**
HFSC (Hierarchical Fair Service Curve)	**SFQ (Stochastic Fair Queuing)**
	CODEL (Controlled Delay)
	FQCODEL (Fair Queuing Controlled Delay)

Table 1 Scheduling Algorithms implemented in thesis

Of the above mentioned scheduling algorithms in table 1, HTB and HFSC are classful scheduling algorithms and can impose priority on the packets based on certain criteria such as TOS (Type of service) and the other four algorithms FIFO, SFQ, CODEL and FQCODEL are classless scheduling algorithms whose sole function is to control the flow of data both at ingress and egress irrespective of priorities. The algorithms will be explained in depth in the following sections

2.3.2 Scheduling Techniques supported by Linux

First In First Out (FIFO): This algorithm is a classless scheduling algorithm where the first packet or process, which enters the queue first is served first and the next packet which is after the first one is transmitted second. Here all the packets, which arrive, are maintained in a single queue for each outgoing network interface card (NIC) and if the queue is completely full then the next incoming packets entering into the queue are dropped until the buffer space is available.

In the Linux kernel, FIFO is implemented as pfifo_fast queuing discipline and is the default queuing discipline on all interfaces. It provides 3 different bands (i.e. 0,1 and 2) for separating traffic. The 0 band is given highest priority, then 1 and 2 bands. The interactive flows are given to the 0 band so that these packets are dequeued before the other bands[18].

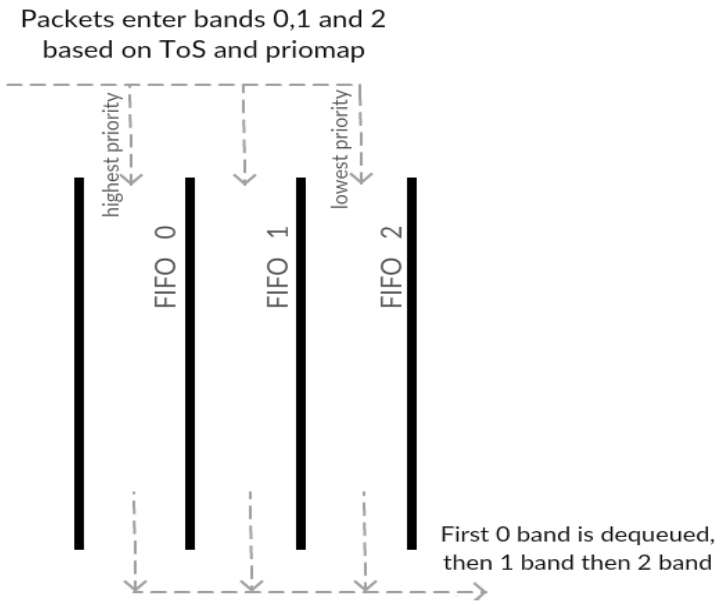

Figure 4 pfifo_fast operation[18]

Stochastic Fair Queuing (SFQ): This algorithm implements fair queuing logic in its scheduling operation. This discipline attempts to fairly distribute the available bandwidth amongst the flows. A hash function is used for traffic control. The different incoming packets are arranged into different FIFO buffers that are dequeued in a round-robin manner. The SFQ algorithm operates in a conversation manner. The algorithm prevents any one conversation from dominating the available bandwidth

This hash function is altered periodically according to the perturbation time in order to avoid unfairness in the division of traffic by the hash function.

The parameters used in configuring SFQ qdisc are:

Perturb: This is the time interval, which stops and alternates the hash function in order to ensure fairness in the division of hash buckets[18]

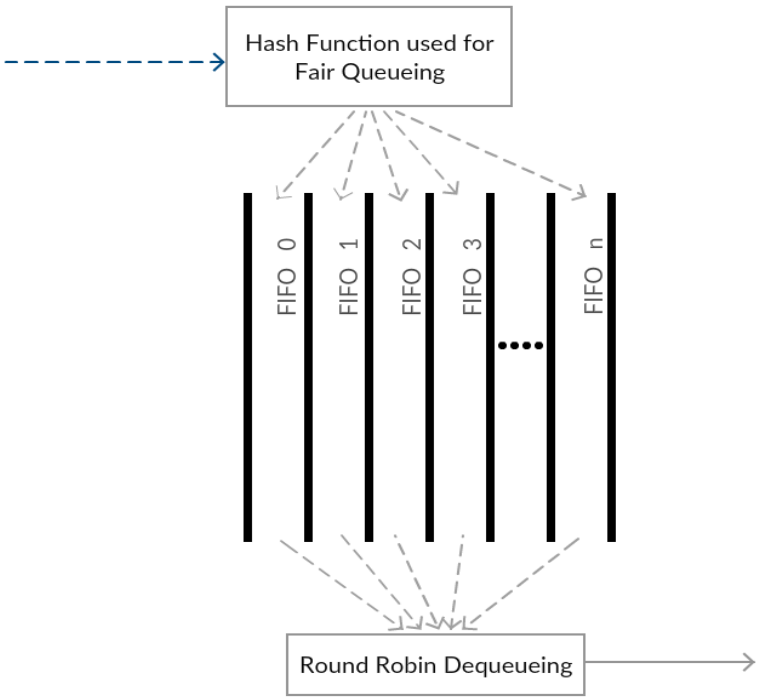

Figure 5 Stochastic Fair Queuing Operations[18]

Controlled Delay (CODEL): This algorithm is an Active queue management algorithm. It measures the minimum local queue delay and compares it to the target queue delay. As long as the minimum queue delay is less than the target queue delay, packets are not dropped. It uses minimum delay as a state-tracking variable to determine whether packets are to be dropped or forwarded.

The qdisc enters a dropping mode when the minimum queue delay has exceeded the target queue delay for a period of time that exceeds a preset threshold. Packets are dropped at different times, which is set by a control law. The control law ensures that the packet drops cause a linear change in the throughput. Once the minimum delay is below target queue delay, packets are not dropped and qdisc resumes forwarding mode[20].

The parameters used in configuring CODEL qdisc are:

Limit: This is the threshold of buffer size. When exceeded, packets are dropped

Target: This is the acceptable queue delay; observing local minimum queue delay identifies it

Interval: This is the period of time within which the queue delay can exceed target queue delay before packets are to be dropped in order to return to this target queue delay

The control law implemented is the time for the next packet drop is inversely proportional to the square root of the number of drops since the dropping mode was entered[20].

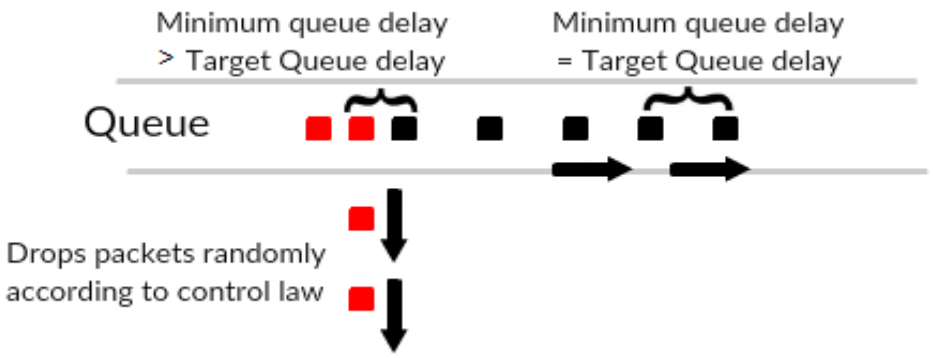

Figure 6 Controlled Delay Operation

Fair Queuing with Controlled Delay (FQ-CODEL): This algorithm combines the fair queuing logic from SFQ and controlled delay AQM logic from CODEL. It classifies the incoming packets into different flows and distributes the available bandwidth to all the flows. Internally each flow is managed by CODEL queuing discipline. Packet reordering is avoided since CODEL uses FIFO queue[21].

The parameters used in configuring FQCODEL qdisc are:

***Flows*:** It is the number of flows into which the incoming packets are classified. Due to the stochastic nature of hashing, multiple flows may end up being hashed into the same slot.

Quantum: It is the amount of bytes used as deficit in the Fair queuing algorithm[21]

Limit, target and interval are the same as CODEL qdisc

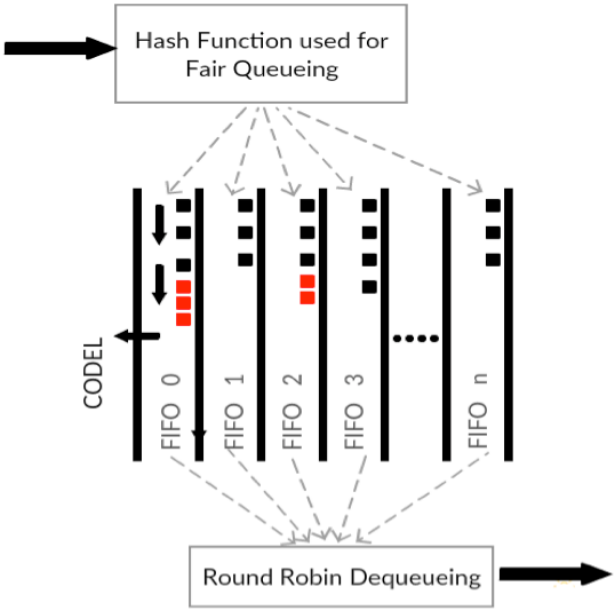

Figure 7 Fair Queuing Controlled Delay Operation

Hierarchical token bucket (HTB): This algorithm is a classful queuing discipline. It can impose priority on different flows and service packets based on hierarchy. It uses token buckets and filters to allow for complex and granular control over traffic. HTB divides classes into root classes and leaf classes.

HTB comprises of a Shaping mechanism and a Borrowing mechanism. The Shaping mechanism occurs in leaf classes. No shaping occurs in inner and root classes as they are used for borrowing mechanism.

Leaf classes borrow tokens from inner root classes if 'rate' is exceeded. It will continue to borrow till 'ceil' is reached after which packet is buffered until more tokens are available[19][22].

The parameters used in configuring HTB qdisc are:

Rate: This is the minimum guaranteed rate to which traffic is transmitted

Ceil: This is the maximum rate allowed by the root qdisc set on the interface

Burst: This is the size of the token bucket in bytes.

Cburst: This is the size of the ceil token bucket in bytes.

These are the various class states:

'Inner root' refers to the bandwidth required by inner class (child class of root class) 'Leaf' refers to bandwidth required by a leaf class (child class of an inner root class)

- **Leaf < rate:** Leaf class will be dequeued depending on tokens and burst rate.
- **Leaf > rate < ceil:** Leaf class will borrow from parent class. If tokens are available, they will be provided in quantum increments and leaf class will dequeue up to cburst bytes.
- **Leaf > ceil:** No packets are dequeued. There is an increase in latency to meet the desired rate.
- **Inner root < rate:** Tokens are lent to child classes.
- **Inner root > rate < ceil:** Tokens are borrowed from parent class, borrowed and then lent to competing child classes in quantum increments per request
- **Inner root > ceil:** Tokens are not borrowed from parent neither lent to child classes

Hierarchical Fair Service Curve (HFSC): This algorithm is also a classful queuing discipline. It operates according to service curve definitions.

It guarantees precise bandwidth and delay allocation for all leaf classes distributes excess bandwidth fairly amongst child classes and minimizes the discrepancies between service curve and the actual amount of service provided during link sharing[23].

Realtime criterion (RT): This criterion ignores the class hierarchy and guarantees precise bandwidth and delay allocation. The packet that is most eligible for sending is the one with the shortest deadline time.

Linkshare criterion (LS): This criterion distributes bandwidth according the class hierarchy

Upperlimit criterion (UT): This criterion enables packet transmission only if current real time is later than fit-time[23][24].

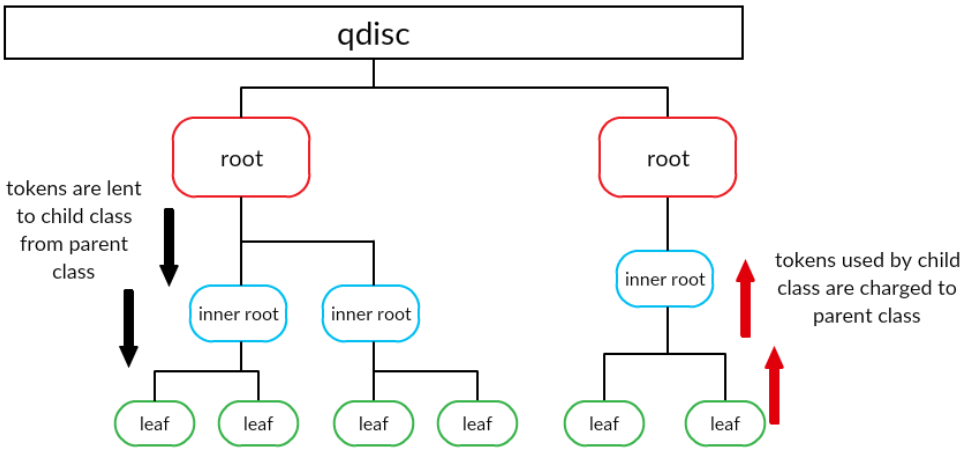

Figure 8 Class Systems for HTB and HFSC Scheduling Algorithms[19]

2.3.3 Scheduling with Open V-Switch

Open vSwitch has a QoS table that implements queuing disciplines on specific ports on the bridge. A queuing discipline will be implemented on a particular port if it is referenced in this table. The user has to manually set the port to a particular queuing discipline, and that Port will point to particular QoS entry in the QoS table/.

The QoS table will contain all QoS entries referenced by all Ports, each QoS entry can point to a queue with configured maximum and minimum rates to control flow of data within the configured port[25][26].

Only the Classful queuing disciplines (HTB and HFSC) can assign maximum and minimum rates in the QoS table, meaning the root class is given a certain bandwidth, so that inner root and leaf classes can borrow from the parent class if needed.

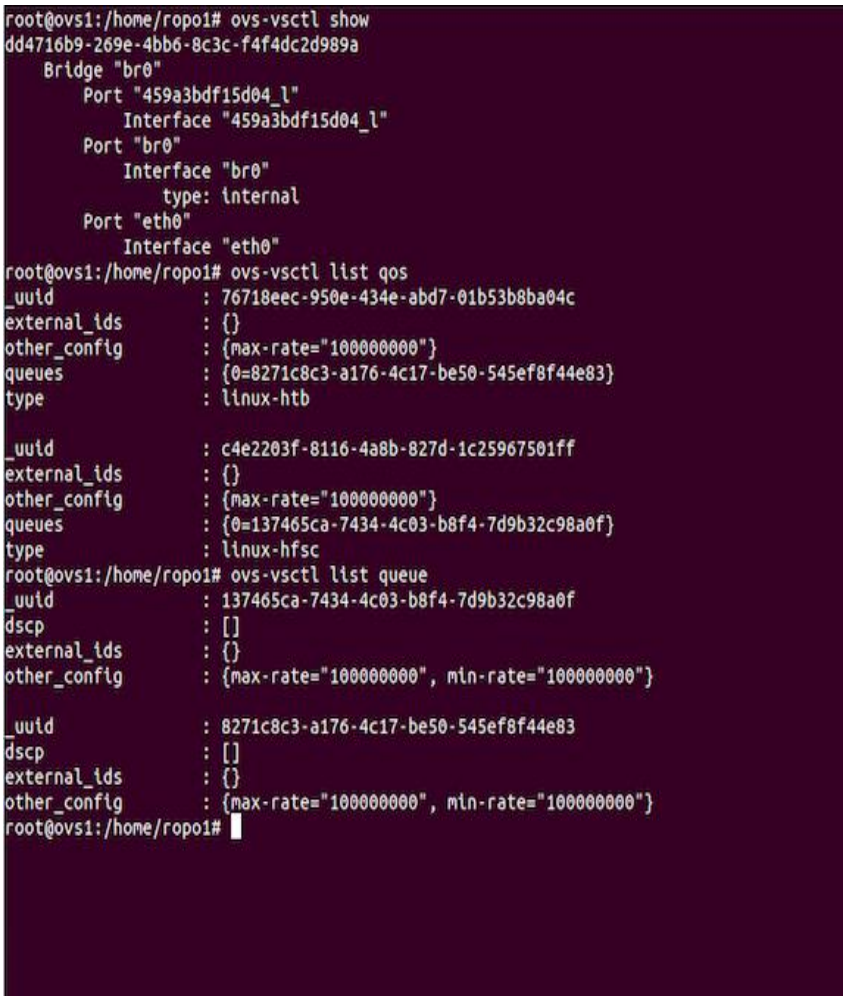

Figure 9 Implementing Scheduling Open vSwitch

In Figure9, HTB scheduling was implemented in Docker Container. The Port table displays the Veth interface linked to the container. The QoS entry in the Port table points to the QoS table entry of the same uuid.

The QoS table displays that entry with other details such as queues and type of scheduling. The queue entry of the QoS table displays the default q0 queue that points to a queue entered in the Queue table.

3 RELATED WORK

In this paper[27], the authors modeled a new performance metric for throughput calculation for NFV elements. The demonstration involved measurement of packet transmission in multiple timescales. This experimentation was performed in Xen and Virtualbox hypervisors. This metric helps to identify performance constraints. It was observed that when the timescale was decreased, the coefficient of throughput variation increased significantly. The timescale can be vital factor in determining the consistency of a network connection.

In this paper[7], the performance of Open vSwitch data plane was analyzed with respect to various scenarios. Multiple forwarding techniques were measured for throughput, flow table caching and CPU load. The device under test was measured for throughput under varying loads with both physical and virtual interfaces. The paper focused on measuring performance with varying packet sizes and flows. This outlined the overhead and the load handling capacity of Open vSwitch. The conclusion of the research was that including virtual switch software in virtual machines was problematic because of significant increase in overhead per packet during the VM/host barrier.

In this paper[11], the entire design and implementation of Open vSwitch was described with high intricacy. It mentioned the flow caching improvements introduced in the later versions. The primary goal of virtual switch software is to reduce resource consumption in order to maximize resources available to the hypervisor to run user workloads. It describes the tuple space classifier used to separate flow entries in the userspace flow table. Microflow caching and Megaflow caching are explained in detail. The design constraints are mentioned in detail.

In this paper[28], performance of different software switches was observed for resource allocation in varying topologies. The research investigates the various factors involved in determining the performance of a software switch such as overhead, CPU frequency and number of rules. The research involved experimentation to achieve certain measurements to compare the performance of Open vSwitch and OFswitch. Both the switches have different lookup techniques for flow match; OF switch has linear lookup whereas Open vSwitch has 2-layer lookup with kernel ad userspace flow tables. The paper concludes that topology design is a major factor in software switch performance and must be considered in cloud network design.

In this paper[29], performance of CPU and network running benchmarks are compared in two different models of micro service architecture for providing the benchmark analysis guidance to system designers. Two models of containers namely master slave and nested container are used to implement Micro service architecture. Network performance is measured under different network virtualization topologies such as Host Network, OpenvSwitch and Linux Bridge across different virtualization environments like bare metals, containers and virtual machines. The Netperf tests like TCP Stream (TCP_STREAM) and TCP Request/Response (TCP_RR) are used to calculate the throughput and latency for each virtualization topology. Here in TCP request/response test buffer size is set to 256K and this test is configured to run in a burst mode to maintain more than one transaction at a same time. In addition, the network throughput and latency of nested containers under different network topologies is calculated and also the throughput and latency on nested containers connected to remote host are calculated.

In this paper[30], real time cloud is built using docker for containers. Here container cloud is divided into two aspects computation and networking. In Computation docker with real time kernel radio access network (RAN) is considered where real time kernels support them. Basically there are two types of real time kernels namely hard real time kernel and soft real time kernel and theses two kernels are

selected in RAN environment as preempt_rt kernel and low latency kernel on hard and soft real time kernels respectively. Maximum latency is measured by considering First In First Out scheduling policy. Cyclist benchmark is used to evaluate latency on two different scenarios like without system load and another with system load. The test takes 500,000 cycles and the latency is compared between Docker, VM and physical on different types of real time kernels. In addition DPDK is considered with Open vSwitch and network performance (throughput and latency) is measured in various containerized environments over different network architectures. The paper finally concludes that the two approaches Docker with DPDK and tuned real time kernels processing latency demonstrates the effectiveness of approaches to reduce latency maintaining the latency optimization.

In this paper[31], virtual environment based on docker is built by using open vswitch on operation system layer. Status of virtual environment is observed using charts and see that whether the containers can provide the security to a system. Experiment is taken as three hosts manager host0, container host1 and host2. Here manager host0 establishes a manage portal to control dockers on host1 and host2 and OVS present on host1 and 2 is used to monitor the network traffic and calculate network performance. The graphs are plotted application performance and network latency is taken into consideration to determine the security. Here both of these metrics are compared with three scenarios Virtual Machines, Native (Linux OS) and Docker. Application performance is measured by using network attack app where cyber attacks programs are considered which are used to send large number of requests to attack targets but the number of requests it sends will depend on different hosts considered. Network latency is measured by using a tool called netperf. Packets are sent from one container to other container, which are present on different hosts. The paper concludes that although the docker performs well in terms of performance, it is in securable in terms of security control but we can strengthen the base of container by some useful tools.

In this paper[32], open vswitch performance is validated with mirroring feature and OVS is considered with DPDK (OVDK) for efficient performance. Here the OVDK performance test was conducted by activating a flow and port mirroring. Network throughput and latency are calculated in OVDK to check the performance. Here in the set up traffic generator is considered to generate the packets and throughput is calculated by varying packet sizes on different conditions i.e. with flow and port mirroring activated and without flow and port mirroring activated. Latency is measured by considering the throughputs, which are obtained by flow mirroring, but to get baseline information latency is calculated without flow mirroring for each throughput value. The paper finally concludes that when the throughput increases latency also increases and latency also increases when port mirroring and flow are activated. As a result this paper conclude that the usage of OVDK results in inconsistency of latency and proved that latency in OVDK is higher than that of modem hardware switches. In addition it results that OVDK when used with port mirroring /flow is feasible in terms of throughput and latency.

4 METHODOLOGY

This section outlines the research methodology of the thesis. The techniques implemented to perform experimental research and reach the goals of the thesis. To achieve the necessary goals, the research technique has to be in par with the desired outcomes. This thesis requires a performance analysis of Open vSwitch scheduling in various scenarios[33].

The different techniques available in modern research are Qualitative and Quantitative analysis:

- **Quantitative Analysis:** It is a research technique that mainly comprises of objective measurements and statistical analysis of data. The data is collected from numerous sources in order to generalize a particular phenomenon or concept. It most often employs mathematical and statistical models[34].

- **Qualitative Analysis:** It is a research technique that mainly comprises of subjective judgment based on unquantifiable data such as industry cycles, strength of research and development and labor relations. Scientific study that can only be observed but not measured[34].

The analysis technique followed was Quantitative analysis. The results and analysis are solely based on numerical findings from experimental tests. The metrics calculated such as Throughput, Latency and Jitter are the basis for our conclusions in determining the best implementation scenario.

This research utilizes Quantitative analysis to reach unbiased results of the performance of Open vSwitch

The research involves modeling of the Open vSwitch performance as opposed to conducting experimental tests on the actual system. An actual system does not exist but a model is designed on which experiments are conducted.

The various models that are available for analysis are: Physical Model and Mathematical Model[35].

- **Physical Model:** It is a realistic physical replica of the actual system. All experimental tests are conducted on the model and the results are assumed to infer that similar observations would occur had the tests been conducted on the actual system. The physical system carries all properties and characteristics of the actual system.

- **Mathematical Model:** It is a mathematical or statistical representation of the actual system. This approach is used in most engineering disciplines. This model may help in explaining the behavior of a system and to analyze the effects of different components.

This research involves a physical model of the system. Experimental operations were conducted on a physical system that had Open vSwitch working on its kernel and were connected to the Internet via a physical switch. No simulation was involved and all tests were conducted in real-time[35].

4.1 Performance Analysis Techniques

The research involves the performance analysis of Open vSwitch in various scenarios. In order to effectively reach this goal, Benchmarking approach was implemented. A benchmark is a standard of reference for accurate measurements. Benchmarking tools are vital in a research.

Benchmarking: It is the process of running a set of programs in order to assess the relative performance of a system. It involves measuring and evaluating computational performance, networking protocols, devices and networks relative to a reference evaluation[36].

The benchmark tools used in this research are NetPerf and Iperf.

NetPerf: This tool is used to measure network performance under different networking scenarios. Its primary focus is on bulk data performance using either TCP or UDP connections. There are multiple tests that can be conducted that provide multiple options to arrive at the necessary operation[37].

NetPerf basically operates in the Client-Server model i.e. the netperf (client) and the netserver (server). The netserver is a background process that runs a specific port (default port 12865). The netperf client is CLI program that establishes a control connection between the client host and the server host. The client port used can change and chooses an available automatically[37].

The connection will be used to pass test configuration data and results to and from the remote system. The control connection will be a TCP connection using BSD sockets. Once the control connection is set up, measurement data is passed using the APIs and protocols depending on the test being conducted.

NetPerf does not interfere with the control connection with traffic during a test. Certain TCP options may put the packets out of the control connection and may result in inaccurate test results[37].

IPerf: This tool is another network testing tool that uses TCP and UDP data streams to measure the performance of a network. It conducts active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters relates to timing, buffers and protocols. Multiple tests exist so that the user can conduct measurements of bandwidth, loss and jitter[38].

This tool also uses the Client-Server model to conduct measurement tests over a network link. The iperf server runs as an active CLI program and must be stopped manually in order to cease the connection. The iperf client is a program that establishes a connection between the local client and the remote server. Data is transmitted from the client to the server; measurements are made based on the type of data sent from client. When the client sends UDP data, jitter and packet loss are calculated at the server[39].

Netperf is an efficient tool and provides numerous test modes. The netserver is daemon process and does not need to be run manually unlike the iperf server. This is the reason netperf was used as the main tool for experimentation. Iperf was used for jitter measurements. Netperf doesn't have test for jitter calculation.

Docker: It is an open source software containerization platform that automates the deployments of Linux applications. The containers store necessary system libraries, tools and code in a file system in order to ensure efficient execution irrespective of the environment[40].

Docker provides an additional layer of abstraction and automation of OS level virtualization on Linux. Docker uses the resource isolation features of the Linux kernel such as the cgroups and kernel namespaces[40].

Docker engine: It is a client-server application that comprises of a server, REST API and a CLI. The server is the daemon process that manages the containers; The REST API specifies the interfaces the programs and daemon and establishes the communication between these two ends. The CLI (command line interface) makes use of the Docker REST API to control or interact with the Docker daemon through scripting[40].

Architecture: It uses client-server architecture. The Docker client communicates with the daemon, which performs significant tasks such as building, running and distributing the containers. The client and daemon can run on the same host or client can connect to a remote daemon[41].

Containers and virtual machines have similar resource isolation and allocation benefits but a different architectural approach. Containers are more portable and efficient. Containers include the application and all of its dependencies but share the kernel with other containers. Virtual machines contain an entire guest operating system along with binaries and libraries that amount to tens of GBs of storage space[41].

4.1.1 Test Bed

Physical Hosts	Dell PowerEdge T110 II Tower server Intel Xeon E3-1230 V2 @3.30GHz 16GB memory, 1TB disk HP Compaq 8300 Elite Micro Tower Intel Core i7-3770 CPU @3.40GHz 16GB memory, 500GB disk
Open vSwitch	Version 2.5.0
Docker	Version 1.12.0
Operating Systems	Ubuntu 14.04 LTS

Table 2 Test Bed Used For Experimentation

4.1.2 Metrics

Throughput: Throughput is the rate of successful packet delivery over a network link. It is used to determine the overall performance of the network devices as well as the capacity of the link between the two end points. It is measured in bits/sec.

This thesis required the measurement of throughput as it gives an outline of the Open vSwitch performance as a bridge connecting two end points and if it acts as a hindrance to achieving line rate performance of a physical switch.

Latency: Latency is the time interval between the source sending a packet to the destination. This is referred to as the One-way latency and if the time taken for the destination to send the packet to the source is also considered then it is referred to as the round trip latency. It is measured as a unit of time

This thesis required the measurement of latency as it gives the operational constraint of Open vSwitch as a software switch. The increase in overhead caused by OVS causes a slight increase in delay and must be measured in order to efficiently analyze the switch performance.

Jitter: Jitter is the difference in delay of received packets. It determines the consistency of the connection between the sender and the receiver. The sender will evenly space out the outgoing packets on the packet stream. It is also measured as a unit of time

This thesis requires the measurement of jitter to analyze the packet handling capacity of OVS during transmission. Each packet being sent out from the egress port on the OVS bridge must reach the destination uniformly as is the ideal case. The variation in packet transmission may be critical in streaming data of larger sizes or even video applications. These services depend on the timely arrival of data without interference.

5 EXPERIMENTATION AND RESULTS

5.1 Experimental Set Up

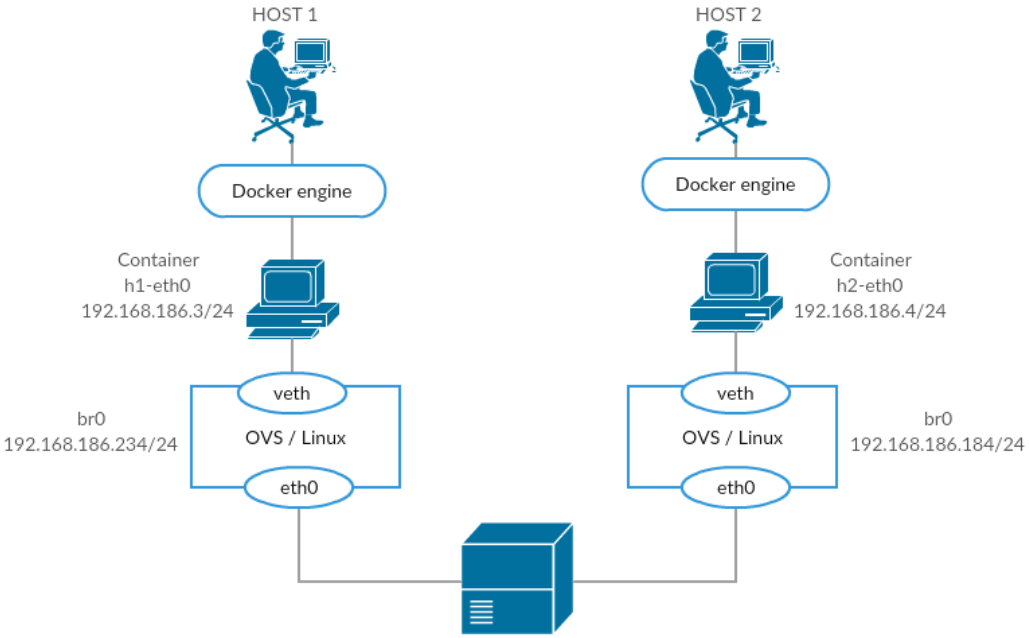

Figure 10 Experimental Set Up with Docker

Figure 10 depicts the experimental setup used in the thesis. The test bed mentions the configuration of each device and software used in the experiments. Two physical hosts were used as a sender and receiver. This model is referred to Client-Server mode.

The container is connected to the OVS bridge/linux bridge by the veth interface. The bridge connects vNIC (i.e. veth) of container to pNIC (i.e. eth0) of host, thereby allowing the container traffic to exit the physical host. The traffic passes through a physical switch to another host and passes to another OVS bridge/ linux bridge that connects pNIC to vNIC and to the receiver container.

Tests are conducted at both end points i.e. from one container to another container. The packets travel from the virtual Ethernet interface (veth) of the container of Host1 through to the Open vSwitch Bridge to the physical interface (eth0) of Host1 to the physical interface (eth0) of Host2 to the Open vSwitch bridge to the virtual ethernet interface (veth) of the Docker container of Host2. The tests are repeated for Linux bridge

The packets will have overhead due to the abstraction of containerization layer in between the two hosts as well as the Open vSwitch bridge/Linux bridge that is handling the forwarding. In addition to the forwarding, the bridge schedules the packets according to the algorithm set in the QOS table for that port interface. Different scheduling algorithms perform differently and handle the packet stream differently.

5.1.1 Tests for each metric on each Scheduling Technique

NetPerf Tests:

TCP_STREAM: This test is used to measure bulk data transfer performance. It is the default test in NetPerf. It is generally used to calculate throughput. It will perform a 10 second test between the local host and remote destination. The socket buffers will be sized according to system defaults or can be manually configured[37].

'Local buffer' refers to send socket buffer size for the connection

'Remote buffer' refers to receive socket buffer size for the connection

The parameters used for the throughput test:

- Local buffer: 32768 bits
- Remote buffer: 32768 bits
- Message size: 256 bits

Multiple iterations had to be taken to get an aggregate result in order to efficiently analyze the system performance. The test was repeated for all scheduling algorithms.

TCP_RR: This test is used to measure Request/Response performance. It is measured as transaction per second. A transaction is an exchange of a single 1-byte request and a 1-byte response. The reciprocal of transaction per second can be used to retrieve round trip latency[37].

The test is run for 100 iterations and the mean value is tabulated.

The parameters used for this latency test:

- Local buffer: 32768 bits
- Remote buffer: 32768 bits

Iperf tests:

UDP STREAM: This test passes stream of UDP packets from client to server and jitter is calculated. The bandwidth and duration of test can be configured manually. The server must be running before the client starts to send packets, and must be checked manually. The server must know that the arriving packets are UDP and the option must be set[39].

At the server, the jitter and bandwidth reached is measured for each run. A single run has duration of 10 seconds, this is repeated for 100 iterations and the overall jitter experienced is averaged and shown at the end. 'UDP buffer' refers to the receiver buffer size of the connection

The parameters used for this jitter test:

- UDP buffer: 208KB
- Datagram: 1470 bytes

5.2 Common Results Between Both Thesis

Common results are taken in context with second companion thesis. VM results are taken from the second companion thesis in order to compare the performance of Open vSwitch with Linux bridge in different virtualization scenarios.

5.2.1.1 Throughput in Docker and VM

Table 3 shows the average throughput measurements obtained from 100 iterations. The test was repeated for all scheduling algorithms in both KVM and Docker connected through Open vSwitch bridge and linux bridge[1]. The obtained results were rounded off to nearest 2 significant digits. From the table it can be seen that all scheduling techniques are almost equal in throughput, slight variation is observed between in both scenarios. Docker performs better than KVM in all scheduling techniques. Open vSwitch offers higher throughput than Linux bridge in KVM but Linux bridge offers higher throughput than OVS in Docker.

Scheduling Algorithms	D0CKER-LINUX	DOCKER-OVS	VM-LINUX	VM-OVS
FIFO	93.46	91.93	87.38	90.82
HFSC	93.41	91.62	87.28	90.81
HTB	93.4	91.51	87.66	91.14
SFQ	93.33	91.63	82.28	90.78
CODEL	93.63	91.77	86.67	90.46
FQ-CODEL	93.71	91.66	86.68	90.91

Table 3 Average Throughput in Docker and VM in Megabits per second (Mbps)

5.2.1.2 Latency in Docker and VM

Table 4 shows the average round trip latency measurements obtained from 100 iterations. The test was repeated for all scheduling algorithms in both KVM and Docker connected through Open vSwitch bridge and linux bridge[1]. The obtained results are rounded off to 3 significant digits for KVM and Docker and 2 significant digits for VM-OVS. From the table it can be seen that the latency in docker containers in all scheduling techniques seem to be constant, whereas in VM's a variant delay is observed in each scheduling technique. In both scenarios Docker performs better than KVM in all scheduling techniques. Ovs offers low latency than Linux bridge in KVM environment whereas in Docker both bridges show similar latencies.

Scheduling Algorithms	DOCKER-LINUX	DOCKER-OVS	VM-LINUX	VM-OVS
FIFO	0.286	0.287	0.509	0.31
HFSC	0.288	0.290	0.513	0.37
HTB	0.284	0.289	0.518	0.36
SFQ	0.285	0.289	0.515	0.35
CODEL	0.284	0.288	0.54	0.38
FQ-CODEL	0.283	0.288	0.514	0.37

Table 4 Average Latency in Docker and VM across different Scheduling Algorithms in milliseconds

5.2.1.3 Jitter in Docker and VM

Table 5 shows the average Jitter measurements obtained from Iperf UDP stream test with a single run of 10 second duration and repeated for 100 iterations in KVM and Docker. The test was repeated for all scheduling algorithms in both KVM and Docker connected through Open vSwitch bridge and linux bridge[1]. The obtained results are rounded off to three significant digits for KVM and four significant digits for docker. From the table it can be seen that Docker has less jitter when compared to that of KVM on OVS. In KVM scenario open v switch shows lower jitter values when compared to linux bridge whereas in docker scenario Linux bridge shows less jitter when compared to OVS.

Scheduling Algorithms	DOCKER-LINUX	DOCKER-OVS	VM-LINUX	VM-OVS
FIFO	0.0025	0.0083	0.016	0.014
HFSC	0.0045	0.0094	0.019	0.018
HTB	0.0048	0.0096	0.026	0.019
SFQ	0.0038	0.0087	0.021	0.015
CODEL	0.0048	0.0042	0.018	0.011
FQ-CODEL	0.0039	0.0048	0.016	0.012

Table 5 Average Jitter in Docker and VM across different Scheduling Techniques in milliseconds

5.3 Individual Results Between Both Thesis

These are the results, which are related to this thesis and will be analyzed separately.

5.3.1.1 Throughput

The table shows the average throughput measurements obtained from 100 iterations. The test was repeated for all scheduling algorithms in Docker connected through Open vSwitch and Linux bridge. From the table it can be seen that on both bridges, throughput varies slightly in all scheduling techniques. Docker on Linux bridge shows higher throughput than that of docker on OVS.

Scheduling Algorithms	DOCKER-LINUX	DOCKER-OVS
FIFO	93.46	91.93
HFSC	93.41	91.62
HTB	93.4	91.51
SFQ	93.33	91.63
CODEL	93.63	91.77
FQ-CODEL	93.71	91.66

Table 6 Average Throughput in Docker across different Scheduling Techniques in Megabits per second

5.3.1.2 Latency

The table shows the average latency measurements obtained from 100 iterations. The test was repeated for all scheduling algorithms. From the table it can be seen that the latency in all scheduling algorithms seems to have constant delay. Docker when connected through both the bridges shows almost similar latencies

Scheduling Algorithms	DOCKER-LINUX	DOCKER-OVS
FIFO	0.286	0.287
HFSC	0.288	0.290
HTB	0.284	0.289
SFQ	0.285	0.289
CODEL	0.284	0.288
FQ-CODEL	0.283	0.288

Table 7 Average Latency in Docker across different Scheduling Techniques in milliseconds

5.3.1.3 Jitter

The table shows the average Jitter measurements obtained from 100 iterations with a single run of 10 second duration. The test was repeated for all scheduling algorithms in Docker connected through Open vSwitch bridge and Linux bridge. From the table it can be seen that Docker has different jitter in different scheduling algorithms. Docker on linux bridge shows less jitter when compared to that of OVS.

Scheduling Algorithms	DOCKER-LINUX	DOCKER-OVS
FIFO	0.0025	0.0083
HFSC	0.0045	0.0094
HTB	0.0048	0.0096
SFQ	0.0038	0.0087
CODEL	0.0048	0.0042
FQ-CODEL	0.0039	0.0048

Table 8 Average Jitter in Docker across different Scheduling Techniques in milliseconds

6 ANALYSIS AND DISCUSSION

This section focuses on the analysis of the measured metrics in each scenario. The entire thesis has been implemented in Docker in this thesis and in KVM in the second companion thesis for both Open vSwitch and Linux bridges. The metrics Throughput, Latency and Jitter have been measured for each scenario to determine Open vSwitch as well as linux bridge performance. A comparative analysis is given by comparing the obtained results of both theses. The following sections will contain explanations to justify the obtained results and to describe the behavior of Open vSwitch in similar scenarios.

6.1 Comparative Analysis for both thesis

Throughput: This is the throughput analysis obtained from running the NetPerf throughput stream test. The test passes stream of TCP packets from sender to receiver for 10 seconds and average throughput is calculated. The test is repeated for 100 iterations and average for 100 runs is taken. 95% confidence interval is considered for each case. The performance is graphed in Figure 11 [1].

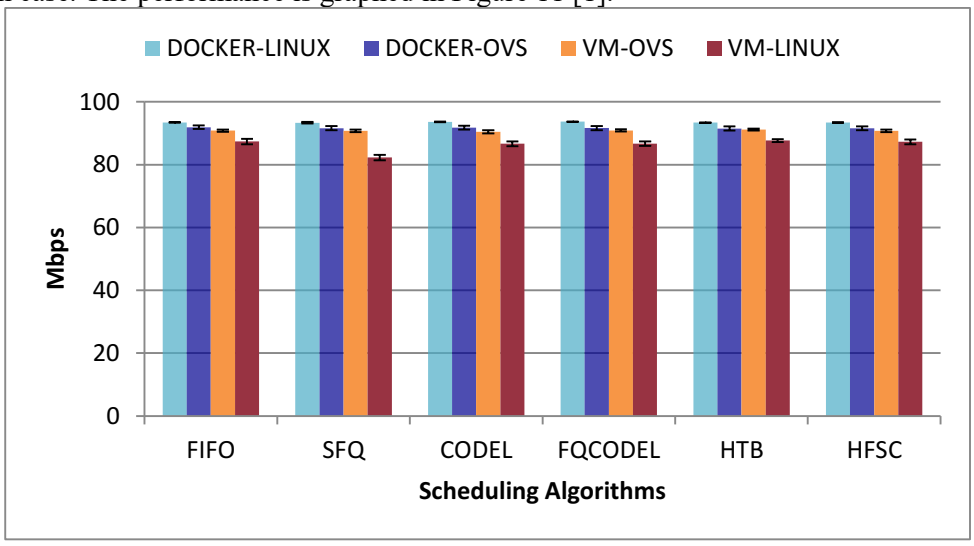

Figure 11 Comparison of Average Throughput in Docker and VM

Figure 11 depicts a change in throughput with virtualization scenario. Open vSwitch bridge performs similar in docker and kvm scenarios. Linux bridge in KVM scenario offered comparatively lesser throughput than other cases. Linux bridge performance in the Docker scenario showed highest throughput measurements.

The physical switch connecting the two hosts has a capacity of 100Mbps. It is practically expected to reach a general throughput close to 100 Mbps. Since it is a software switch that is forwarding the packets between the two hosts, the overhead due to virtualization is expected to reduce the throughput slightly than the maximum possible throughput obtained in bare metal or native scenarios.

Figure 11 clearly shows that Docker performs better than KVM in terms of average throughput. The containers are lightweight and sharing of resources is more efficient. Hence the packet handling capacity of the containers is more efficient than that of KVM VMs. Packets travel from the veth interface to the eth0 interface

(physical interface of the host) faster than from the virtual interface of the KVM guest to the eth0 interface thereby resulting in higher throughput.

Linux bridge offers higher throughput than OVS bridge in docker scenario as implementation is more complex. The ovs-docker python utility consumes more CPU leading to slightly lesser throughput. The default linux bridge implementation consumes lesser CPU leading to higher packet handling by the software switch leading to higher throughput. In KVM Open vSwitch offers slightly higher throughput than linux bridge. Open vSwitch is programmed to handle VM traffic better than the default linux bridge. This is the reason OVS showed slightly higher throughput.

If the precise values of each scheduling algorithm are compared, it can be seen that HTB in both open vswitch and linux bridge performs the best compared to other scheduling algorithms in KVM environment. In Docker environment, CODEL and FQ-CODEL have high throughputs for Open vSwitch and Linux bridge respectively. The optimum performance of the switch in these scheduling techniques is taken into account. All scheduling techniques exhibit similar throughput in their individual scenarios.

Latency: This the round trip latency analysis obtained from running the NetPerf request/response TCP test. The test was conducted over a TCP connection and a 1-byte data is exchanged between the client and server and transaction rate per second is calculated. The reciprocal of transaction rate is taken to give round trip latency. The test is repeated for 100 iterations and average is taken. 95% confidence interval is considered for each case. The performance is graphed in Figure 12 below.

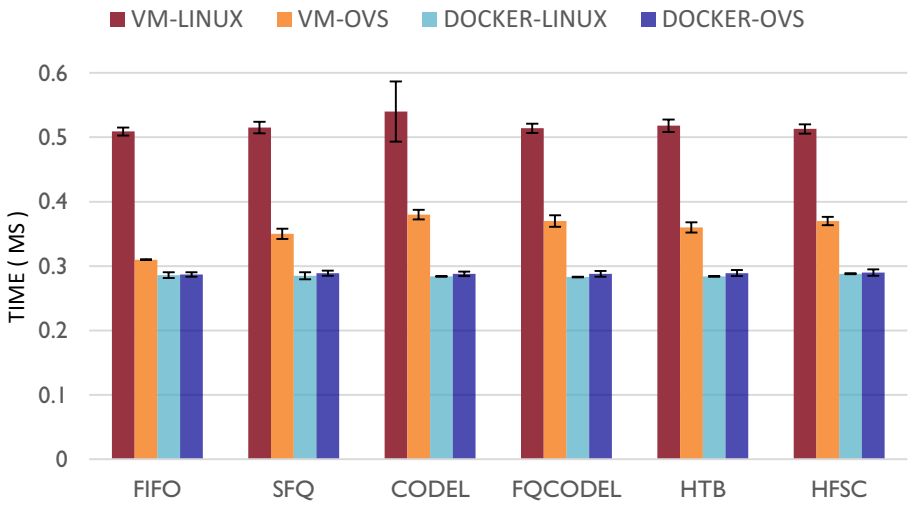

Figure 12 Comparison of Average Latency in Docker and VM

Figure 12 shows a variation in measured round trip latency in both scenarios. Based on the above graph, Docker clearly performs better than KVM in terms of round trip latency. The packet handling is very fast in containers; the resources consumed are significantly lesser contributing to the reduced latency. In KVM, full virtualization causes each virtual machine to contain isolated resources for consumption; this takes comparatively longer time to process contributing to higher latencies.

In KVM, Linux bridge showed higher round trip latency than OVS bridge. Time taken by the Linux bridge to forward traffic is more than the OVS bridge. This can be attributed to the ability of Open vSwitch to handle packets more efficiently than the default Linux bridge.

In Docker environment all scheduling algorithms showed similar latency measurement in both Open vSwitch and Linux bridges. Whereas in KVM environment, for both open vswitch and linux bridges, the VMs showed a varying delay depending on the scheduling technique employed. CODEL showed highest latency in the KVM environment followed by FQCODEL. FIFO showed least delay because of its simple implementation and fast execution. The difference between Linux bridge and OVS bridge for each scheduling algorithm is at least 0.16 ms in CODEL and up to 0.21 ms in case of FIFO.

Jitter: This is the Jitter analysis obtained from running the IPerf UDP Stream test. This was conducted using UDP stream that passes from client to server using the Iperf tool. Each test runs for 10 seconds and average jitter is measured. The test is repeated for 100 iterations and average of 100 runs is taken. The performance is graphed in Figure 13 below.

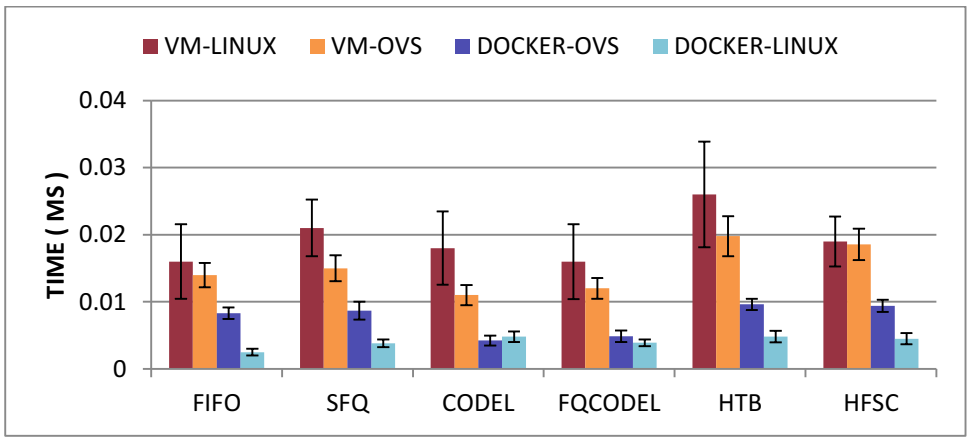

Figure 13 Comparison of Average Jitter in Docker and VM

Figure 13 shows a variation in measured jitter values in KVM and Docker implementation scenarios for both Open vSwitch and Linux bridges. This analysis depicts the consistency of packet forwarding by Open vSwitch and Linux bridge in different scheduling algorithms in different virtualization scenarios.

In this metric as well, Docker outperforms KVM in all scheduling algorithms for both. The memory copying operation that occurs during resource sharing is implemented differently for Docker and KVM. Since KVM is a full virtualization hypervisor, each virtual machine is treated as a separate operating system and is allocated resources by the kernel on boot up. The shared memory between the host and the virtualized guest operates more efficient in Docker than KVM causing difference in jitter. The KVM guests showed more jitter than Docker containers

In the KVM and Docker environments, HTB showed highest jitter along with HFSC for both Open vSwitch and Linux bridges. These algorithms check packet parameters during scheduling, the more that need to be checked for an algorithm, the more the computational time that is required. Since HTB and HFSC require more parameters that need to be checked, they are more complex and need more time to process packets and are slower. The more the number of packets the more the congested queue, the more the jitter. FQ-CODEL and CODEL in all scenarios showed least jitter proving the delay of the packets are being controlled. FIFO showed low jitter measurements. The large 95% confidence intervals indicate a high variation in measurements. SFQ and FQCODEL have multiple queues that implement FIFO internally and hence the jitter is lesser. The large 95% confidence intervals indicate a high variation in measurements.

6.2 Performance Analysis of Open vSwitch in Docker environment

This section outlines the Performance analysis of Open vSwitch in Docker environment. The individual results are explained in detail and analysis is done for each scheduling algorithm in the Docker environment alone. The performance metrics used to determine Open vSwitch performance are Throughput, Latency and Jitter.

Throughput:

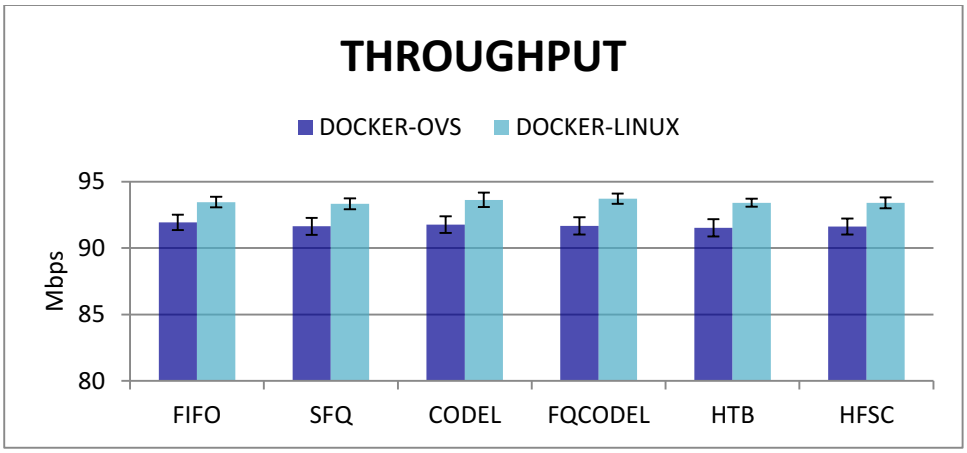

Figure 14 Average Throughputs in Docker for OVS and Linux bridge

Figure 14 displays the difference in throughput measurements for all scheduling algorithms implemented by Open vSwitch and Linux bridge in Docker environment. Linux bridge shows higher throughput when compared to Open vSwitch bridge.

The throughputs for different algorithms only vary slightly for both the bridges. This is because, the physical switch supports a certain bandwidth and Open vSwitch can send as many packets as the switch can handle.

The scheduling algorithms were compared in their ability to achieve bitrates that are closest to line rate. Since the switch has a capacity of 100 Mbps the performance of Open vSwitch is good. The main objective of any software switch is to achieve minimum resource consumption in order to allow maximum resources to be available for the container engine to process the user workloads, which are of higher priority. The above results indicate that Open vSwitch and Linux bridge performs significantly good in terms of reaching that goal of near-line-rate performance.

Latency:

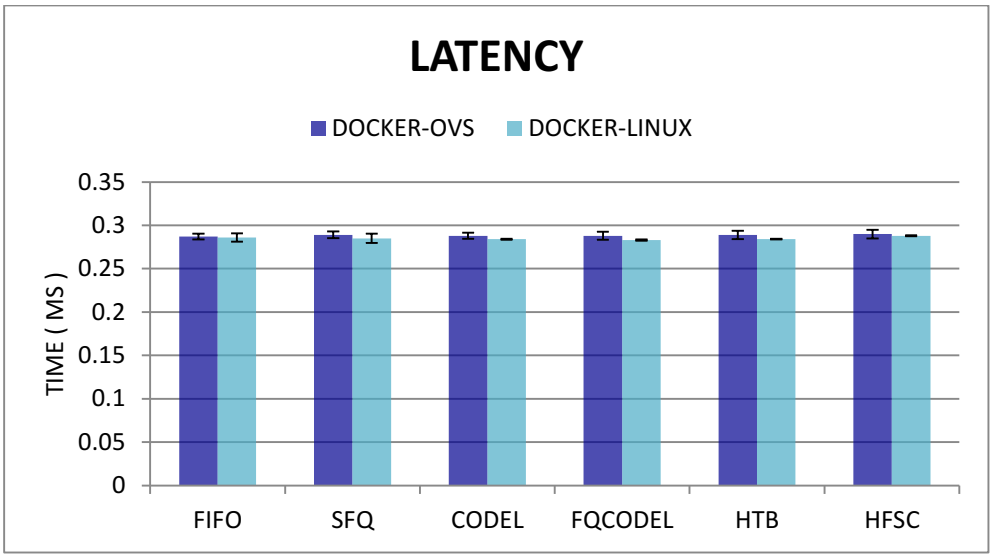

Figure 15 Average latency in Docker for OVS and Linux bridges

Figure 15 displays the difference in latency measurements for all scheduling algorithms implemented by Open vSwitch and Linux bridge in Docker environment. Different scheduling algorithms show similar performance in terms of round trip latency for both Open vSwitch and Linux bridges. The scheduling algorithms are compared in their ability to send packets from one host to another in the shortest time possible. Thereby the ability of Open vSwitch to implement these scheduling algorithms efficiently is determined.

FIFO has the least round trip latency measured on Open vSwitch bridge, it is the simplest as well as default qdisc implemented on an interface by the Linux kernel. This functioning of the algorithm is hard wired by the OS and its processing therefore takes the least computational time.

FQ-CODEL has the least round trip latency measured on Linux bridge as it controls the delay of each packet passing through the queue. FQCODEL and CODEL performed similarly in terms of round trip latency. Higher latencies were seen in HFSC and SFQ but all scheduling algorithms are almost similar in both the bridges.

Jitter:

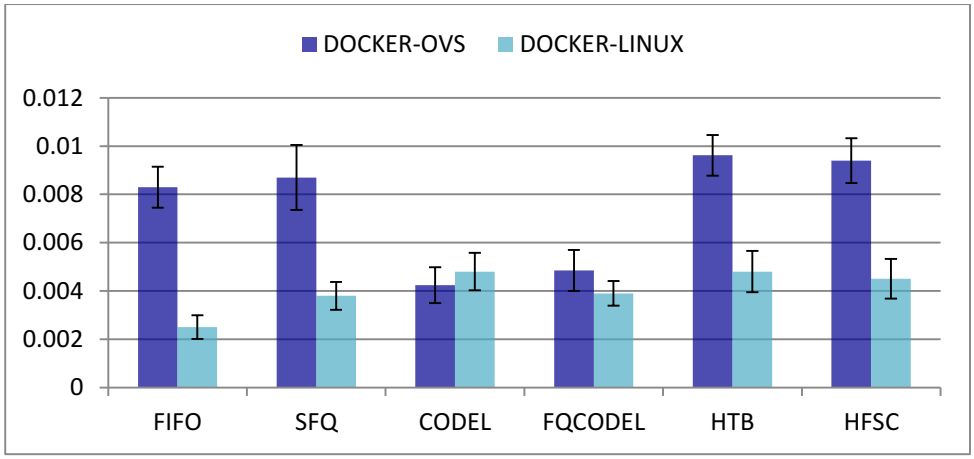

Figure 16 Average jitter in Docker on OVS and Linux bridges

Figure 16 displays the difference in jitter measurements for all scheduling algorithms implemented by Open vSwitch and Linux bridge in Docker environment. Different scheduling algorithms behave differently in terms of jitter. The scheduling algorithms are compared in its ability to send packets consistently with least variation in packet transmission. Thereby the ability of Open vSwitch to implement these scheduling algorithms efficiently is determined.

From Figure 16, it is clear that each scheduling algorithm has a different jitter measurement. The difference in jitter can be related to each scheduling algorithm complexity during default parameter implementation. Since each algorithm is executed in default parameters for 1 flow this is the basic implementation for the scheduling algorithm without any extra configurations.

HTB and HFSC on OVS bridge have higher inconsistencies in transmitting packets due to the nature of their scheduling algorithms. They require more parameters to be checked than classless algorithms. The target bandwidth was high at 95 Megabits per second and so the number of packets being handled is very high. The simpler algorithms showed lower jitters as the logic involved to transmit packets are simpler and therefore transmitting packets at equal times was more achievable in the case of simpler algorithms.

CODEL and FQCODEL on OVS bridge had the least jitter observed in the 100 second UDP stream test. The technique involves usage of hash function to distribute the packets into separate FIFOs. The dequeuer used is Round Robin. Therefore, the jitter exhibited by the system is least. FIFO algorithm simply forwards the packets sequentially as they come irrespective of the number of packets being handled. FIFO and FQCODEL have similar jitter measurements.

HTB and CODEL on default linux bridge shows similar jitter performance and when compared to other scheduling algorithms these two algorithms show more jitter. FIFO and SFQ show least jitter.

7 CONCLUSION AND FUTURE WORK

This section outlines the conclusion and inferences from the research. It provides a section for the possible future work that can be conducted to contribute to the field of study of Network Virtualization and improve the study of software switch performance. It answers the previously mentioned research questions individually and provides insight into the motivation and contribution of the thesis.

7.1 Conclusion

The aim of the thesis was to analyze the performance of Open vSwitch in Docker virtualization scenario. The results obtained from measuring metrics such as Throughput, Latency and Jitter have given an insight into the performance of Open vSwitch in this virtualization scenario. The performance of default Linux bridge is also measured and analyzed in comparison to OVS in order to determine whether Open vSwitch is an enhanced software switch.

From the combined results in both the theses it is inferred that Docker performs better than KVM for both Open vSwitch and Linux bridges irrespective of the scheduling technique employed. The Docker environment in itself performs faster and more efficient than full virtualization KVM hypervisor. The resource sharing in KVM causes isolation in CPU, memory, I/O as well as network resources this causes the difference in performance between the two scenarios. Resource sharing in Docker is less isolated and hence is easily processed. It contributes to the speedy processing of packets in each scheduling algorithm. In case of Docker, Linux bridge perform better than Open vSwitch bridge whereas in case of KVM, OVS bridge performed better.

In the Docker scenario analysis, it was seen that throughout measurements were similar in all scheduling algorithms. Default Linux bridge performed better than OVS bridge in terms of throughput. All algorithms performed similarly in terms of achieved bitrate, which was considerably close to line rate performance. The difference in throughput between all scheduling algorithms is very small in the range of 0.5 Mbps and it can be said that all scheduling algorithms have the same bitrate i.e. close to 93Mbps, which is almost line rate performance. So the Open vSwitch performs better in all scheduling algorithms.

In the round trip latency measurements, both the bridges show almost similar performance. It was seen that FIFO had the least latency measured on open vswitch bridge owing to its simple execution and scheduling logic. FQ-CODEL has the least round trip latency measured on Linux bridge as it controls the delay of each packet passing through the queue. HFSC shows higher latency on both bridges.

In the jitter measurements, it was seen that HTB and HFSC on OVS bridge showed highest jitter. CODEL and FQCODEL showed least jitter. HTB on linux bridge exhibited highest jitter. When compared to both bridges docker on OVS showed highest jitter than that of linux bridge.

Research questions

1. **How can we measure the network performance of Open vSwitch in Docker environment?**

Open vSwitch must be installed and configured on the host. The aim is to connect the containers directly to the OVS bridge. Docker container engine attempts to connect the launched container to the docker0 Linux bridge by default. This procedure has to be bypassed in order to measure the exact performance of container-to-container communication via OVS bridge. Additional overhead would be incurred if packets passed from container to Docker Bridge to OVS Bridge. Using ovs-docker utility provided by the Open vSwitch contributors, python script must be compiled in the Open vSwitch execution directory to enable this feature. Using those commands, the Docker container can directly connect to the OVS Bridge with static IP address. The IP address must lie within the subnet of the OVS bridge interface for obvious reasons. Once the containers connect to the OVS bridge performance can be measured using benchmark tools to perform tests and collect metric measurements. If the packets have to leave that physical system, then the physical NIC of the host must be added as a port on the OVS Bridge and eth0 IP must be given to the bridge interface. Packets can travel from the container veth interface to the eth0 interface of Host1 to the eth0 interface of Host2 to the container veth interface.

2. **How can we model the performance of an Open vSwitch with respect to scheduling technique employed?**

Once the Open vSwitch Bridge is configured and the Docker Containers are connected to separate ports of the Open vSwitch Bridge, packets can be transmitted from one host to another. Packet scheduling is implemented in OVS by configuring the particular port at the QoS. A port implementing a particular QoS policy references the QoS table. The user has to manually set the port to a particular queuing discipline, and that Port will point to particular QoS entry in the QoS table. Each QoS entry can point to a queue with configured maximum and minimum rates to control flow of data within the configured port. To measure the performance of Open vSwitch with respect to a scheduling technique, the sender port and the receiver port have to configure at their respective QoS tables and each port must point to its own queue configuration. The packets will be scheduled based on the scheduling algorithm implemented. Different scheduling algorithms perform differently in different conditions based on the logic involved.

3. **How can one determine network performance of Open vSwitch in different virtualization scenarios? Does it vary with different virtualization scenarios?**

Open vSwitch is an efficient virtual switch that is compatible with the latest virtualization trends. It has the capacity to facilitate inter VM traffic (for KVM) as well as inter container traffic (for Docker). But, different virtualization techniques have different forwarding mechanisms due to different resource emulation by the hypervisor, container engine. Measuring the network performance when OVS is

connected two KVM guests and when connecting two docker containers gives us the ability of OVS to handle packets in these two virtualization environments.
Yes, different virtualization environments perform differently in terms of resource sharing and level of abstraction. KVM is a full virtualization hypervisor that emulates the hardware resources into virtual resources for the virtual machines. Each virtual machine is an operating system into itself and functions as a separate entity with isolated resources. Whereas Docker container engine shares the kernel with other containers. They are simply lightweight OSes that run on images of actual operating systems. They are easier to deploy and hence perform better in all aspects. This is seen in the comparison of the results between the two theses.

7.2 Future Work

In the future, the analysis can be taken to the next step by employing more efficient scheduling algorithms that are more fit for heterogeneous environments. Algorithms that can efficiently handle packets of higher bit size as well as maintain load. Different algorithms can be used at the same time at the sender and the receiver to see performance of enqueuer of one scheduling algorithm and the dequeuer of another scheduling algorithm. Recent developments have been made to improve software switches by using hardware acceleration techniques such as DPDK. The analysis can be done using such hardware acceleration techniques. Open vSwitch is a very efficient software switch but improvements can be made in its packet handling techniques. Jitter measurements suggest that some scheduling techniques perform poorly under higher packet sizes. The QoS policing can be improved and buffer management can be optimized.

REFERENCES

[1] R. POTHURAJU, "MEASURING AND MODELLING THE PERFORMANCE OF OPEN V-SWITCH ,Implementation in KVM," Student Thesis, Blekinge Institute Of Technology, Karlskrona, 2016.

[2] "What are the disadvantages of FCFS scheduling algorithm? - Engineering Questions Answers QnA - Agricultural, Electrical, Civil, Computer, Mechanical." [Online]. Available: http://www.enggpedia.com/answers/1697/what-are-the-disadvantages-of-fcfs-scheduling-algorithm. [Accessed: 05-Sep-2016].

[3] "Network function virtualization," *Wikipedia, the free encyclopedia*. 03-Sep-2016.

[4] "Software-defined networking," *Wikipedia, the free encyclopedia*. 02-Sep-2016.

[5] "What is Software Switch? - Definition from Techopedia," *Techopedia.com*, 05-Sep-2016. [Online]. Available: https://www.techopedia.com/definition/25852/software-switch. [Accessed: 05-Sep-2016].

[6] "Open vSwitch." [Online]. Available: http://openvswitch.org/. [Accessed: 13-Sep-2016].

[7] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, 2014, pp. 120–125.

[8] Valerie Young, "ETSI GS NFV-INF 004 V1.1.1," *ETSI GS NFV-INF 004 V1.1.1*, 11-Jan-2016. [Online]. Available: https://www.google.se/search?q=etsi+hypervisor+domain&oq=etsi+hy&aqs=chrome.0.69i59j69i57j0l4.1595j0j9&sourceid=chrome&es_sm=93&ie=UTF-8. [Accessed: 11-Jan-2016].

[9] L. Xingtao, G. Yantao, W. Wei, Z. Sanyou, and L. Jiliang, "Network virtualization by using software-defined networking controller based Docker," in *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, 2016, pp. 1112–1115.

[10] "openvswitch/ovs," *GitHub*, 05-Sep-2016. [Online]. Available: https://github.com/openvswitch/ovs. [Accessed: 05-Sep-2016].

[11] B. Pfaff *et al.*, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2015, pp. 117–130.

[12] "Ubuntu Manpage: ovs-vswitchd - Open vSwitch daemon." [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man8/ovs-vswitchd.8.html. [Accessed: 05-Sep-2016].

[13] "Ubuntu Manpage: ovsdb-server - Open vSwitch database server," 05-Sep-2016. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/ovsdb-server.1.html. [Accessed: 05-Sep-2016].

[14] "INSTALL.md (Open vSwitch 2.5.90)," 05-Sep-2016. [Online]. Available: http://openvswitch.org/support/dist-docs/INSTALL.md.html. [Accessed: 05-Sep-2016].

[15] "7. Queue Management and Packet Scheduling," 05-Sep-2016. [Online]. Available: http://www.isi.edu/nsnam/ns/doc/node64.html. [Accessed: 05-Sep-2016].

[16] "Linux Packet Scheduling [OpenWrt Wiki]," 05-Sep-2016. [Online]. Available: https://wiki.openwrt.org/doc/howto/packet.scheduler/packet.scheduler.theory. [Accessed: 05-Sep-2016].

[17] T.-Y. Tsai, Y.-L. Chung, and Z. Tsai, "Introduction to Packet Scheduling Algorithms for Communication Networks," in *Communications and Networking*, J. Peng, Ed. Sciyo, 2010.

[18] "6. Classless Queuing Disciplines (qdiscs)," 05-Sep-2016. [Online]. Available: http://linux-ip.net/articles/Traffic-Control-HOWTO/classless-qdiscs.html. [Accessed: 05-Sep-2016].

[19] “7. Classful Queuing Disciplines (qdiscs),” 05-Sep-2016. [Online]. Available: http://linux-ip.net/articles/Traffic-Control-HOWTO/classful-qdiscs.html. [Accessed: 05-Sep-2016].
[20] “tc-codel(8) - Linux manual page,” 05-Sep-2016. [Online]. Available: http://man7.org/linux/man-pages/man8/tc-codel.8.html. [Accessed: 05-Sep-2016].
[21] “tc-fq_codel(8) - Linux manual page,” 05-Sep-2016. [Online]. Available: http://man7.org/linux/man-pages/man8/tc-fq_codel.8.html. [Accessed: 05-Sep-2016].
[22] “HTB manual - user guide,” 05-Sep-2016. [Online]. Available: http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm. [Accessed: 05-Sep-2016].
[23] “HFSC Scheduling with Linux,” 05-Sep-2016. [Online]. Available: http://linux-ip.net/articles/hfsc.en/. [Accessed: 05-Sep-2016].
[24] “tc-hfsc(7) - Linux manual page,” 05-Sep-2016. [Online]. Available: http://man7.org/linux/man-pages/man7/tc-hfsc.7.html. [Accessed: 05-Sep-2016].
[25] Vandyblog, “Deep Dive: HTB Rate Limiting (QoS) with Open vSwitch and XenServer,” *Virtual Andy*, 29-Apr-2013. .
[26] “Ubuntu Manpage: A database with this schema holds the configuration for one Open,” 05-Sep-2016. [Online]. Available: http://manpages.ubuntu.com/manpages/precise/man5/ovs-vswitchd.conf.db.5.html. [Accessed: 05-Sep-2016].
[27] D. Stezenbach, K. Tutschku, and M. Fiedler, “A Performance Evaluation Metric for NFV Elements on Multiple Timescales,” presented at the Global Communications Conference (GLOBECOM), 2013 IEEE, Atlanta, USA, 2013.
[28] Y. Zhao, L. Iannone, and M. Riguidel, “Software Switch Performance Factors in Network Virtualization Environment,” in *2014 IEEE 22nd International Conference on Network Protocols (ICNP)*, 2014, pp. 468–470.
[29] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, “Performance Evaluation of Microservices Architectures using Containers,” *ResearchGate*, Nov. 2015.
[30] C.-N. Mao *et al.*, “Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks,” 2015, pp. 611–616.
[31] J. C. Wang, W. F. Cheng, H. C. Chen, and H. L. Chien, “Benefit of construct information security environment based on lightweight virtualization technology,” in *2015 International Carnahan Conference on Security Technology (ICCST)*, 2015, pp. 1–4.
[32] S. Shanmugalingam, A. Ksentini, and P. Bertin, “DPDK Open vSwitch performance validation with mirroring feature,” in *2016 23rd International Conference on Telecommunications (ICT)*, 2016, pp. 1–6.
[33] “Qualitative vs Quantitative,” 05-Sep-2016. [Online]. Available: http://www.sal.tohoku.ac.jp/ling/corpus3/3qual.htm. [Accessed: 05-Sep-2016].
[34] “Qualitative vs Quantitative Data | Simply Psychology,” 05-Sep-2016. [Online]. Available: http://www.simplypsychology.org/qualitative-quantitative.html. [Accessed: 05-Sep-2016].
[35] “WSZiB : Lectures of Prof. dr Peter Sloot,” 05-Sep-2016. [Online]. Available: http://artemis.wszib.edu.pl/~sloot/1_2.html. [Accessed: 05-Sep-2016].
[36] “Benchmark (computing),” *Wikipedia, the free encyclopedia*. 10-Jul-2016.
[37] “netperf(1): network performance benchmark - Linux man page,” 05-Sep-2016. [Online]. Available: http://linux.die.net/man/1/netperf. [Accessed: 05-Sep-2016].
[38] “iPerf - The TCP, UDP and SCTP network bandwidth measurement tool,” 05-Sep-2016. [Online]. Available: https://iperf.fr/. [Accessed: 05-Sep-2016].
[39] “IPERF - The Easy Tutorial,” 05-Sep-2016. [Online]. Available: https://openmaniak.com/iperf.php. [Accessed: 05-Sep-2016].
[40] “What is Docker? | Opensource.com,” 05-Sep-2016. [Online]. Available: https://opensource.com/resources/what-docker. [Accessed: 05-Sep-2016].

[41] “Docker Overview,” *Docker*, 30-May-2016. [Online]. Available: https://docs.docker.com/engine/understanding-docker/. [Accessed: 05-Sep-2016].