NTNU

Norwegian University of
Science and Technology

Testbed-based Evaluation of QoS Differentiation for Network Slicing in Multi-Application Scenarios

Subedi, Raj Kumar

Submission date:	July 2023
Main supervisor:	Zinner, Thomas, NTNU
Co-supervisor:	Lange, Stanislav, NTNU

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Testbed-based Evaluation of QoS Differentiation for Network Slicing in Multi-Application Scenarios
Student:	Subedi, Raj Kumar

Problem description:

Today's modern shared network infrastructures need to serve unpredictable heterogeneous services requiring distinct network resources such as bandwidth, computation, storage, and so on. We also need to provide isolation of traffic of such services to implement policies for security implementation and service level agreements. Especially at the data center and in the enterprise network, there are large aggregate flows of various traffic that demand specific levels of network resources to function satisfactorily. Thus, managing available network resources among the dynamic applications requesting services from the respective servers and proper isolation between the application is a challenge to avoid the addition of physical infrastructure. At the same point, optimizing the allocation of available network resources in such multi-applications scenarios is imperative to guarantee a certain level of performance for each service.

Without installing new instruments to integrate every time new services emerge, Slicing is a potential enabler to perform differentiated traffic treatment and deal with the outlined heterogeneity in an efficient way. Utilities such as the Intel Data Plane Development Kit (DPDK) Quality of Service (QoS) framework are a potential tool to implement or emulate the aspects of slicing, but uncertainty regarding the applicability, feasibility, flexibility, and performance of the DPDK QoS framework for the above task remain and are hence the subject of this thesis.

Approved on:	2023-01-20
Main supervisor:	Zinner, Thomas, NTNU
Co-supervisor:	Lange, Stanislav, NTNU

Abstract

In today's heterogeneous network environments, different applications have varying network resource requirements, such as throughput, latency, and reliability. Traditional approaches to addressing this heterogeneity involve adding additional infrastructure to provide separate resources for each application, ensuring QoS but increasing installation and maintenance costs. Moreover, scalability becomes a challenge as the number of users and emerging applications grow, requiring constant reconfiguration and infrastructure expansion. To tackle these issues and reduce costs while accommodating heterogeneous services within a shared physical network infrastructure, network slicing has emerged as a promising solution. Network slicing allocates physical resources to specific services or groups of services, ensuring guaranteed network resources even during peak periods. However, implementing network slicing comes with implementation complexities and trade-offs in terms of cost, deployment complexity, and system performance.

This thesis aims to investigate a network slicing technique that shares network resources among slices to provide QoS for each service. Various methods for implementing network slicing exist, including client service division into slices in software-defined network platforms and hierarchical scheduling in hierarchical QoS (HQoS) approaches. We focus on utilizing the DPDK QoS framework for network slicing. We employ a hierarchical token bucket queue and define rules to identify and assign traffic to specific slices. Our approach leverages the DPDK QoS framework in a testbed where a Linux server hosts heterogeneous services in Docker containers, accessed by numerous clients. We fine-tune the control knobs of the DPDK QoS framework and monitor traffic patterns to evaluate its characteristics and efficiency in utilizing shared physical network resources in a heterogeneous applications environment. The contributions of this thesis include the development of a physical testbed for multi-application environments with traffic differentiation, facilitating reproducible and extensible experiments. Additionally, a feasibility study is conducted to assess the effectiveness of the DPDK QoS framework for achieving network slicing and ensuring slice isolation. These contributions advance the understanding and practical implementation of network slicing techniques, aiding in resource allocation optimization, QoS assurance, and enhanced efficiency of shared physical network infrastructure in heterogeneous multi-application environments.

Contents

List of Figures

List of Acronyms

API Application Programming Interface.

ASICs Application Specific Integrated Circuit.

Bpp Bytes per packet.

Bps Bytes per second.

bps bits per second.

CMDB Configuration Management Database.

COTS Commercial Off-The-Shelf.

CPU Central processing unit.

DASH Dynamic Adaptive Streaming over HTTP.

DMA Direct Memory Access.

DPDK Intel Data Plane Development Kit.

DPI Deep Packet Processing.

DUT Device-under-Test.

EAL Environment Abstraction Layer.

FPGAs Field Programmable Gate Array.

GbE gigabit Ethernet.

GBPS Gigabit per second.

HQoS Hiererchical Quality of Serivce.

HTB	Hierarchical token bucket.
HTTP	Hypertext Transfer Protocol.
INT	Integer.
IoT	Internet of Things.
IP	Internet Protocol.
KB	Kilo byte.
MB	Mega Byte.
Mbps	Megabit per second.
mbufs	message buffers.
Mpps	Mega or million packet per second.
MTU	Maximum transmission unit.
NFV	Network Functions Virtualization.
NIC	Network Interface Card.
NUMA	Non Uniform Memeory Access.
P4	Programming Protocol-independent Packet Processors.
PDV	Packet Delay Variation.
PMD	Poll Mode Driver.
pps	Packet per second.
PTE	Page Table Entry.
QoE	Quality of Experience.
QOS	Quality of Service.
QoS	Quality of Service.
RAM	random-access memory.
RED	Random Early Detection.

RX Receiver.

SDN Software Defined Networking.

SDNPS Software Defined Networks with P4 switches.

SLAs service level agreements.

TB Tocken Bucket.

TC Traffic Class.

TCP Transmission Control Protocol.

TLB Translation Lookside Buffer.

TX Transmitter.

UDP User Datagram Protocol.

VALE Virtual Local Ethernet.

VLAN Virtual Local area Network.

VoD Video On Demand.

VoIP Voice over IP.

WRR Weighted Round Robin.

YAML Yet another markup language.

Chapter 1
Introduction

In a heterogeneous network, applications have different network resource requirements: capacity, processing time, storage, and so on which are measured by metrics such as reliability, throughput, delay, jitter, availability, or a combination of these metrics. For instance, Video On Demand (VoD) streaming services require larger throughput, Voice over IP (VoIP) services require low latency, Internet of Things (IoT), where many sensor nodes possibly hundreds of thousands are connected to a network, need massive low bandwidth connections, secure shell for remote network management require low bandwidth, high reliability, conference calls demand high bandwidth, high reliability, and low latency, and so on.

To deal with the heterogeneity to provide optimum security and resource allocations to the services, we can add additional infrastructure to our existing network infrastructure and provide separate resources to applications. This can solve the interference of the traffic as the services have their own routes and no fighting for the resources, meaning QoS is maintained for individual applications. However, it increases the cost of installation and maintenance of additional devices, and the dynamic nature of users causes the resources to be idle for inactive periods because the resources are already allocated for the services whether the application is active or inactive. The most important problem is scalability. We cannot predict the number of users requesting a service at any interval of time and an emerging application with distinct network resource requirements in modern technology, meaning growing users and a new service requires the addition of network infrastructure and their re-configuration, and adding policies for achieving satisfactory user experience.

Another promising technique for solving all those issues including the reduction of costs and coping with heterogeneous services in a shared physical network infrastructure is network slicing. Network slicing is a technique that allows the allocation of physical resources to a specific service or a group of services in a heterogeneous network so that a service is guaranteed to receive network resources once admitted to the network even at the overloading period with a set of rules to identify traffic. This technique comes with the complexity of implementation as various technologies have been introduced; however, fine-tuning the implementation, and selecting one that best suits a network certainly optimize the efficiency of resources allocation and their usage and reduce the cost of expanding physical network infrastructure. This work purpose a technique of network slicing where the network resources are shared among the slice to provide QoS for each service. A user is concerned about the network when the user does not receive a satisfactory experience from the service which is latency in a voice call, throughput in VoD, and so on which can be solved by using QoS. QoS is a mechanism that ensures the performance of an application with limited resources in a shared network.

There are various methods for implementing network slicing in different technologies. Hiererchical Quality of Serivce (HQoS) uses hierarchical scheduling, dividing services among different slices in the hierarchy. The set of rules to identify traffic and implementation of network slice comes with its corresponding trade-offs regarding cost, the complexity of deployments, and the performance of the overall system. The simulation studies [1] shows promising results regarding the use of HQoS for Quality of Experience (QoE)-aware resource allocation where common properties of services are divided among slices, and services in a slice have resource sharing based on pre-defined rules, while perfect isolation of network resources between each slice. To maximize the benefits of such slicing mechanisms, several issues need to be addressed: the relationship between allocated resources and the resulting user-perceived service quality, the degree of resource isolation between slices, and the fairness in terms of service quality under limited resource availability.

In this work, we use DPDK QoS framework [2] for the specific aspect of link capacity slicing, which uses a hierarchical token bucket queue. We define a set of rules to identify traffic and sort them onto slices. In the testbed Fig 1.1, the Linux server is running QoS framework provided by DPDK for QoS implementation for heterogeneous services hosted in docker containers which are being accessed by numerous clients. We tune the control knobs of QoS framework provided by DPDK and the service-specific parameters and monitor the traffic patterns and evaluate the characteristic of DPDK QoS for the efficiency of shared physical network resources in a heterogeneous applications environment.

1.1 Objectives

The main objectives of this thesis work are as follows:

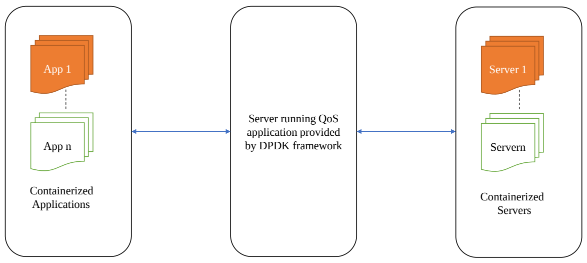
App 1
App n
Containerized Applications
Server running QoS application provided by DPDK framework
Server 1
Servern
Containerized Servers

Figure 1.1: General Architecture for testbed

1. Physical testbed for multi-application environments with traffic differentiation, reproducible and extensible experiments.

2. Feasibility study regarding the use of the DPDK QoS framework to achieve network slicing and slice isolation.

3. Guidelines for appropriately configuring the DPDK QoS framework to achieve QoS for heterogeneous applications in a shared network infrastructure.

1.2 Thesis Contribution

The details of the achieved contribution are demonstrated in the later chapters of this thesis report. The main contribution can be summarized as follows: We evaluate the performance of QoS framework provided by DPDK in reproducible heterogeneous services that share a network infrastructure. There are various parameters from service levels to the framework to tune for which the system behaves differently. From the documentation of the framework, we measure, evaluate, and validate the traffic pattern by tuning such control knobs and finally conclude the usability of the framework to provide QoS for different applications in a shared network infrastructure.

1.3 Thesis Structure

This project contains a total of 5 chapters including this current introductory first chapter. A brief description of further chapters in this report is as follows:

1. **Chapter 2: Background and Literature Review**
   This Chapter gives an overview of the background and describes various terminologies used in this project.
2. **Chapter 3: Methodology**
   This chapter provides design goals, components, parameters, and metrics for the physical setup for the experiments.
3. **Chapter 4: Evaluation**
   This chapter contains evaluations of the data resulting from tuning various parameters in the experiments.
4. **Chapter 5: Conclusion**
   This chapter summarizes the complete project thesis along with the challenges and limitations that occur in the process.

Chapter 2
Background and Literature Review

This chapter provides a brief introduction to the previous work that enlightens the idea behind this thesis. This chapter also covers a general introduction for understanding the terms and technologies used to achieve the goals.

2.1 QoS-Based Traffic Handling and Network Slicing

Ensuring QoS is crucial in a shared network infrastructure composed of heterogeneous applications for several reasons. Firstly, QoS directly impacts user satisfaction by delivering a consistent and high-quality experience, regardless of network conditions or concurrent users. Secondly, different applications have unique performance requirements, and QoS ensures that they can operate optimally by meeting their specific needs, whether it's low latency for real-time communication or high throughput for data-intensive tasks. Additionally, QoS is necessary to fulfill service level agreements (SLAs) and avoid penalties or contract terminations. It also plays a vital role in resource management, ensuring fair allocation and preventing resource monopolization [3]. Lastly, QoS mechanisms can contribute to maintaining security and isolation, protecting sensitive data, and preventing unauthorized access. Overall, QoS is essential to optimize performance, satisfy users, meet contractual obligations, manage resources efficiently, and enhance security in shared network infrastructure with heterogeneous applications [4].

Network Slicing We need to adapt our existing shared network infrastructure that supports many different subscriber types with diverse and sometimes contradictory requirements, and varying application usage. Rather than applying a monolithic network serving multiple purposes, virtualization and Software Defined Networking (SDN) allow us to define a logical network on top of underlays network infrastructure, called network slices. Network slicing is a technique that overlays multiple unique logical and virtualized networks [5] over a shared network domain that consists of a set of shared network and computing resources. Traditionally, a virtual private network in an Ethernet network implements slicing which provides only logical connectivity-oriented slicing, but nowadays we use various technologies namely SDN [6], Network Functions Virtualization (NFV), orchestration, and automation to generate network slices that provide QoS and other resources for their own logical topology, security rules, and performance characteristics impose by the limit of shared network infrastructure. Various slices can be implemented for different purposes, such as ensuring specific applications have high-priority access to capacity and latency, best-effort delivery, isolating traffic for specific users or services, and so on. The idea behind the network slice is to create virtual networks on top of shared physical infrastructure isolating distinct traffic flows.

Some properties of network slices are isolation, scalability, and usability. First, there is the isolation of connectivity and isolation of performance between sets of users/services/applications [7]. In the isolation of connectivity, connectivity within a slice is managed within it but between slices needs external intervention outside the slices. Performance is an end-to-end issue from application to server, hence managing the impact of resource sharing needs to be addressed appropriately, even when it crosses the management domain. Secondly, there must be methods for the assurance of isolation without which it can not be trusted. It depends on the regulatory framework and level of trust requirement; ranging from medical services to watching video-on-demand entertainment content which obviously has different requirements for trust. Thirdly, for scalability, isolation deployment needs to be with minimal effort, otherwise, the costs outweigh the benefits. Heterogenous services have distinct requirements of network services that are expressed at a high level, with lower-level configurations automatically generated to deliver the isolation requested. Fourthly, good slicing should cover the full range of use cases. The goal should be delivering a set of capabilities that applications can reuse rather than re-inventing. For instance, an operator of the domain-specific slice (e.g. banking applications) can add value by aggregating network slice capabilities across multiple providers and enforcing policies that are appropriate to that domain.

There are different components of QoS internal function. Some of them are briefly discussed below.

2.1.1 Traffic Shaping in QoS

Traffic shaping [8] in QoS is a technique used in QoS to control the flow of ingress network traffic. It regulates the rate at which packets are processed for the scheduling stage of QoS to prevent congestion and optimize resource utilization. By enforcing specific traffic profiles, traffic shaping smoothes outbursts of traffic and ensures a consistent flow, improving overall network performance and ensuring fair allocation of resources.

2.1.2 Traffic Policing in QoS

Traffic policing [9] is another mechanism employed in QoS to enforce compliance with defined traffic policies and limits. It monitors incoming traffic and takes action when it exceeds specified thresholds. Policing can involve dropping or marking packets to prioritize traffic, control bandwidth usage, and prevent congestion. By enforcing traffic policies, traffic policing helps maintain network performance and ensure that traffic conforms to desired parameters.

2.1.3 Hierarchical Token Bucket

HTB is a traffic-shaping algorithm used in computer networks to control and manage the flow of packets. It is a mechanism that allows network administrators to prioritize and allocate capacity to different classes or categories of traffic. HTB implements the concept of a token bucket along with a class-based hierarchical system and filters to supervise complex and granular control over packet flows [10]. A token bucket is a conceptual bucket that holds a certain number of tokens. Tokens are "units of permission" that determine whether a packet can be transmitted or not. HTB allows users to customize the tokens and buckets and allows the user to nest the buckets in an arbitrary fashion with rate and ceil where rate defines the assured data rate, while ceil defines the peak data rate allowed for the flow. In HTB, classes are configured as a tree according to the relationship of traffic aggregation and only leaf classes have a queue to buffer packets belonging to the class. Children's classes borrow tokens from their parents when their packet flow exceeds their rate. A child continues to attempt to borrow until it reaches ceil, at which point it begins to queue packets for transmission [11].

HTB can emulate link slicing which is a subset of slicing that we focus on for the thesis. We use Linux based HTB where each token bucket is indicated as a scheduler where the policies are defined, and depending on the rules, packets are forwarded onto the upper bucket/scheduler as shown in Fig 3.3. Each token bucket in the hierarchy defined a network slice. Token buckets are created only if the traffic of a related set of rules exists. For defining rules for the QoS purpose, source and destination IP addresses, port numbers, and protocol (Transmission Control Protocol (TCP)/User Datagram Protocol (UDP)) are considered for the parameters.

2.1.4 RED

RED [12] is a method used in a QoS in computer networks. It helps prevent network congestion and ensures fair allocation of network resources. RED works by monitoring the length of a queue that holds incoming packets. When the queue reaches a certain threshold, random packets are dropped to signal congestion to the sender. By proactively dropping packets before severe congestion occurs, RED aims to maintain network performance and avoid complete buffer overflow. The dropping probability is determined based on the queue length and configured thresholds, allowing for a fair and efficient congestion control mechanism.

2.2 Different QoS Approaches

Network devices play a critical role in implementing QoS mechanisms to ensure optimal performance and user satisfaction. However, the capabilities and limitations of network devices vary, leading to different approaches for implementing QoS. This section explores two key approaches: utilizing network devices and employing server-based solutions.

2.2.1 Network Devices

1. **Pros:** Network devices, such as routers and switches, offer fast data path processing, enabling efficient packet forwarding and reduced latency. They are designed with specialized hardware to handle high-speed data transmission, ensuring quick delivery of packets.

2. **Cons:** Network devices often have limited computation resources. While they excel at data forwarding, complex QoS computations and control may strain their processing capabilities. Additionally, network devices are typically vendor-specific, with proprietary architectures and programming interfaces.

Vendor Specific Network Devices

Vendor-specific network devices implement QoS mechanisms to prioritize and control network traffic flow based on specific criteria. Traffic classification, priority queuing, traffic marking, traffic shaping, bandwidth reservation, and congestion avoidance are some techniques implemented by the devices which have their specific method and rules to apply with the constraint of commands, while the exact implementation may vary across different vendors. Due to the hardware-based nature, they have limitations compared with the server based-approach such as scalability, granularity, less flexibility and customization, lack of rapid adaption and upgrades, and so on. Therefore, it is recommended to refer to the vendor's documentation and configuration guides for detailed information on how a particular vendor's device implements QoS.

Programming Protocol-independent Packet Processors (P4) [13]

P4 is a high-level domain-specific language for programming the switch for packet processing. It is independent of any vendor-specific hardware meaning suitable for expressing the behavior of various switch types (e.g. fixed-function Application

Specific Integrated Circuit (ASICs), software switches, Field Programmable Gate Array (FPGAs). The language abstracts packet parsing and processing. P4 code is logically organized into data declaration, parser logic, and match plus action tables and control flow sections [13]. The policies of the QoS for traffic in the P4 switch uses the P4 code that schedules and transfers the packets according to the network resources allocated for the packets in the code[14] using the algorithm for communication mechanism in the hierarchical control plane.

2.2.2 Server-based Approach and Data Plane Acceleration Survey

A brief introduction of DPDK with a comparison with other similar technology.

1. **DPDK (Data Plane Development Kit) [2]:** DPDK is a widely used solution for fast packet processing on servers. It provides libraries and tools that leverage modern server hardware capabilities, such as multi-core processors and hardware offloading, to achieve high-performance packet processing. DPDK uses its own driver to have direct access to network interfaces, bypassing the kernel network stack and reducing processing overhead. This approach offers greater flexibility and control over QoS mechanisms, making it suitable for advanced and customized QoS requirements.

2. **Netmap [15]** Netmap is a framework for high-speed packet input and output implemented with Virtual Local Ethernet (VALE) [16], a switched ethernet for virtual machines, the software switches a single kernel module available for various operating systems such as Linux, FreeBSD. VALE is a very fast Virtual Local Ethernet using the Netmap API feature of Netmap module that deploys multiple virtual switches for interconnecting Netmap clients, including traffic sources and destinations, packet forwarded, userspace firewalls, and so on[17].

Netmap is designed as a general-purpose framework for fast packet I/O. It provides a simple Application Programming Interface (API) for user applications to access network interfaces directly, bypassing the traditional networking stack. Netmap focuses on efficient packet capture and injection, making it suitable for network monitoring, traffic analysis, and virtualization use cases. DPDK, on the other hand, is specifically developed for building high-performance data plane applications, such as NFV and SDN. It offers a comprehensive set of libraries, drivers, and APIs optimized for packet processing, enabling applications to achieve wire-speed packet forwarding and processing on commodity hardware. Netmap operates at a lower level of abstraction, providing direct access to network interfaces. It exposes network devices to user space, allowing applications to directly read from and write to the network device's buffers.DPDK operates at a higher level of abstraction compared to Netmap. It provides a higher-level API and a rich set of libraries for packet processing, including memory management, packet classification, and flow control.

Netmap is designed to be a lightweight and straightforward framework. Its API is relatively simple, providing direct access to network interfaces and buffers. While DPDK, compared to Netmap, offers a more comprehensive and feature-rich framework for building high-performance packet processing applications which introduces more complexity.

Overall, DPDK's comprehensive feature set, broad community support, hardware compatibility, and track record of high performance make it an approach for implementing QoS in server-based solutions. However, the choice of approach should ultimately consider the specific requirements and constraints of the network infrastructure, ensuring the optimal balance between performance, flexibility, and ease of deployment.

2.3 Intel's DPDK

This section explains the essential part of DPDK use in the thesis.

Intel's DPDK was originally designed to support on Intel chips and hardware which is developed over the years to support any hardware commodity. DPDK contains built-in libraries to boost packet processing in data plane applications which can be implemented in various CPU architectures. This is an open-source software framework with a wide variety of development contributors. DPDK has a set of libraries that can be implemented to develop data plane applications. DPDK makes communication between Network Interface Card (NIC) and application in user space possible without kernel involvement resulting in faster processing of networking applications. At the same time, it comes with several dependencies of the DUT from hardware processing power, NIC, storage, and the design of application using DPDK libraries. These constraints impose complexity on the design and implementation of packet processing by the DUT. Thus, a study for the feasibility of DPDK is necessary and the thesis performs the evaluation of QoS application provided by DPDK framework.

Generally, Linux kernel provides the functionalities for network-related tasks demanded by user-space applications where a packet destined for the application is copied from Receiver (RX) NIC port to kernel space to user space, executes the networking instructions and copied to kernel space for transmitting to Transmitter (TX) NIC port. For multiple applications running at a time, the kernel attempts to balance resources in the running processes. However; packet processing applications require highly tuned and specific network resource requirements thus kernel functionality does not support the requirements and features like scheduling and associated context switches introduce additional overhead.DPDK access the Ethernet controller directly skipping kernel. Meaning DPDK uses its own driver to handle NIC meaning takes full control of NIC from the hosting operating system, such as Linux, and FreeBSD, and copies the incoming packets directly from NIC to userspace bypassing network stack and kernel space. The available libraries in DPDK for driver function and forwarding mechanism bypass kernel space hence eradicating Kernel User Overhead[18]. It may seem that we can develop the entire application in kernel space where the application has complete access to hardware, but with this approach, the development becomes more difficult. In addition to that, we can depend on more tools in user space, and more techniques are known in user space than in kernel space. For the background, we summarize the aspects of the given set of libraries in DPDK which is used throughout the thesis. However, optimization is required for meeting the requirements of heterogeneous applications even with the supplied libraries.

2.3.1 Poll Mode Library

One of the most essential features of DPDK is the poll mode library that uses Poll Mode Driver (PMD). The DPDK's driver in User Space is PMD which periodically checks the input/output interfaces. Since PMD does not need a System call mechanism, it eradicates the Interrupt Context Switch Overhead [19]. Thus, the user application directly polls the received buffers bypassing the kernel. The API provided handles groups of packets for transmitting and receiving called bulks. This technique reduces overhead as well. In this technique, the Ethernet controller uses Direct Memory Access (DMA) for transferring packets from and to defined memory locations. Packets arrivals and transmissions are signaled to the user space application when head and tail pointers of TX and RX rings are updated.

2.3.2 Message Buffers and Packet

Programs generally depend on the socket API for accessing the network for which the Linux kernel provides a set of features for the ease of programmers. A pointer to a mbufs is implemented for the representation of a packet in DPDK whereas the buffer holds the entire packet. The packets are transferred from one core to another simply by conveying the pointer instead of copying the complete packet hence the packet progress through a pipeline without a single copy. With this approach DPDK outweighs the standard Linux kernel. The mbufs also retain metadata such as packet length, memory pool of the mbufs, and so on that provides headroom [20]. The headroom is a memory allocated before the start of the packet data which is used during packet encapsulation without requiring to move the entire packet including the payload in memory for facilitating extra space requirement.

2.3.3 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) is responsible for obtaining access to low-level resources, for example, hardware and memory resources, and provide a generic interface that hides the environment specifics from the applications and libraries. It initializes routine to decide how to allocate these resources which are memory space, devices, timers, console, and so on. It provides mechanisms for assigning execution units to specific cores and creating execution instances, facilitates reservation of various memory zones, interrupts handling, alarm functions, and so on. Hence, programmers can assign each thread to a specific core for applications to run and can achieve highly tuned workloads [21].

2.3.4 DPDK QoS Scheduling App

As shown in Fig 2.2, the RX thread receives packets from the RX port, classifies them, and sends them to the ring queue, where traffic is distinguished into a slice, and schedule to send packets to TX port. We can have multiple cores for the traffic management and scheduling for QoS and transmission to TX port or a single core. In multicores case, as shown in Fig 2.2, separate cores perform scheduling and transmission of a packet to NIC TX denoted by the lower picture. On the opposite, single core perform scheduling and the transmission of a packet to NIC TX denoted by the upper picture as shown in Fig 2.2. As shown in Fig 2.1 Outer Virtual Local area Network (VLAN) number assigns one of eight subport (one token bucket per subport) that provides an upper limit per Traffic Class (TC). At the same time, the inner VLAN number assigns one of the pipes defined by the double-tagged Etherframe as illustrated in Fig 2.3. The QoS application parse the received Etherframe for extracting outer VLAN id, inner VLAN id, and destination address for mapping to a queue of a specific pipe. Fig 2.3 depicts three different kinds of Etherframe, regular Etherframe, single, and double VLAN tagged Etherframe and the specific byte number to look for outer VLAN, inner VLAN, and destination Internet Protocol (IP) address of received Etherframe. A grinder (a short list of active pipes currently under processing that contains temporary data during pipe processing) with 1 bit is associated with each pipe. A grinder indicates a filled queue in a pipe that reduces overhead for searching every 4096 pipes of each support as shown in Fig:2.4 [22] for a queue with a packet. In this way DUT should not waste the CPU cycle for searching pipes with a packet as only pipes with packets are processed for the scheduling process and empty pipes are omitted.

Each pipe has 13 TC. Each support, pipe, and the respective TCs are assigned a capacity during the runtime of the QoS application of DPDK. All high priority TC (TC0-TC11) of a pipe has one queue while the lowest priority TC which is TC12 has 4 queues served in a weighted round-robin scheduling [23]. Incoming traffic is allocated in a pipe according to the last octet of the destination address in the

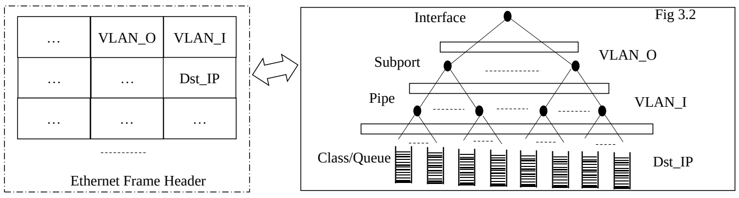

Figure 2.1: Assignment of an Ethernet frame in a mbufs using HTB method by QoS application of DPDK

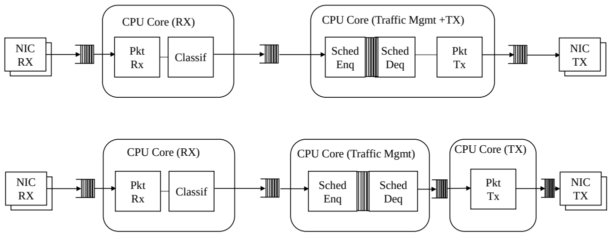

Figure 2.2: DPDK QoS scheduler application architecture [23]

respective 16 queues with modulo 16 such that an increasing number of traffic flows are assigned at one of the 16 traffic queues of a pipe. The packets at lower priority TCs of a pipe are able to reuse capacity currently unused by higher priority. The traffic class priority is in decreasing order with the TC0 being the highest priority. The packets at one pipe can not use the unused capacity of another, resulting in strict network resources between any pipes.

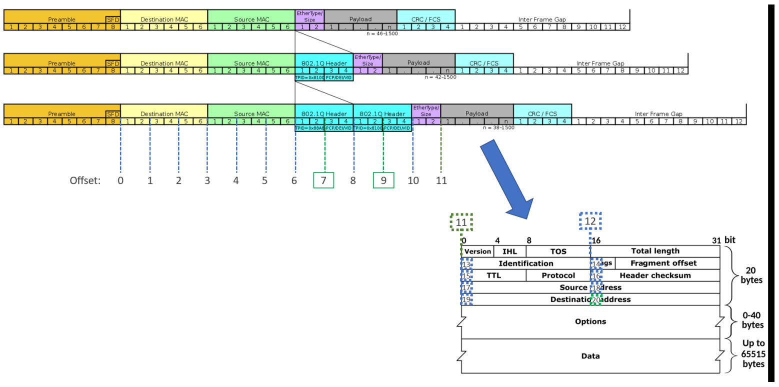

Figure 2.3: Double-tagged Etherframe for DPDK packet assigning

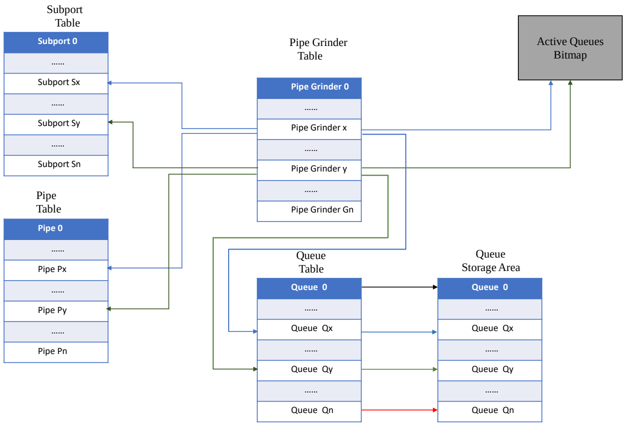

Figure 2.4: Internal data structures per port

2.4 Network Performance Metrics

Specific parameters are considered for the evaluation of system performance also referred to as metrics. These metrics assist in the comparison between the implementation of the system with similar functionality. They can be useful for the optimization of the system using the evaluated metrics results.

2.4.1 Throughput and Capacity

Throughput is one of the key metrics essential for analyzing network device performance. Throughput measures the number of packets that arrive at their destination successfully. The data rate is one of the most important things required to monitor network performance, and throughput as well as capacity is used for measuring it. How fast packets are delivered from source to recipient or vice versa determines the amount of information or data that can be sent in a given timeframe. Whereas, network capacity is defined as the maximum transfer throughput capacity of a network. In this thesis, we calculate throughput on a 10-gigabit Ethernet (GbE) link and can be computed for any links with a different capacity[24]. Since we test heterogeneous applications in this thesis, meaning different applications have distinct packet sizes, with a double-tagged frame overhead of 24 bytes. We measure throughput in bytes per second.

The throughput, denoted by T, can be measured in bytes per second and is limited by the capacity of the link. It can be expressed as:

$$T = \text{Capacity} \times \text{Efficiency} \tag{2.1}$$

where the capacity is the maximum achievable throughput of the link and the efficiency represents the utilization of the link.

2.4.2 Packet Delay

Packet delay is defined as the time difference between a packet received by the destination host and the packet sent from the source host[25]. It is a fundamental metric, also refers as latency, for performance analysis of a network device which can be measured by applying timestamps on a packet. Packet delay is introduced by transmission delay (time for a packet to travel in link), queueing delay (time a packet is stored in input and output queue before processing and transmitting), processing delay (time taken by sending, network, and destination node to process) [26] as shown in Fig 2.5. This has indicated one of the crucial metrics for 5G network [27] with less than a millisecond for mission-critical applications such as remote surgery. The source host put A timestamp to a packet and upon reception by the destination

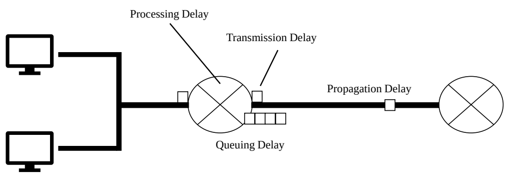

Figure 2.5: Packet switched delay

host, subtracts its clock from the timestamp. Clock synchronization across the hosts is essential for the accurate measurement of packet delay.

The one-way packet delay can be calculated using the equation:

$$\text{Packet delay} = \text{Time of packet received} - \text{Time of packet sent} \tag{2.2}$$

In the one-way packet delay, the time that elapses by a packet from a source node to the destination node is calculated. But round trip delay refers to the time taken for a packet of data to travel from a source to a destination and back again. It represents the total time elapsed for a round trip between two network devices.

2.4.3 Sensitivity and Performance Isolation

Sensitivity is another metric essential for the performance analysis of a system implementing network isolation. Changing the data distribution function and the impact of changes in the parameters of the system can be a sensitivity analysis. In this thesis, variation of throughput of an application by the introduction of another application in the same or different pipe is considered as sensitivity and performance isolation metric of DUT. In perfect network isolation, there should not be a variation in the throughput of an application in one network from ideal to heavy traffic in another network. Even, the throughput of an application in one network is expected to have flat constant throughput [28].

2.5 Network Automation

Network automation is the process of using software to automate network and security provisioning and management for continuously maximizing network efficiency and functionality [29]. Network automation increases the efficiency and speed of application deployment through complete application lifecycles and across the data center and cloud environment. This approach is less error-prone than manual provisioning and management, hence network engineers can focus on the optimization of network infrastructure than the configuration. It implements software with API [30] for the most efficient way to map, configure, provision, and manage a network. Hence, replacing manual, vendor's specific command-line instructions to configure each networking device which can be invoked directly or go through a programming language, for example, Java, Python, or Go. There are several tools for network automation sometimes refers as DevOps [31] tools such as Docker, ansible, and Kubernetes.

2.5.1 Docker

Docker is a set of platform-as-a-service as shown in Fig 2.6 products in cloud architecture enabling OS-level virtualization to deliver software in packages called containers. The cloud refers to servers accessed over the Internet which provide on-demand availability of computer resources from data storage to computing power without direct management by the user [26]. It provides all the packages and dependencies for an application in a virtual container that runs on top of any operating system (Windows, Linux, macOS) as shown in Fig 2.7. This can be run at any location extending from on-premises [32] to public or private cloud [33]. The most important point about Docker is that it is a containerization platform and runtime.

2.5.2 Ansible

Ansible is an open-source software tool for simplifying network automation [34]. It provides simple but powerful automation for cross-platform computers used to automate infrastructure, applications, networks, containers, security, and the cloud [35]. It is easy to understand and does not use any other third-party tool. As illustrated in Fig 2.8, it applies a Yet another markup language (YAML) [36] file called ansible-playbook in Ansible for end-to-end automation of components which is written in a simple human-readable language. As all configuration files are mostly written in YAML, they can be easily understood for machine-level automation or code for debugging.

In this thesis, we are implementing Ansible with a Docker container for automating heterogeneous applications requesting resources from servers. Since Ansible

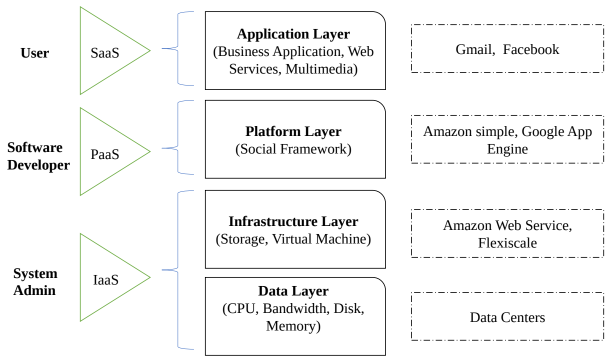
User
SaaS
Application Layer
(Business Application, Web Services, Multimedia)
Gmail, Facebook
Software Developer
PaaS
Platform Layer
(Social Framework)
Amazon simple, Google App Engine
System Admin
IaaS
Infrastructure Layer
(Storage, Virtual Machine)
Amazon Web Service, Flexiscale
Data Layer
(CPU, Bandwidth, Disk, Memory)
Data Centers

Figure 2.6: Cloud computing architecture

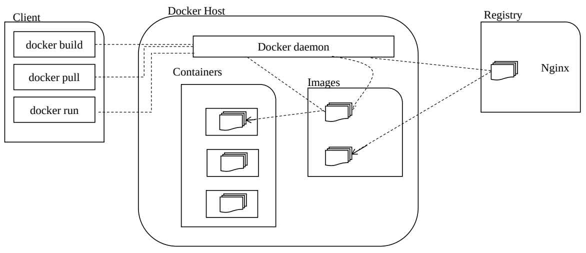

Figure 2.7: Docker architecture

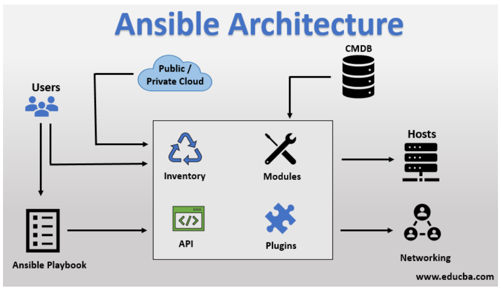

Figure 2.8: Ansible architecture [37]

is agentless and does not use third-party vendors or agents' software, it is easier to deploy than Kubernetes[38]. Unlike Kubernetes, Ansible is easy to understand deployment. As with Kubernetes capable of handling complex structures and continuous monitoring of Pods, our work is simple and does not need such computational overhead. Making Ansible an easy choice.

Chapter 3

Methodology

We are interested in providing limited capacity, delay, jitter, and throughput for heterogeneous applications in a shared infrastructure adhering to service-level agreements for the users. We choose DPDK framework for providing QoS to various applications and isolation of the traffic. Initially, we need to create double-tagged VLANs, first create outer VLAN to an interface connected towards DUT that forwards the packets to a destination node, and then create inner VLAN to the logical outer VLAN interface, in both servers and clients machines so that a packet is encapsulated in a double-tagged Ethernet Frame as shown in Fig 3.1. Each container is attached to the inner VLAN network that must be the same in both servers and associated clients requesting the service as shown in Fig 3.1. We need to add an entry in a routing table of machines hosting servers and clients so that the traffic is forwarded towardsDUT interface that provides QoS and forwards to the destination to the hosted heterogeneous services or clients. As for the deployments, we use Ansible which deploys the servers and clients in the associated VLAN networks. For the performance analysis of DPDK for QoS for heterogeneous applications requesting access to their respective servers, this thesis has completed experiments in a physical testbed. As shown in Fig 3.1, we have implemented three server hardware connecting through 10 GbE in a series connection. One end server hosts various servers with different network resource requirements in docker containers, and the other end host clients in docker containers that request service to the respective server. The middle server running DPDK QoS application is responsible for forwarding traffic between respective applications requesting service to the server after applying the assigned quality shaping parameters to the service. Afterward, we deploy different numbers of services with different network resource allocations and evaluate the performance parameters, delay, throughput, adaptive bit rate, and sensitivity.

3.1 Thesis Design

The thesis work is reproducible with extensive parameters with the help of network automation cleanly and easily. We are implementing the following applications:

NGINX and Wrk2 NGINX is open-source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. In this thesis, we use it for a default nginx web page hosting and evaluate the delay introduced by DUT by accessing it through wrk2. wrk2 is a modern Hypertext Transfer Protocol (HTTP) benchmarking tool capable of generating significant load when run on a single multi-core CPU which provides a details latency report that we use to plot packet percentile against delays [39].

Dashjs Dashjs is an open-source JavaScript library used for implementing Dynamic Adaptive Streaming over HTTP (Dynamic Adaptive Streaming over HTTP (DASH)) in web browsers. It simplifies the integration of DASH into web applications,

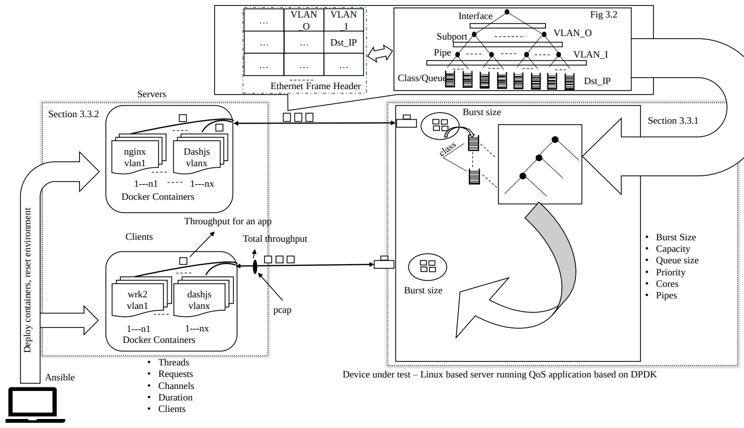

Figure 3.1: Experimental design with Ansible deployed heterogeneous applications and clients where DUT provides QoS with application specifics and DUT specific control knobs

providing adaptive bitrate streaming, support for various multimedia formats, and advanced features like trick play and time shifting [40]. Dash.js is a valuable tool for enhancing video streaming capabilities in web applications. The quality level for clients in our dashjs application is measured in the following way which is stored at the server in different representations and served to clients.

1. **0** = 480 pixels with 214876 bits per second (bps)

2. **1** = 720 pixels 406802 with bps

3. **2** = 1080 pixels with 797965 bps

4. **3** = 1080 pixels with 1610009 bps

5. **4** = 1080 pixels with 3393083 bps

Iperf3 IPerf3 is an open-source network testing tool used to measure network performance and diagnose issues. It measures parameters such as throughput, and packet loss and supports TCP and UDP traffic streams. It requires a client and server setup and is used for capacity planning, troubleshooting, and optimizing network performance.

The system consists of heterogeneous servers, nginx, iperf3, and dahsjs in our cases in separate VLANs. All the servers are containerized in docker containers using Ansible for deployment. Likewise, the number of clients are deployed in docker containers using Ansible by allocating in their respective VLANs as in the servers. For instance, server nginx is in VLAN1 and the clients accessing the server launch the wrk2 application in VLAN1 by setting the parameters of wrk2, which are the number of threads, requests, channels, and duration for measuring round trip delay introduced by DUT that is Linux-based server running QoS application as shown in Fig 3.1.

The DUT is a Linux-based server running QoS application which is configured with EAL parameters, runtime, and supplied configuration file. Each incoming double-tagged Etherframe is mapped to the respective pipe's queue by parsing its outer VLAN, inner VLAN, and destination address. Control knobs at DUT are the capacity of the link, queue size, number of cores, number of pipes, priority of traffic, ring size, mbufs size, queue threshold parameters, and so on explained in 3.2. The DPDK has batch processing meaning the assigned burst size at the runtime of QoS application determines how frequently the system hops between the prefetch stage and scheduling process which is explained in 3.2.1. Defining separate cores for different flows allows parallel processing of traffic flows that boost the faster processing of the system and reduce latency introduced by DUT.

Each application has its own control knobs. For nginx, clients can assign numbers of threads, constant rate traffic, number of open channels, and duration of the overall requests which are explained in 3.3.2. For dashjs, the control knobs would be a number of clients, videos, and available resolutions.

For various configurations of control knobs at application levels and DUT, we capture a pcap file at the client-facing interface as shown in Fig 3.1 using tcpdump. We analyze the throughput, delay of individual applications for a client, and sensitivity introduced by DUT for deducing how the DUT behave in various environments of the setup by tuning control knobs.

3.2 DPDK QoS Application

The QoS framework provided by DPDK implements HTB based queuing discipline for enforcing QoS policy for aggregate flows. Traffic class nodes are configured in a hierarchical structure with three levels (root, sub-root of aggregated traffic, and a leaf). In a similar manner, each physical interface/port (root) has multiple subport (sub-root) defined with a token bucket which consists of 4096 pipes assigned a token bucket to each in a hierarchical manner. Each pipe has 16 nodes representing traffic classes with a queue and priority assigned for enforcing QoS as shown in Fig 3.2.

The user space memory of the computer running DPDK is divided into different memory pools with a token bucket with a set of rules. The packets are then accounted

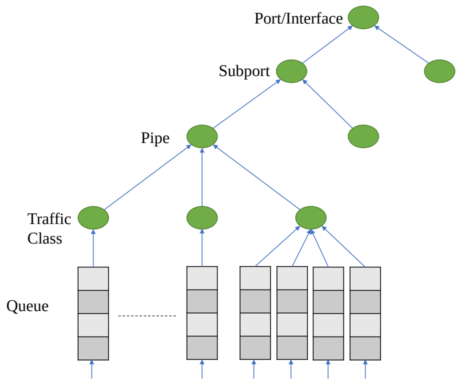

Figure 3.2: Scheduling hierarchy per Port [23]

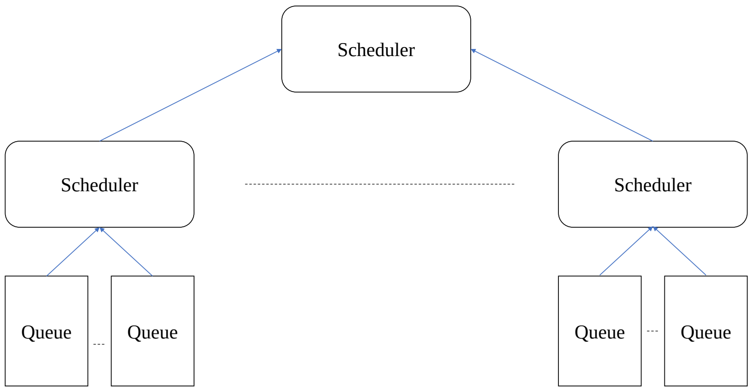

Figure 3.3: Hierarchical scheduling

for against the respective token bucket. In a token bucket, further policies are applied to provide priority to the packets so that the packet with the highest priority within a bucket is processed and forwarded first for QoS purposes. As shown in Fig 3.3, each token bucket is indicated as Scheduler where the policies are defined, and depending on the rules, packets are forwarded onto the upper bucket/scheduler. Each token bucket in the hierarchy defined a network slice. Token buckets are created only if the traffic of a related set of rules exists. For defining rules for the QoS purpose, outer VLAN id, inner VLAN id, and destination IP address of received Etherframe are considered for the parameters.

For the purpose of providing QoS using QoS application from DPDK, compilation and running the dpdkqos_sched application (available in the DPDK software) with DPDK defined configuration file along with EAL parameters and application parameters are executed as shown below. Each section is described below [1].

./<build_dir>/examples/dpdk-qos_sched [EAL options] – <APP PARAMS>

Here – seperate EAL parameters with application parameters

3.2.1 Pipeline

There are enqueue pipelines and dequeue pipelines in QoS application of DPDK.

1. **Enqueue Pipeline** : The details sequence of steps per packet is available in [43]:

   There is a need for a prefetch mechanism to prevent data cache misses due to strong dependency between the three steps in which the required data structure is prefetch in advance which comes with an execution latency. During the execution latency, the processor should not attempt to access the data structure currently under prefetch. This duration should be utilized for the execution of other work, which is to execute various stages of enqueue sequence of operation on other input packets resulting in a pipelined implementation for the enqueue operation as shown in Fig 3.4.

2. **Dequeue Pipeline**: The details sequence of steps to schedule the next packet from the current pipe is available in [43]:

   To avoid cache misses, the data structures (a pipe, queue, queue array, mbufs are prefetched in advance of being accessed. The strategy for hiding the latency of the prefetch operations is to switch from the current pipe (in grinder A) to another pipe (in grinder B) immediately after a prefetch is issued for the current pipe. This gives enough time for the prefetch operation to complete before the execution switches back to this pipe (in grinder A) [43].

   Because of the prefetch mechanism, the performance of DPDK QoS application suffers the performance in lower packet processing, as the processor remains idle during the prefetch period. In other words, the port scheduler performance is optimized for a large number of queues or packets. If the number of queues or packets is small, then the performance of the port scheduler for the same

[1]We implemented the dpdk-stable-2.11.1 version from [41] for our experiment. It is required to set a minimum of 4 gigabytes of huge pages for each of the used sockets [42], our hardware system has 2 sockets, 128 cores, 10Gigabit per second (GBPS) NIC,256 Gigabyte random-access memory (RAM)

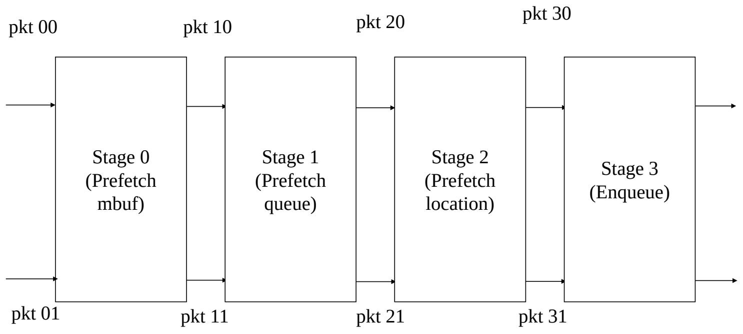

Figure 3.4: Prefetch pipeline for the hierarchical scheduler enqueue operation

level of active traffic is expected to be worse than the performance of a small set of message-passing queues [43].

3.3 Control Knobs and Paramters

There are two sections of control knobs. First is the parameters of QoS application of DPDK and the heterogeneous services which are being served by the DPDK.

3.3.1 QoS Application Parameters

There are considered optional application parameters which include interactive mode, main and worker cores index, ring size, buffer size, burst size, mbufs size, configuration file, and so on. We can choose the default value or assign it at runtime [2].

We need to supply the configuration file called profile configuration file as an application parameter of QoS application of DPDK. It is the file that contains the resource allocation for each service using the DPDK QoS application and isolation between each service residing in a different pipe (each pipe represents a VLAN out of 4096). Listing A.1 is a sample configuration file that needs to be modified according to the requirements for serving QoS purpose to each application in the system. The traffic shaping for subport and pipe is implemented using a token bucket per subport/per pipe. Each token bucket is implemented using one counter that keeps track of the number of available credits. As the greatest common divisor for all packet lengths is one byte, the unit of credit is selected as one byte. The number of credits required for the transmission of a packet of n bytes is equal to (n+h), where

[2]For the complete details of the parameters please visit [42]

h is equal to the number of framing overhead bytes per packet. As a result of packet scheduling, the necessary number of credits is removed from the bucket. The bucket is set to a predefined value, e.g. zero or half of the bucket size. Credits are added to the bucket on top of existing ones, either periodically or on-demand, based on the bucket rate. Credits cannot exceed the upper limit defined by the bucket size, so any credits to be added to the bucket while the bucket is full are dropped. The packet can only be sent if enough credits are in the bucket to send the full packet (packet bytes and framing overhead for the packet) [22] [3].

The essential function of DPDK QoS application to reduce searching overhead during the scheduling stage for a pipe with packets is that active queues bitmap is used. The bitmap maintains one status bit per queue: queue not active (the queue is empty) or queue active (the queue is not empty). The queue bit is set by the scheduler to enqueue and cleared by the scheduler to dequeue when the queue becomes empty. Bitmap scan operation returns the next non-empty pipe and its status (16-bit mask of the active queue in the pipe). The grinder contains temporary data during pipe processing. Once the current pipe exhausts packets or credits, it is replaced with another active pipe from the bitmap [22].

These are the mandatory application parameters and must be defined at the beginning of the running program. They are listed as:

1. –pfc "RX PORT, TX PORT, RX LCORE, WT LCORE, TX CORE": Packet flow configuration. Multiple pfc entities can be configured in the command line, having 4 or 5 items (if TX core is defined or not) [42].

The parameters we can tune from the QoS framework provided by DPDK are:

1. **Queue Size**: DPDK has batch processing.The runtime queue size defines the execution time delay and the number of packets processed at a time. We can define queue size in the power of 2 (64 being the default) that stores the arriving packets in mbufs during enqueue and transmit to the TX interface for the traffic after applying QoS policy defined in runtime.

   The DPDK processed a fixed number of requests at a time that is defined by the queue size because of the prefetch stage in enqueue process. The core shifts from the prefetch stage to a hierarchical scheduling process and dequeue process for transmitting packets to the server interface after the queue is filled with threshold and the working core is available for the scheduling process.

   For a small queue size, DUT can process a small number of packets at a time which reduces throughput and delay for packet processing. For instance, a

[3]The complete description of each component of configuration file parameters is available in [43]

packet waits a long time in a larger queue compared to a small queue causing a higher delay for the larger queue. On the other hand, a larger queue can hold large amounts of packets at the prefetch stage and schedule the packets, making the throughput of the DUT higher.

Because the constant request packets are transmitted from the client side per second, and the remaining retransmission of requests from the previous instance of request, the delay is significant at low traffic. At the same time, as the higher queue size wait for the queue to be filled beyond the threshold level, a significantly larger queue size results in a longer prefetch delay. The tradeoff between prefetch delay and processing delay has a noticeable impact on the delay introduced by the DPDK.

2. **Burst Size**: This values defined the batch processing of the QoS framework of DPDK. The default value is 64 meaning the QoS application processes the packets once there are 64 packets in a queue. Thus, large delays are introduced at low traffic because of the waiting time in the prefetch stage as well as the scheduling stages. The minimum value for the application to run is 8 meaning the batch processing is performed for every 8 packets from reading packets from NIC RX at the prefetch stage till writing to NIC TX. In more detail, the main core reads 8 packets from the NIC RX, writes to output software rings, and the worker core reads from input software rings followed by QoS dequeue size of 8. Finally, write to NIC TX by worker core at 8 packets of burst size.

3. **Capacity**: We can change the capacity of a pipe that provides the upper limit of throughput for traffic at the pipe. The lower capacity of a pipe results in lower throughput while a larger capacity provides larger throughput.

4. **Pipe**: Assigning traffics in different pipe leads to the isolation of traffic that causes checks active bit maps of each pipe of 1 bit (1 for traffic in the pipe and goes for processing in scheduling). This approach facilitates the assignment of the capacity for the different applications at different VLANs. Each subport can have a maximum of 4096 pipes where pipe 0 must be assigned with minimum Tocken Bucket (TB) rate of 250 bytes per second and TCs with 1 byte per second without which the QoS application fails to run which is an issue with the development because pipe 0 does not hold any packets as pipe number represents inner VLAN id that starts from 1. We can define individual TB rates for each queue in a pipe. The overall allocation of TB rate at pipes for a subport should not exceed the TB rate of the subport otherwise there will be oversubscription. Oversubscription for subport traffic class X is a configuration-time event that occurs when more capacity is allocated for traffic class X at the level of subport member pipes than allocated for the same traffic class at the parent subport level. The existence of the oversubscription for a specific subport and traffic class is solely the result of pipe and subport-level

configuration as opposed to being created due to the dynamic evolution of the traffic load at run-time (as congestion is) [44].

and the batch processing in multiple pipes has better scheduling than the same pipe.

3.3.2 Heterogeneous Services Parameters

We use nginx, dashjs, and iperf3 applications for the measurement of the experiments. Especially nginx is deployed for evaluating delays introduced by the DUT.

The parameters we can tune for the nginx application which is been requested by the clients running wrk2, which is used for achieving the least delay, (application-specific parameters that change based on the choice of application) are:

1. **Channels**: This indicates the number of HTTP connections that define how many packets can be sent from the application level to the interface at an instance of time. Larger values with fewer requests/response rates cause a larger sleep time after reset of HTTP connections from the wrk2 application.

2. **Request**: The request parameter defines the constant number of requests sent from wrk2 at every second. If DPDK QoS can not process accumulated requests, the wrk2 resends the requests over time causing a larger overall delay.

3. **Threads**: The number of threads defined by the parallel processing of generated request/response rates from the application.

We implement iperf3 as the cross-application for our thesis. The parameters we can tune for the iperf3 application, which is used as cross-application for the least delay requiring application (nginx in our case) are:

1. **Socket Buffer Size**: The default socket buffer size in Linux is 16 Kilo byte (KB) [45]. Increasing the value at the server can increase the number of iperf3 packets transmitted through DUT to the iperf3 client. We changed to 32 Mega Byte (MB).

2. **Maximum transmission unit (MTU) Size**: This parameter specifies the iperf3 packet size. We can generate a large number of packets for a given capacity of a link if we use the least size of packets in iperf3. The iperf3 is used as a cross-application for an application (nginx for our thesis) with low traffic requiring the least delay. A minimum packet size supported by iperf3 is 88 bytes which is used in the experiments.

3.4 Observed Metrics

In this thesis, we perform experiments with a variety of parameters from application levels and DUT and measure throughput, delay, and performance isolation agnostic applications as shown in Fig 3.1 with available network resources. For this, the pcap file is captured at the client interface connecting to DUT, filtered the appropriate packets, and measure our metrics of interest.

1. **Delay**: We use the nginx application with a capture response at the client interface. At the client side, wrk2 is used with various requests, threads, and channel numbers which output the delay measured from the first byte of the request to the complete byte of the response. The server introduced at most 1 millisecond of processing delay to generate a response to each request.

2. **Throughput**: By assigning a specific level of capacity to each pipe and running multiple numbers of clients accessing service from the server, we observe the throughput which is in terms of expected throughput to observed throughput. We observe the pcap file capture at the client interface and filter specific clients' packets from the total input packets at the interface.

3. **Performance Isolation**: We capture pcap file at the physical client interface and parse individual clients packets. By observing the variation of throughputs of individual clients and the total throughput, we measure how DUT reacts in terms of the throughputs for the initialization of an application, introduction of another application, parallel processing of different clients or applications, and destruction of another application. For the purpose, of two different nginx at distinct pipes with different capacities, we observe how the DUT responds to the two traffics at two different pipes. The setup enforces 20 seconds of each nginx running periodically. We set up the experiment in such a way that both nginx instances are active for 10 seconds of each round, and the other nginx instance is deactivated for the remaining 10 seconds of each instance.

4. **Heterogeneous Applications**: By running two different applications in the same pipe and different pipe and plotting the throughput for the applications for both scenarios from capture pcap file at clients' interface as shown in Fig 3.3, we observe the behavior of DUT for providing QoS for heterogeneous applications. We use the nginx application which requires low delay and a dashjs application for adaptive bitrate VoD applications that require higher throughput for optimum users' QoE. First, we measure the nginx's request rate for 100 Megabit per second (Mbps) capacity link with the least delay performance by DUT. A different setup to observe the mean quality level of an increasing number of clients at 100 Mbps capacity link of dashjs application which is deployed as shown in Fig 3.3. Using the optimum request rate for

nginx and the number of clients for dashjs, we measure the throughput of both applications running at same network (pipe) of 200 Mbps link over time and two separate pipes of 100 Mbps capacity each.

3.5 Traffic Patterns

It is the response of the DUT for the various applications with tuned parameters of both service and QoS application. Traffic pattern helps to understand the behavior of DUT under control knobs configuration. We measure traffic patterns of low delay and low throughput under the least burst size (8) by plotting throughput against the average delay from the pcap file capture at the client's interface. Under the same setup, increasing burst size results in increasing average delay and throughput for the same amount of traffic. The delay becomes higher and throughput remains saturated after the load exceeds the capacity of the link because of the congestion control for high traffic before the packets are stored at mbufs.

In like cases, network isolation, throughput, and sensitivity have a pattern in the environment with tuned control knobs that are measured. Throughput increases for increasing link capacity.

3.6 Experimental Design

This section covers the experimental design for performance measurement of DUT.

3.6.1 Delay Measurement

For delay, there are numerous ways of tuning the delay where we use the nginx application and wrk2 as the client requesting the service. We observe the behavior of DUT by the experiments below to achieve low delay for the nginx application because, for each parameter, the DUT behave in a specific way.

1. We use various capacity links and measure the average delay for nginx application at different requests per second that measure utilization of link and the delay response.

2. We change the number of channels, fixed threads (1), and the offered traffic and measure the average delay.

3. We measure the actual loads by supplying various offered loads from using the wrk2 client.

4. We provide constant offered traffic (1000 requests) at various queue sizes of QoS framework of DPDK from 64 to 1024 and measure average delay response.

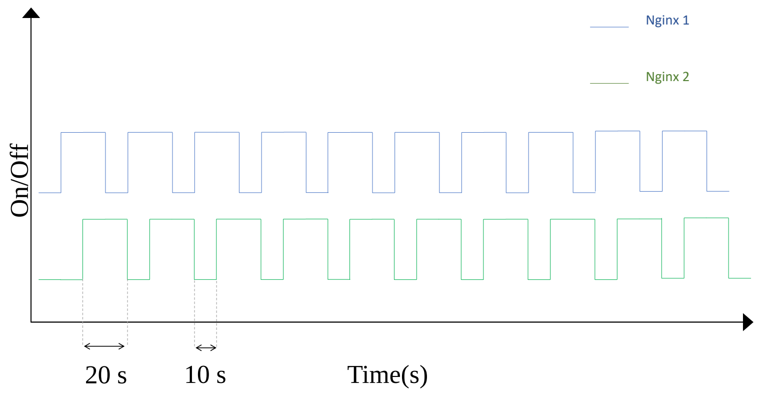

Figure 3.5: Two Nginx at pipe 2 and pipe 3

5. We supply constant link capacity (200 Mbps), offered traffic (1000 requests), channels (4000), threads (4), and an increasing number of iperf3 instances (with 32M socket buffer size and 88 bytes of the least packet size for generating a large number of possible packets as cross-application) in a separate pipe and measure the average delay.

6. We use various burst sizes to observe average delays at fixed offered traffic using the nginx application. Followed by the best burst sizes and various offered traffic and observe the average delay distribution.

7. We supply constant link capacity (200 Mbps), offered traffic (1000 requests), channels (4000), threads (4), and an increasing number of iperf3 instances (with 32M socket buffer size and 88 bytes of the least packet size for generating a large number of possible packets as cross-application) in different pipes (iperf3 instances is equally distributed among various pipes) and measure the average delay.

3.6.2 Performance Isolation

We use 2 nginx instances in separate pipes as shown in Fig 3.5 with 100 Mbps and 50 Mbps of link capacity of pipes. Here, the 2 instances run for 20 seconds periodically with 10 seconds of overlapping activation period and measure the sensitivity of DUT response with the initiation of different applications and overlapping activation period of applications.

3.6.3 Throughput and Capacity

We monitor the throughput by providing different link capacities. The pcap file is captured at the client interface for running the nginx server accessed by a client with various request rates The packet dropping occurs before the prefetch process by means of the congestion control mechanism of DPDK that helps congestion control and processing overload minimization of DUT.

3.6.4 Heterogeneous Services

As shown in Fig 3.3, first nginx run at a pipe of 100 Mbps capacity for various request rates to discover the request rate for minimum delay performance by DUT. Second, dashjs run at a pipe of 100 Mbps capacity for increasing the number of clients for finding the highest number of clients for the highest bit rate for each client from the setup. Afterward, We set up two pipes of 100 Mbps capacity each and recorded throughput for each nginx and dashjs by capturing pcap at the client interface as shown in Fig 3.3. For the second part, both applications run at the same pipe of 200 Mbps capacity, and we record the throughput for each of them.

Chapter 4

Evaluation

We have conducted various experiments by tuning the parameters of DPDK QoS application, probing applications, and cross-applications with the setup described in the methodology Section 3.1. We capture pcap file at the client interface and plot the graphs for evaluation. This section will evaluate the performance analysis of DPDK for heterogeneous applications shared environment.

4.1 Tuning Parameters

We have changed the queue size and burst size of DUT and observed the average delay measured by the nginx server which is accessed by a client running the wrk2 application. We observe from various experiments that a large amount of HTTP channels for a low request rate results in a reset of HTTP connections and longer sleeping time until the channels are established again. Likewise, there is at most 1 millisecond of processing time by the nginx server to produce a response for a request. We choose 200 Mbps of link capacity for an experiment. For the client side, we choose 4 threads, 4000 channels, and various numbers of request rates and observe the increasing throughput against average latency resulting from the wrk2 application.

4.1.1 Average Delay with Burst Size and Queue Size

As shown in Fig 4.1 by tuning parameters explained in Section 3.3.1 where we start the experiment with request rates as 100, 1000, 2000, 5000, 10000, 20000, 24000, and 30000, where 1 request rate generates a packet of 933 bytes, for all four different queue size and burst sizes represented by 4 lines in the given figure, it is observed that for a lower burst size, the average delay is drastically reduced compared with the same queue size. In our case, either the green vs red lines which represent 64 queue size with burst sizes 64 and 8 respectively, or blue and yellow lines which represent 128 queue size with burst sizes 64 and 8 respectively for DUT. On the contrary, reducing burst size results in decreased throughput of the DUT. As the throughput

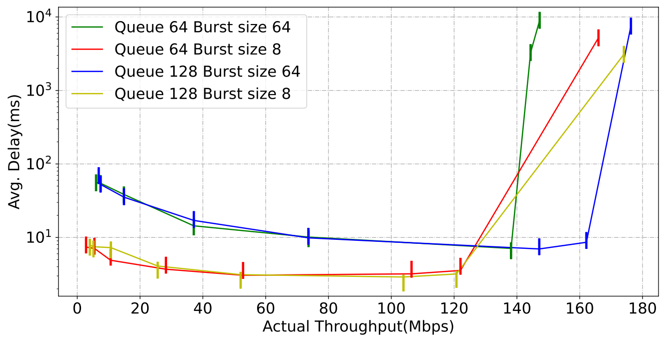

Figure 4.1: Evaluation of average delay of wrk2 application accessing nginx web server and achieved throughput for different request rates for various configured queue size and burst size of DUT at 200 Mbps capacity pipe

increases the delay increases faster for low burst size. Again, a large queue size also increases the throughput which is depicted by blue and green lines that represent 128 and 64 queue sizes with 64 burst sizes respectively. Another important aspect is that there is a high average delay for low traffic (100 request rate in our case) that reduces for an increasing number of packets, maintains a window of lower average delay, and skyrockets. The reason for the higher delay at low traffic is the hopping between the prefetch stage and the scheduling stage explained in 3.2.1 where a lower traffic rate requires longer prefetch time before the scheduling process by DUT. For larger delay on the right side of Fig 4.1 is that as the traffic rate approaches the available capacity of the link (200 Mbps in this case) the DUT drops the packets before the prefetch stage by RED implementation of the DUT. In summary, lower burst size results in lower throughput and low average delay compared with higher burst size, and a larger queue can hold more packets during the prefetch stage which results in larger throughput for the same link capacity.

4.1.2 Capacity

As illustrated in Fig 4.2 by tuning parameters explained in Section 3.3.1, setting higher link capacity results in larger throughput. We implement 2 setups where the link capacity of DUT is set at 200 Mbps and 1000 Mbps given by green and red lines respectively. We use the nginx server accessed by a client running the wrk2 application that request rates an increasing number of request rates starting from 100 request rates. We use 4 threads and 4000 channels as the wrk2 parameters explained in Section 3.3.2 with an increasing number of request rates. Capturing the pcap file at the client interface and measuring response as actual perceived throughput

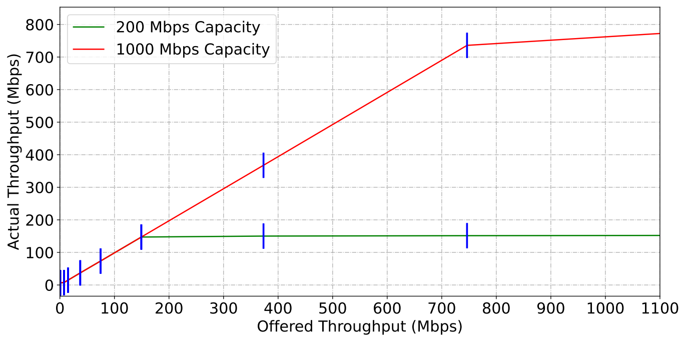

Figure 4.2: Measurement of throughput of nginx at various request rates at 200 Mbps and 1000 Mbps configured pipe capacity

for offered throughput suggests that larger link capacity provides larger throughput compared with lower capacity.

4.2 Packet Delay Using wrk2-nginx Application

This experiment involves the parameters explained in Section 3.3.1 and Section 3.3.2. We are using the detailed latency percentile information reported by the wrk2 application for accessing an nginx server [39]. The link is set up at 200 Mbps and offered nginx traffic is 1000 request rates which result in around 7.5 Mbps of offered throughput, 4000 channels, and 4 threads. The line with high denotes additional 2 iperf3 instances with windows size 32 and packet size 88 bytes, and 6 dashjs instances in distinct pipes for generating large packets as cross-application to lower delay performance for nginx application by DUT as shown in Fig 4.3. We can evaluate that the reduction of burst size as is in experiment 4.1.1 drastically reduced the delay performance of DUT indicated by the red line with low burst size 8 compared with the green line which has equal queue size 64 and burst size 64. Another method for reducing QoS delay metric is the introduction of additional packets which is given by sky blue line for an equal queue size(64 in this case) and equal burst size (8 in this case) instead of reducing burst size because reducing burst size also reduces the throughput of DUT evaluated in 4.1.1. If we combine low burst size with additional traffic, the delay is also significantly reduced indicated by the sandy-beach-color line (64 queue size and 8 burst size) compared with a low-traffic green line but only a few milliseconds of reduced delay compared with the same additional high traffic and high queue size. Similarly, increasing queue size also reduces burst size in all experiments as compared with lower queue size with other similar DUT and application-specific control knobs.

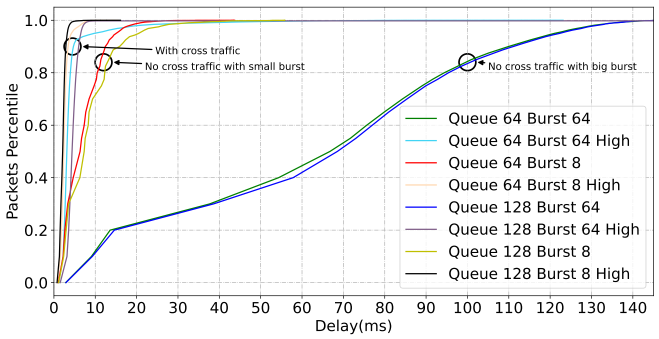

Figure 4.3: Evaluation of packets percentile for delay(ms) of nginx application offered round 7.5 Mbps throughput rate with various burst size, queue size, and cross-applications for configured 200 Mbps capacity link

4.3 Throughput and Capacity

As illustrated in Fig 4.4 by tuning parameters explained in Section 3.3.1 with a fixed capacity of the link set at 200 Mbps, we observe that larger queue size and burst size represented by blue line results in higher throughput for a given link capacity compared with low burst size and low queue size. The setup is similar to 4.1.2 for a fixed capacity link. The grey dotted line represents the ideal case where the DUT provide equal throughput for the traffic being processed. More downward a line is from the grey line, worsens the performance for throughput delivery by the DUT.

Because of the congestion control implementation, the throughput never reaches the capacity of the link and sudden. There is a sweet spot between low traffic and high traffic that is defined by the capacity of the link assigned by QoS application for the interface. Lower assigned capacity introduces a small window for the low delay because the capacity is exceeded by the introduced load. On the contrary, higher assigned capacity provides a larger window for a low delay because the link is not exhausted with the increasing amount of traffic as compared to the lower capacity. The effect arrives from the utilization of the link regardless of the link capacity. As the utilization of the link increased, the delay decreased and remains a marginal difference for a window of utilization and again increased drastically for increased utilization.

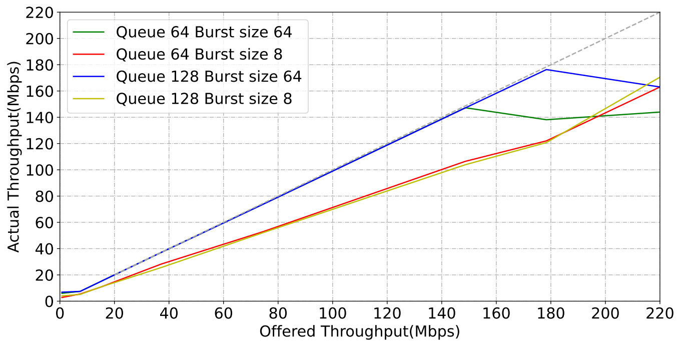

Figure 4.4: Measurement of actual throughput for different offered wrk2 request rate for configured various burst size and queue size of DUT at 200 Mbps capacity pipe

4.4 Performance Isolation under Cross/Competing Traffic

As illustrated in Fig 4.5 by tuning parameters explained in Section 3.3.1 with two capacity links set at 100 Mbps and 50 Mbps, we deploy two nginx instances in such a way that 2 instances at different pipes 100 Mbps and 50 Mbps capacity each run for 20 seconds periodically with 10 seconds of overlapping activation period and measure the sensitivity of DUT response with the initiation of different applications and overlapping activation period of applications as explained in 3.6.2. Here, each nginx runs with a 1000 request rates, 4000 channels, and 4 threads as elaborate in 3.3.2. The pcap file capture at the client's interface and plotting the graph for throughput under the experimental design suggest that DUT provide a high degree of performance isolation applications at the distinct networks (pipe in our case). As can be observed in 4.5, the rise of the second nginx application (blue line) at 50 Mbps capacity network does not impact the packet scheduling performance of DUT for an nginx application (green line) at 100 Mbps and vice versa. Instead, the overall throughput increases for two applications at different networks which reduces as the second application halt. However, packets for the application at the low-capacity link (blue line) are processed in a tiny amount even after the application is stopped.

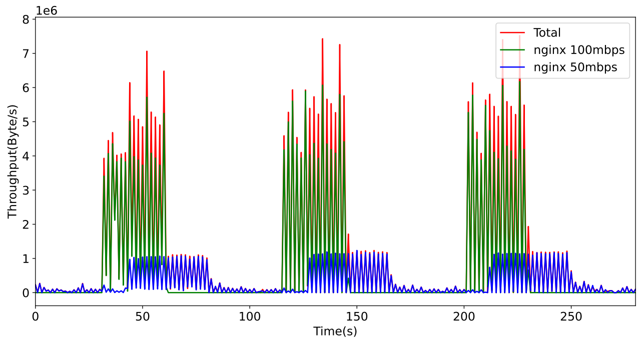

Figure 4.5: Throughput of two nginx web servers accessed by wrk2 with 5000 request rate at pipe 2 and pipe 3 where each instance runs 20 seconds and sleep for 10 seconds periodically

4.5 Heterogeneous Applications in DUT

Similar experiment as 4.1.1, we find out that the 6500 request rate for 100 Mbps provides the least delay for queue size 64 and burst size 64 by tuning parameters explained in Section 3.3.1. For the number of clients, as shown in Fig 4.6, the DUT provides the best quality rate for 50 clients at 200 Mbps capacity link.

First, we have the experiment for running nginx with a 6500 request rate about 48.5 Mbps of offered throughput rate and dashjs with 50 clients at pipe 2 and pipe 3 of 100 Mbps capacity each respectively, and measuring the throughput for each of them as shown in Fig 4.7. Second, we perform the experiment for running nginx with a 6500 request rate of about 48.5 Mbps of offered throughput rate and dashjs with 50 clients at same pipe 200 Mbps capacity as shown in Fig 4.8

The DUT provides QoS for heterogeneous applications running at different pipes in such a way that the average delay for nginx is measured at 14 milliseconds at separate pipes while 151 milliseconds at the same pipe and the average quality level of 50 clients for dashjs application are 3.2 for different pipe deployment while 3.8 for same pipe case. Quality 4 is the best as explained in 3.1.

When we have applications in the same pipe, the delay performance application suffers for approximately the same throughput represented by a red line in both figures 4.8, and 4.7. While the average quality for the clients increases for the dashjs which is represented by a blue line in both figures 4.8, and 4.7 because there is more capacity available in the same pipe configuration (200 Mbps) than in the different pipe (100 Mbps). The resources are shared in the same pipe between applications

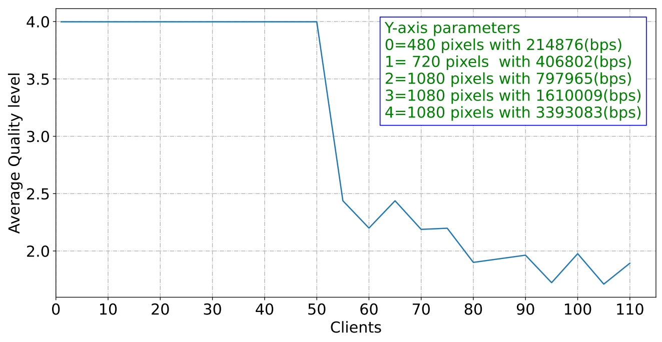

Figure 4.6: Average quality levels for dashjs clients for 200 Mbps capacity of configured pipe

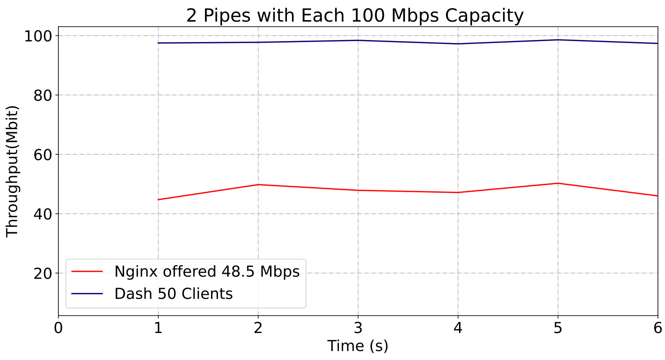

Figure 4.7: Evaluation of throughputs for nginx server providing about 48.5 Mbps of traffic rate and 50 clients accessing dashjs server at two separate pipes of 100 Mbps capacity each

which causes the nginx application to share the scheduling process with the dashjs clients for each iteration among the TCs making a higher delay perceived by the client using the nginx application. On the other hand, the DUT have network isolation causing faster scheduling for the nginx application running in a different network/pipe than the dashjs application providing lesser delay compared to the same pipe case.

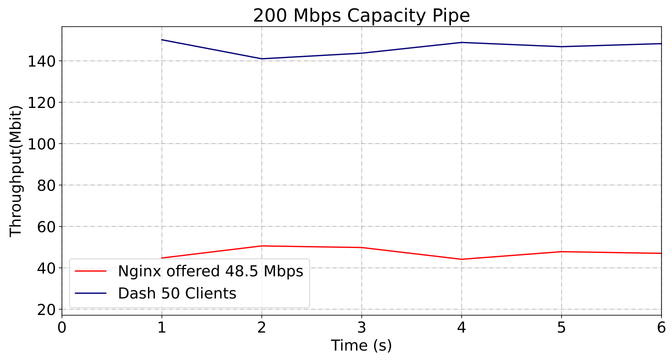

Figure 4.8: Evaluation of throughputs for nginx server providing about 48.5 Mbps of traffic rate and 50 clients accessing dashjs server at the same pipe of 200 Mbps configured capacity

4.6 Discussion Regarding Performance of DPDK

There is a noticeable impact on the performance of DUT by parameters of DUT, DPDK QoS application parameters, and the heterogeneous applications parameters which are sharing the network resources of DUT. Increasing the capacity of a pipe increases the throughput. Lower assigned capacity introduces a small window for the low delay because the capacity is exceeded by the introduced load. On the contrary, higher assigned capacity provides a larger window for a low delay because the link is not exhausted with the increasing amount of traffic as compared to the lower capacity. The effect arrives from the utilization of the link regardless of the link capacity. As the utilization of the link increased, the delay decreased and remains a marginal difference for a window of utilization and again increased drastically for increased utilization. Higher burst size results in higher capacity utilization resulting in large throughput with high delay while low burst size results in low delay and low throughput. Lower traffic at a pipe experience higher delay even though the capacity is significantly higher in the pipe for which cross-application is needed to generate additional packets for DUT which is a wastage of capacity of the DUT. However, higher traffic up to 70-80 % of capacity experience a lower delay. DUT provides better delay performance when different applications are implemented at different pipes compared with running at the pipe. Capacity is shared among applications running at the same pipe whereas the strict isolation of capacity between pipes. Meaning there is performance isolation in terms of capacity utilization between pipes.

Hence, the DPDK performance is better when heterogeneous applications are set in a different pipe with their own capacity compared with adjusting in the same pipe. The queue size and burst size is runtime parameters that provide QoS for all application by DUT which must be taken care of. This is because lower burst and queue size provides low latency but lower throughput for a configured capacity of a pipe. There is this trade-off between latency and throughput bu DPDK which must be taken into account depending on the requirement for the heterogeneous applications sharing the shared network infrastructure.

Chapter 5
Conclusion

Modern internet infrastructure consists of heterogeneous applications that have their own network resource requirement served in a shared network infrastructure. The best-effort approach for delivery packets in the shared infrastructure creates contention of packets at high traffic making unsatisfactory QoE of users. A QoS mechanism can solve the problem and there are numerous methods to apply QoS in a shared network infrastructure from network device based to server-based QoS mechanism. We use QoS framework provided by DPDK which is a server-based approach that has fast packet forwarding capability and implements network slicing concepts for QoS.

A DPDK is a feasible solution for providing QoS for heterogeneous applications in a shared network infrastructure where we use DUT running QoS application provided by DPDK. We use Ansible for deployments of heterogeneous applications and clients accessing the services. All the traffic from clients to the servers and vice versa are forwarded by DUT by providing QoS to the application by changing control knobs of DUT as well as application-specific parameters and resulting QoS is evaluated. We capture pcap file at a client interface, plot the traffic pattern, and evaluate the changing in the metrics to conclude the performance of DPDK.

A DPDK is a complex system whose performance changes by varying its control knobs as well as the parameters that come with the applications. For instance, reducing burst size reduces the delay metric of DUT but also reduces the throughput, changing the capacity of a link increases throughput, and increasing the number of clients in different networks/pipes reduces the delay of applications. We can isolate available network resources in different network slices and schedule the QoS for applications based on using HTB that can be configured during each runtime utilizing the configuration file. Since DPDK can be run in Linux, and FreeBSD operating systems, a hardware-independent, server-based open-source program that can be modified according to need, it has a potential for implementing QoS in shared network infrastructure in production which save cost compared with traditional network devices.

One disadvantage of DPDK for QoS framework is that the performance of the DUT reduces for low traffic because of the prefetch stage that includes execution delay before scheduling stages. We need to generate additional packets by means of cross-application to achieve low delay for the application with low traffic in a separate pipe which is clearly a waste of capacity. Another is the modification of the QoS application of QoS that needs to comply with dynamic heterogeneous applications. Also, there is a strict isolation of capacity between pipes that causes idle resources even if an application in a pipe is overloading but free in another pipe.

In summary, QoS framework provided by DPDK can be a solution to the need of providing QoS for heterogeneous applications in shared infrastructure. For future expansion, we can also study the feasibility where DUT changes its configuration of network slicing, resource allocation among network/pipes, and also allocating unused capacity from one network to another network according to dynamic admitted clients of the heterogeneous applications in the system.

References

[1] M. Bosk, M. Gajić, *et al.*, «Using 5g qos mechanisms to achieve qoe-aware resource allocation», in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 283–291.

[2] I. Cerrato, M. Annarumma, and F. Risso, «Supporting fine-grained network functions through intel dpdk», in *2014 Third European Workshop on Software Defined Networks*, 2014, pp. 1–6.

[3] B. F. Koch, «A qos architecture with adaptive resource control: The aquila approach», in *Proceedings of the IEEE International Conference on Communications (ICC)*, Siemens, ICN WN CC EK A 19, Jun. 2000, pp. 1522–1527.

[4] R. Bless, «Towards scalable management of qos-based end-to-end services», in *2004 IEEE/IFIP Network Operations and Management Symposium (IEEE Cat. No.04CH37507)*, vol. 1, 2004, 293–306 Vol.1.

[5] R. P. Esteves, L. Z. Granville, and R. Boutaba, «On the management of virtual networks», *IEEE Communications Magazine*, vol. 51, no. 7, pp. 80–88, 2013.

[6] M. Casado, T. Koponen, *et al.*, «The Road to SDN: An Intellectual History of Programmable Networks», *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[7] N. Alliance, «Description of network slicing concept», *NGMN 5G P*, vol. 1, no. 1, pp. 1–11, 2016.

[8] L. Georgiadis, R. Guérin, *et al.*, «Efficient network qos provisioning based on per node traffic shaping», *IEEE/ACM transactions on networking*, vol. 4, no. 4, pp. 482–501, 1996.

[9] W. M. H. Azamuddin, R. Hassan, *et al.*, «Quality of service (qos) management for local area network (lan) using traffic policy technique to secure congestion», *Computers*, vol. 9, no. 2, p. 39, 2020.

[10] D. G. Balan and D. A. Potorac, «Linux htb queuing discipline implementations», in *2009 First International Conference on Networked Digital Technologies*, 2009, pp. 122–126.

[11] C.-H. Lee and Y.-T. Kim, «Qos-aware hierarchical token bucket (qhtb) queuing disciplines for qos-guaranteed diffserv provisioning with optimized bandwidth utilization and priority-based preemption», in *The International Conference on Information Networking 2013 (ICOIN)*, 2013, pp. 351–358.

[12] B. Siregar, M. Manik, *et al.*, «Implementation of network monitoring and packets capturing using random early detection (red) method», in *2017 IEEE International Conference on Communication, Networks and Satellite (Comnetsat)*, 2017, pp. 42–47.

[13] P. Bosshart, D. Daly, *et al.*, «P4: Programming protocol-independent packet processors», *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[14] H. Jiang, Y. Wang, *et al.*, «Self-supervised learning for 3d object detection in point clouds», *IEEE Transactions on Signal Processing*, vol. 70, pp. 461–474, 2022. [Online]. Available: https://cdn.techscience.cn/ueditor/files/TSP-CMC-67-1/TSP_CMC_14576/TSP_CMC_14576.pdf.

[15] L. Rizzo, «Netmap: A novel framework for fast packet i/o», in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[16] L. Rizzo, R. Sombrutzki, and A. Kirk, «VALE: A Switched Ethernet for Virtual Machines», in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012, pp. 1–6. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[17] FreeBSD, *Vale*, Ubuntu Manpage Repository, Apr. 2021. [Online]. Available: https://manpages.ubuntu.com/manpages/bionic/man4/vale.4freebsd.html.

[18] Scott McCarty (2018) Architecting Containers Part 1: Why Understanding User Space vs. Kernel Space Matters [Online]. Available: [Online]. Available: https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters (last visited: Jan. 16, 2023).

[19] Virtualization Consultant (2018) Virtual Networking: Poll Mode vs Interrupt [Online]. Available: [Online]. Available: https://nielshagoort.com/2017/10/13/virtual-networking-poll-mode-vs-interrupt/ (last visited: Jan. 16, 2023).

[20] Memory Headroom . Available: [Online]. Available: https://www.atarimagazines.com/compute/issue127/90_HeadRoom_20.php (last visited: Jan. 16, 2023).

[21] Environment Abstraction Layer . Available: [Online]. Available: https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html?highlight=what%5C%20eal (last visited: Jan. 16, 2023).

[22] Internal data structures per port . Available: [Online]. Available: https://doc.dpdk.org/guides/prog_guide/qos_framework.html?highlight=grinder.

[23] QoS Scheduler Sampling Application. Available: (last visited: Jan. 16, 2023).

[24] Throughput calculation in Bits per second and Packets per second. Available: [Online]. Available: https://community.arubanetworks.com/community-home/digestviewer/viewthread?MID=30778.

[25] K. Lai and M. Baker, «Measuring link bandwidths using a deterministic model of packet delay», in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '00, Stockholm, Sweden: Association for Computing Machinery, 2000, pp. 283–294. [Online]. Available: https://doi.org/10.1145/347059.347557.

[26] Definition of Cloud . Available: [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/ (last visited: Mar. 19, 2023).

[27] M. Chen, Y. Qian, *et al.*, «Data-driven computing and caching in 5g networks: Architecture and delay analysis», *IEEE Wireless Communications*, vol. 25, no. 1, pp. 70–75, 2018.

[28] R. d. S. M. Júnior, A. P. Guimaraes, *et al.*, «Sensitivity analysis of availability of redundancy in computer networks», *CTRQ 2011*, p. 122, 2011.

[29] Packet switched Network delay. Available: [Online]. Available: https://www.vmware.com/topics/glossary/content/network-automation.html#:~:text=Network%5C%20automation%5C%20is%5C%20the%5C%20process,in%5C%20conjunction%5C%20with%5C%20network%5C%20virtualization. (last visited: Mar. 19, 2023).

[30] M. P. Robillard, E. Bodden, *et al.*, «Automated api property inference techniques», *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[31] C. Ebert, G. Gallardo, *et al.*, «Devops», *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[32] A. Leff and J. T. Rayfield, «Integrator: An architecture for an integrated cloud/on-premise data-service», in *2015 IEEE International Conference on Web Services*, 2015, pp. 98–104.

[33] G. Suciu, E. G. Ularu, and R. Craciunescu, «Public versus private cloud adoption — a case study based on open source cloud platforms», in *2012 20th Telecommunications Forum (TELFOR)*, 2012, pp. 494–497.

[34] Ansible . Available: [Online]. Available: https://www.ansible.com (last visited: Mar. 19, 2023).

[35] N. K. Singh, S. Thakur, *et al.*, «Automated provisioning of application in iaas cloud using ansible configuration management», in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, 2015, pp. 81–85.

[36] V. Sinha, F. Doucet, *et al.*, «Yaml: A tool for hardware design visualization and capture», in *Proceedings 13th International Symposium on System Synthesis*, IEEE, 2000, pp. 9–14.

[37] Ansible Architecture. Available: [Online]. Available: https://www.educba.com/ansible-architecture/ (last visited: Mar. 19, 2023).

[38] Kubernetes . Available: [Online]. Available: https://kubernetes.io/ (last visited: Mar. 19, 2023).

[39] Delay measurement for HTTP request and response. Available: [Online]. Available: https://github.com/giltene/wrk2.

[40] DASH streaming service. Available: [Online]. Available: https://github.com/fg-inet/DASH-streaming-setup (last visited: Mar. 19, 2023).

[41] Download DPDK . Available: [Online]. Available: https://fast.dpdk.org/rel/dpdk-22.11.1.tar.xz (last visited: Mar. 19, 2023).

[42] DPDK quality of Service application. Available: [Online]. Available: https://doc.dpdk.org/guides/sample_app_ug/qos_scheduler.html?highlight=qos_sched (last visited: Mar. 19, 2023).

[43] Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation. Available: [Online]. Available: https://doc.dpdk.org/guides/prog_guide/qos_framework.html?highlight=dequeue.

[44] Subport Traffic class overscription. Available: [Online]. Available: https://doc.dpdk.org/guides/prog_guide/qos_framework.html?highlight=qos%20scheduler (last visited: Apr. 16, 2023).

[45] Socket buffer size in Linux operating system . Available: [Online]. Available: https://man7.org/linux/man-pages/man7/tcp.7.html#:~:text=The%5C%20default%5C%20value%5C%20is%5C%2016,used%5C%20by%5C%20each%5C%20TCP%5C%20socket. (last visited: Mar. 19, 2023).

Appendix A

Appendix

This section presents the configuration file of DPDK QoS application used for implementing the experiments.

A.1 Default Sample DPDK QoS Configuration File

This is a sample DPDK QoS configuration file provided in the **43**.

Listing A.1: Sample configuration file**43**

```
; Port configuration
[port]
frame overhead = 24
number of subports per port = 1

; Subport configuration
[subport 0]
number of pipes per subport = 4096
queue sizes = 64 64 64 64 64 64 64 64 64 64 64 64 64

pipe 0−4095 = 0                ; These pipes are configured with pipe p

[subport profile 0]
tb rate = 1250000000           ; Bytes per second
tb size = 1000000              ; Bytes

tc 0 rate = 1250000000         ; Bytes per second
tc 1 rate = 1250000000         ; Bytes per second
tc 2 rate = 1250000000         ; Bytes per second
tc 3 rate = 1250000000         ; Bytes per second
tc 4 rate = 1250000000         ; Bytes per second
tc 5 rate = 1250000000          ; Bytes per second
tc 6 rate = 1250000000          ; Bytes per second
tc 7 rate = 1250000000          ; Bytes per second
tc 8 rate = 1250000000          ; Bytes per second
tc 9 rate = 1250000000          ; Bytes per second
tc 10 rate = 1250000000         ; Bytes per second
tc 11 rate = 1250000000         ; Bytes per second
tc 12 rate = 1250000000         ; Bytes per second

tc period = 10                  ; Milliseconds

; Pipe configuration
[pipe profile 0]
tb rate = 305175                ; Bytes per second
tb size = 1000000               ; Bytes

tc 0 rate = 305175              ; Bytes per second
tc 1 rate = 305175              ; Bytes per second
tc 2 rate = 305175              ; Bytes per second
tc 3 rate = 305175              ; Bytes per second
tc 4 rate = 305175              ; Bytes per second
tc 5 rate = 305175              ; Bytes per second
tc 6 rate = 305175              ; Bytes per second
tc 7 rate = 305175              ; Bytes per second
tc 8 rate = 305175              ; Bytes per second
tc 9 rate = 305175              ; Bytes per second
tc 10 rate = 305175             ; Bytes per second
tc 11 rate = 305175             ; Bytes per second
tc 12 rate = 305175             ; Bytes per second

tc period = 40                 ; Milliseconds

tc 12 oversubscription weight = 1

tc 12 wrr weights = 1 1 1 1
```