Nimble: Scalable Rate-Limiting on Today's Programmable Switches

BY

KOMAL SHINDE
B.Tech, Dr. Babasaheb Ambedkar Technological University, 2016

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2020

Chicago, Illinois

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my master thesis advisor, Dr. Brent Stephens, for his unwavering support. He is an excellent source of knowledge and working with him has helped me gain a deeper understanding of computer networks. I am grateful for the opportunity he has given to work as a graduate research assistantship in his lab. It was inspiring to see his work and ability to come up with creative ideas. He has always been there and gone beyond in guiding me meet my career goals. I will always be grateful to you for giving me the opportunity to present our work at NSDI'20 conference.

Secondly, I would to like to thank Dr. Balajee Vamanan, and Dr. Besma Smida, for being members of my defense committee. I wish to thank my reasearch team members: Balaji Vamanan, and Mojtaba Malekpourshahraki who I am fortunate to work and collaborate with. All of you have helped me improve with feedback on my work, through meetings in-person and online. Finally, I wish to thank my family and friends for their support and encouragement throughout my study.

KS

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

PIFO	Push-In First-out
IXP	Internet Exchange Point
FIFO	First In First Out
KB	2^{10} Bytes
MB	2^{20} Bytes
GB	2^{30} Bytes
Gbps	10^9 bits per second
ECN	Explicit Congestion Notification
TC	Traffic Class
DC	Data Center
UIC	University of Illinois at Chicago

SUMMARY

Sharing the network resources in today's network has been quite challenging where multiple tenants and applications with different requirements such as delay-sensitive applications like VOIP, video streaming, and throughput sensitive applications like file transfer, email services are competing with each other. Since a multi-tenant network collates lots of different services, it is important to have better isolation between them. Rate limiters are key implementation in network QoS management such as performance isolation and policy-based bandwidth allocation. Rate limiting can be performed both on software as well as hardware. There are many recent work on rate limiters [1], and [2], but they lack scalability. This thesis solves the performance isolation problem by implementing scalable rate limiters which provides high throughput, low latency, high precision, and quick updates.

Chapter 1 describes the motivation behind this thesis. It will list the challenges faced by today's network devices. Later discussing how Nimble has overcome these problems and contributed to providing performance isolation. Chapter 2 gives background on rate limiters, different congestion control mechanisms, and programmable networks. Chapter 3 will describe Nimble design and implementation in detail. It explains how Nimble enforces rate limiting policy and implements a leaky bucket algorithm. Chapter 4 will show evaluation from experiments performed and validate Nimble implementation. Chapter 5 will be the overall conclusion about Nimble.

CHAPTER 1

INTRODUCTION

Speeds of today's networks are rapidly increasing, and it is important to provide applications and users with performance that is close to the raw underlying network performance [3,4]. We can measure the network performance in terms of both throughput and latency. Throughput can be characterized in bits successfully transmitted during a specific interval of time divided by the time. It is important to provide both high throughput and low latency to applications. There are many different applications sharing the networks like latency-sensitive applications such as web servers, search, and key-value stores, throughput-sensitive applications such as web indexing, and applications with higher priority [5–8]. These varieties of different applications have different performance metrics and requirements. Anytime applications share the resources, e.g., send packets to the same output port of a switch, they are competing for the performance and result in impacting each other's performance [9]. Thus, efficient resource sharing in the network is significant.

Today's networks such as data centers and IXPs forward from a large number of competing tenants and applications. These multiple tenants have to share the network like bandwidth and buffer space according to some high-level policy [5–7,9–11]. TCP is one way to handle sharing which uses a congestion control mechanism to effectively share network resources. Unfortunately, this approach has considerable limitations. For example, an application with a high-speed connectivity can hog the available network bandwidth. This can impact the throughput and latency of other low rate applications such as web traffic, real-time applications like VOIP, and cause this applications to suffer from high latency and low throughput.

Rate limiters are tool that many systems today have used to address this performance isolation problem [1, 2, 5, 6, 9, 12–14]. Correctly implemented rate limiters can reduce latency and increase throughput. In general, this is done by using the rate limiters to divide up the available capacity of the network. This ensures that we are never oversubscribed and the traffic under one rate limit does not impact the traffic under another rate limit [15, 16].

In general, there are two places to implement rate limiters, one of them is at the edge and other inside the network. There has been a lot of recent work and many different systems that have successfully implemented rate-limiting at the edge [2, 12, 14, 17]. Unfortunately, this approach to rate-limiting is not useful in every scenario. There exist networks like IXPs which need isolation but the network operator does not have control over the end-host. Additionally, in a hierarchical rate-limiting system like Google's BwE [9], some types of policies are easiest to precisely enforce inside the network.

In particular, in-network rate-limiting can be provided either with software or with hardware. However, neither of the approach is suitable for every use case. Software rate limiters are scalable. Unfortunately, hardware rate limiters are not scalable. Similarly, hardware rate limiters provides high throughput whereas software rate limiters impact throughput.

In more detail, we present the following two scenarios to further illistrate the usefulness of rate limiters.

Scenario 1: Consider three traffic classes, namely, red, blue, and green sharing a port in a 100 Gbps line rate switch. When a red traffic class tenant sends UDP traffic at 75 Gbps which is more than what is allocated, this makes the blue and green tenants running TCP to react to congestion control feedback by switch and send at a rate of available bandwidth i.e. 25 Gbps, getting 12.5Gbps each. In this situation, if rate limiters were set to limit the red traffic class rate to 33.3Gbps, it will make 33.3Gbps available to each blue and green tenants maintaining the fair share among them.

Scenario 2: Without rate limiters, TCP and its variance [16, 18] will lead to per-flow fairness. This means when one tenant opens significantly more flows than another, it will get more bandwidth than their fair share. This will then cause unnecessary increased latency and lower throughput. For example, consider two traffic classes, namely, red, and blue sharing a port connected to a 100 Gbps line rate switch. When red tenant opens 3 times more flows than blue tenant, it achieves 75Gbps throughput whereas blue tenant gets 25Gbps. With rate limiters, red traffic flows sharing same queue will drop packets to converge to implied policy rate i.e. 50Gbps.

In this thesis, there was primarily concern with implementing rate limiters inside the network. This is because there are many other systems that can already provide rate-limiting at the edge. There are number of key limitations that prevent today's approaches of in-network rate-limiting from being useful. Specifically, the key limitations to existing approaches that we have identified are as follows:

- **Scalability**: Since the network demands continue to increase with time [3], we expect that today's networks need large number of in-network rate limiters. For example, prior work as estimated a need for 10s to 100s of thousands of rate limiters [1]. Unfortunately, switches available today only provide tens of rate limiters [10]. Due to this significant mismatch today's switches cannot be used to implement rate limiters at the scale of data centers.
- **Low Throughput**: Software rate limiters are not guaranteed to run at 100Gbps line rates [1, 2, 12]. However, some applications require line rate performance [3, 19]. This means the rate limiters cannot be used in every applications or scenarios.
- **High Latency**: There are some data center and IXP applications that demand low latency [20]. Because existing software rate limiters inside the network can incur tens of microseconds latency, they cannot be used in every applications or scenarios [21].
- **Low Precision**: Network congestion events typically happen at a timescale less than single network RTT, which is in the order of microseconds in the data center and IXPs networks [22, 23]. As such, to avoid congestion-related buffer overflows in today's shallow buffer switches, it is necessary to provide nanosecond scale precision rate limiters. Unfortunately, software rate limiters are not this precise [5].
- **Not TCP Friendly**: Because of limited buffering capacity in switches, most in-network rate-limiting schemes enforce rate limits by dropping packets (*e.g.*, traffic policing [24]). Unfortunately, packet dropping hurts application performance because it leads to low TCP throughput and poor tail response times [25].

1.1 Contribution

This thesis introduces Nimble, a new system that performs scalable rate-limiting inside today's programmable switches. Because Nimble is able to provide rate-limiting inside a programmable switch without requiring any queuing or scheduling resources, it is scalable, provides high throughput, provides high precision, and can be updated quickly. Further, because this thesis also introduces a new ECN marking based rate-limiting scheme, Nimble is TCP friendly. Specifically, Nimble overcomes the limitations of previous in-network rate limiter designs in the following ways:

- **Scalability**: Nimble provides scalability in the network by implementing in-switch rate-limiters without requiring any queuing or scheduling resources. Nimble approximates the behavior of virtual queues with fewer resources in existing approaches. This is because Nimble utilizes counters to perform network monitoring and enforce policies. Counters in today's programmable switches only require SRAM which are plentiful [26, 27]. As a result, Nimble is able to scale up to 64K rate limiters given 1MB SRAM.
- **High Throughput**: PISA switches are designed so that applications that run on them can operate at full line rate even if every port is receiving a minimum sized packet [26, 27]. Because of this, the current prototype of Nimble is able to operate at 100Gbps line rates.
- **Fast Dynamic Updates**: Bursty applications can cause congestion at microseconds scales [22]. Nimble addresses this problem by providing low latency updates through the data plane.

- **Precision**: The current prototype of Nimble provides precise rate-limiting. Specifically, this thesis performed experiments that showed that Nimble can enforce rate limits with less than 1% error across a range of rates measured from 1Gbps to 100Gbps.
- **TCP Friendliness** Nimble introduces a novel TCP friendly rate-limiting mechanism that uses ECN marking. This allows Nimble to be more TCP friendly than any existing approaches for in-network rate-limiting.

To demonstrate the effectiveness of Nimble, we have created a prototype implementation and performed a series of evaluations on a 100Gbps high-speed programmable switch. We used three different metrics for our experiments: throughput, latency, and queue length. These experiments showed that using TCP without any rate limiters can lead to reduced throughput, increased network latency, and large queue lengths. The performance of an application is hugely impacted if bandwidth-hungry applications or tenants send more flows or more traffic in the network than allocated bandwidth. In contrast, we found our implementation of Nimble can guarantee high throughput, low latency, and small queue lengths in a multi-tenant network.

1.2 Organization

This thesis is organized as follows Chapter 2 The background on rate-limiting, traffic control mechanism, programmable networking, and virtual queuing is mainly discussed in this chapter along with its scalability limitations. It will introduce P4 programming and a description of PISA architecture. Chapter 3 discusses Nimble, the proposed idea with implementation. Finally, Chapter 4 concludes this thesis.

CHAPTER 2

BACKGROUND

At first, this chapter will see the motivation for this thesis explaining the importance of performance isolation in today's network, then discuss different existing systems like SILO [5], BwE [9], EYEQ [6], SWAN [28], and HULL [13] who have implemented rate limiters in section 2.1. Section 2.2 will provide background on rate limiters, see difference between traffic policing and shaping, and token bucket vs leaky bucket. Section 2.3 will cover different congestion control mechanisms like RED, ECN, and DCTCP. In section 2.4, we will discuss in brief about different programmable SDN networks including Openflow and P4. Section 2.5 gives information on related work and it will conclude by stating the need for having a scalable rate limiter instead.

2.1 Motivation

Network isolation is important in a computing environment where resources are shared especially distributed services with dynamically changing demands and to prevent malicious tenants and applications from affecting other tenants and applications in the shared network. To achieve good performance, a tenant needs to be isolated from other tenants it shares the resources with. There are many existing mechanisms such as TCP congestion-control for network sharing. Even though TCP does scalable resource allocation, it does not provide performance isolation. For example, when workloads from multiple tenants struggle for shared resources, performance of one tenant may get affected by a malicious tenant which consumes more network capacity by opening more flows. Along with throughput, it also impacts the network latency of a genuine tenant's flow. DCTCP improves latency when compared with normal TCP but it still has a non-zero queue length and fairness issues as it converges to per-flow fairness.

Rate limiters can be used to solve this problem. In an SDN network, as long as we know the set of active tenants and applications, we can create allocations. Prior work [1, 2, 5, 12] has shown that rate limiters can overcome the limitations of TCP and DCTCP. In general the way these systems work is, they set rate limits to ensure nobody can exceed the policy-compliant allocation.

There are two different ways to implement rate-limiting, one of them is at the edge and other inside the network. Although edge rate-limiting approach is good and it should exist, but we are going to focus on in-network rate-limiting for two reasons: some networks like IXPs do not have control of the end-host and it can more precisely enforce policies in a hierarchical rate-limiting system.

Rate limiters can be implemented in different places for example on a CPU core, on a NIC, and in a switch. However, existing end-host software rate limiters [5, 6] consume more CPU and memory, increasing the latency. Therefore, there is a need to facilitate in-network rate-limiting without requiring changes to end-host and additional resources. Although today's programmable switches have lots of SRAM, there is so far no approach to utilize this to perform rate-limiting.

There are existing approaches to in-network rate-limiting. Specifically, most in-switch rate-limiting today is done with virtual queuing [29,30]. In traditional switches, all the forwarding traffic from different flows gets buffered at input queues. The method of queuing the packets for ingress port was to store the traffic destined for different egress ports in a single physical queue of the switch. Virtual queueing is a technique used in the network switch to eliminate head-of-line blocking due to packet congestion. Each input port is assigned a separate virtual queue for each output queue. Therefore, without impacting any other flows in the physical queue, congestion on egress port will only block that particular virtual queue. Unlike traditional switches, the virtual queue does not send traffic to egress interfaces until the egress interface is ready to transmit packets. Virtual queue maximizes the utilization of the consumption of resources by avoiding packets that later were dropped.

2.2 Rate Limiting

Bursty traffic is the key reason for congestion in a network. To help manage congestion, packets need to be enforced to transmit at an appropriate rate. Rate limiters in a network are used for preventing the traffic from exceeding certain throughput constraint. It provides better performance by maximizing the network throughput and minimizing end-to-end latency in a multi-tenant network. There are several congestion management approaches to help rate limit traffic. There are two main approaches for rate-limiting: Traffic Shaping, and Traffic Policing. Section 2.2.1 will describe the difference in detail.

2.2.1 Traffic Control Mechanism

Incoming rates are controlled to prevent or reduce congestion in a network. There are various control mechanisms like traffic shaping, traffic policing, traffic discarding, and packet scheduling to achieve traffic control in a high-speed network. Let's discuss some of them.

2.2.1.1 Traffic Shaping

Traffic shaping limits the bandwidth exceeding the committed rate at the interface. It is useful for delay-sensitive applications such as VOIP and video, which do not tolerate congestion. Traffic shaping uses a token bucket mechanism in its implementation. Excess packets are not dropped but pushed in a queue and transmitted in the later available bandwidth. Here, the burstiness of traffic is bounded, insuring the flow not transmitting more than allowed rate and without packet drop. Token will be refilled in a queue at a speed of sending rate, maintaining well-paced output packets.

However, traffic shaping is not possible for in-network rate limiting. it requires more buffer space on switches [26]. It will not be an ideal approach to use traffic shaping in switches with limited buffer.

2.2.1.2 Traffic Policing

Traffic policing monitors the incoming traffic rate and drops or mark packets that exceed the maximum allowed rate. The key advantage of traffic policing is unlike traffic shaping, it does not require additional memory for buffering excess packets. Policing verifies the sender adheres to the specified policy rate.

Traffic policing is not TCP friendly. In policing rate with static mechanism, TCP performance can degrade and can cause significant packet loss with large bucket sizes. A possible solution would be to use dynamic bucket size for rate limiting.

2.2.1.3 First-In First-out

The idea of FIFO queuing, also called first-come, first-served (FCFS) queuing, is simple: The first packet that arrives at a router is the first packet to be transmitted. Given that the amount of buffer space at each router is finite if a packet arrives and the queue (buffer space) is full, then the router discards that packet. This is done without regard to which flow the packet belongs to or how important the packet is. This is sometimes called tail drop since packets that arrive at the tail end of the FIFO are dropped. FIFOs are necessary for implementing rate limiters and its usefulness in the design implementation of Nimble will be discussed further in this thesis.

2.2.2 Traffic Control Algorithm

2.2.2.1 Token Bucket

There are three main components of token bucket: Committed Information Rate (CIR) represents the amount of data that can be transmitted in per unit time i.e. bits per second, Burst size (BC) can be represented in bytes (in policing) and bits (in shaping) which is the amount of data allowed to burst over the configured rate, and Time interval (Tc) is time measured between each burst.

$$CommittedRate = BurstSize/TimeInterval$$

Tokens are added to the buffer depending on the allowed transmitted rate. Policing and shaping updates the buffer differently. Token bucket is monitored for every update. If there are enough token to pass the traffic, then the packet is allowed to forward. If not, traffic shaping will buffer the excess traffic into another buffer, whereas, traffic policing will drop the packet. If the token bucket is full, then the packet is dropped. In traffic policing, a bucket is filled based on:

$$(Time\ difference\ between\ packets * DesiredRate)/8 = Tokens\ Refilled$$

2.2.2.2 Leaky Bucket

It is similar to a token bucket, but the desired rate is achieved with the amount of data that can be drained from the bucket. If the bucket is full, any incoming packet that overflows the bucket which will be discarded. Bucket will be drained at a constant desired rate. A leaky bucket algorithm is mostly used in traffic policing implementation.

2.3 Congestion Control Meachanism

2.3.1 Random Early Detection

RED [31] is an active queue management scheme that computes average queue size and takes the decision of probabilistic packet marking or dropping depending on average queue length. There are two thresholds in a queue considered during the decision, namely, low and high thresholds. When an average queue length is between these two thresholds, probabilistic drop will be implied on the incoming packet. When the queue length is above the max threshold, packets will be dropped.

2.3.2 Explicit Congestion Notification

The Explicit Congestion Notification (ECN) [32] option allows active queue management mechanisms such as RED to probabilistically mark (rather than a drop) packets when the average queue length lies between two thresholds, if both the sender and receiver are ECN-capable (determined at connection setup time. In this case, the receiver echoes back to the sender the fact that some of its packets were marked, so the sender knows that the network is approaching a congested state. The sender should, therefore, reduce its congestion window as if the packet was dropped, but need not reduce it drastically, e.g., set it to one or two segments.

2.3.3 DCTCP

Data Center TCP (DCTCP) [18] is an enhancement to TCP congestion control scheme based on explicit congestion notification (ECN). When queue length exceeds the threshold, switch signals the sender about the congestion in a network by marking the ECN bit, which sender reacts with limiting sending rate. In DCTCP, sender responds to congestion much faster than traditional TCP. Another advantage being, it reduces the latency occurred with rate limiters. DCTCP maintains low queue length, and high throughput, making the network more stable than traditional schemes like TCP. Zhu *et al* [33] proved ECN to be better than delay-based congestion signal. Overall, it can be concluded, ECN is essential in modern congestion control with DCTCP.

2.4 Programmable Networking

2.4.1 SDN

A traditional network switch includes control plane and data plane integrated into a single device form. Control plane is responsible for establishing packet processing policies and routes using distributed protocols. The data plane, in contrast, is responsible for utilizing this computed routes to forward packet in the network. Devices being closed and proprietary does not provide limited interface and programmability for network operators. To address this issue, Software Defined Networking (SDN) [34] evolved with an idea of decoupling control and data plane, providing more flexibility and scalability in network management.

In SDN, programmability is centralized in the control plane called controller and making data plane a part of hardware implementation. This controller sends a command to switches for taking the forwarding decision.

2.4.2 OpenFlow

Openflow [35] is an SDN control protocol. It specifies the protocol header on which it operates. Openflow targets a predetermined set of header fields that process the packet using a small set of predefined actions. This inflexibility makes it impossible to reconfigure protocol processing. There is a need for an extensible approach to implement a flexible mechanism for parsing packets and matching header fields.

2.4.3 Background on P4

P4 [36] is a new domain-specific programming language that can express how a switch is to be configured and how packets are to be processed. P4 programs include the forwarding behavior of a switch described with P4 language components. P4 programs refer to a particular P4 architecture that represents the programming model of a switch. P4 architectures are implemented by software or hardware switches called P4 targets. Target-specific compilers then translate P4 programs into code that can be executed on the P4 target.

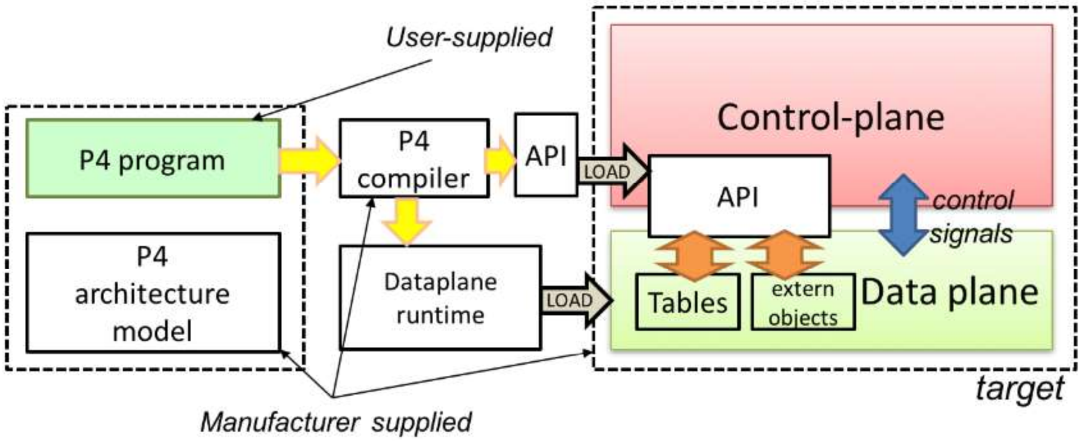


Figure 1: P4-16 Specification

P4 programs are structured in into many components as follows:

2.4.3.1 Headers

Flexibility to describe the packet format which includes ordered sequencing of fields and sizes within a packet. Every design begins with defining header format.

2.4.3.2 Parser

This specification describes how to identify the packet headers and valid header sequences within the packet. It defines a set of transitions from one packet header to another, by extracting the value of current header. All extracted headers proceed further to match + action stage in a switch pipeline.

2.4.3.3 Tables

Allows a user to define match key: a single or multiple packet headers/metadata fields to look against the table and associate them with corresponding actions for forwarding decision. The read attribute includes different types of match types (exact, lpm, and ternary) a header should match, while action attribute consists of possible actions that a packet can take after a match. Tables are implemented using memory such as TCAM or SRAM.

2.4.3.4 Match-Actions

This specification includes a set of match-action stages in the ingress as well as egress pipeline of the hardware switch. Executes an appropriate action at runtime for corresponding matched table key.

2.4.3.5 Control flow

P4 packet processing program defines the order in which the Match + Action tables are applied to the packet, defining the control flow in the network.

2.4.4 PISA Architecture

PISA [37] is a framework providing an abstraction for defining data plane pipeline described using P4, a high-level programming language. PISA model parses packet headers for

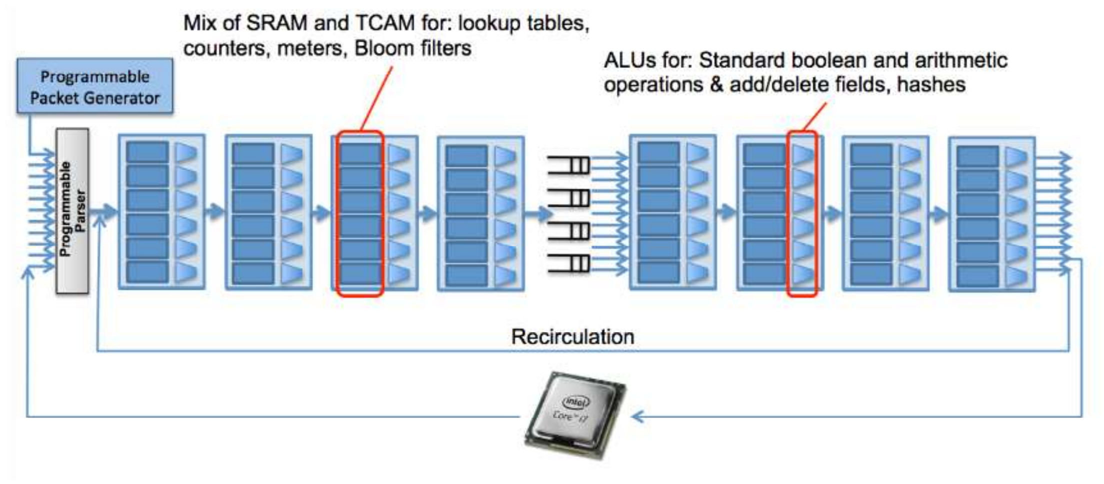

Mix of SRAM and TCAM for: lookup tables, counters, meters, Bloom filters
ALUs for: Standard boolean and arithmetic operations & add/delete fields, hashes
Programmable Packet Generator
Programmable Parser
Recirculation

Figure 2: PISA Switch Architecture

several common protocols but can be extended to extract other information, such as timestamps and queue size from traffic manager. To differentiate from conventional switches, a range of varieties of protocol-independent switch architecture (PISA) switches (e.g., Barefoot Tofino, Netronome) offer features like programmable parsing, customizable packet processing pipeline, generic match-action logic with stateful registers, counters, and meters, as well as fixed-function queueing logic.

Figure 2 shows how PISA switches allocate packet vector header (PHV) for every incoming packet in the parser. PHV can be a combination of both standard headers like ipv4, ipv6, as well as custom metadata. Ingress and egress pipeline consists of a set of match-action units (MAUs). These MAUs comprise of around 10MB of memory (SRAMs and TCAMs) [37]. At MAU, a set of custom actions defining forwarding rules get executed on matching packet header fields and metadata. For reading states at subsequent stages, it needs to be store in a metadata field. State of a packet is not transferable and maintains to that only stage. When a packet reaches last pipeline i.e. programmable deparser, it gets serialized with modifies headers and payload, and back onto the wire for queueing. Along with stateful memory, there are also hardware in-built timers that can be invoked in the switch CPU and utilize to carry out periodic computation such as calculating interpacket gap time or updating match-action table entries with packet interval.

2.5 Related Work

Rate limiting is an important primitive for maintaining network resources. There has been considerable work to implement rate limiters on end switches like SENIC [1], Carousel [2], and Eiffle [12]. SENIC applys to NIC and delivers scalable hardware rate limiting. Eiffle, and Carousel are good examples that implement efficient and scalable rate limiters on software. Below are the existing systems which have implemented rate limiters.

Silo: provides guaranteed latency and bandwidth with burst allowance for multi-tenants in data centers by using token-based rate limiters. It does that by limiting the bandwidth as well as burst size for tenants exceeding their fair share.

Bandwidth Enforcer: Google's Bwe [9] does rate-limiting at end-host in a inter-DC network. It deals with the available bandwidth to maximize the average available bandwidth and provide min-max fair allocation. This bandwidth enforcer in Google's B4 [38] SDN WAN allocates bandwidth at the granularity of tunnels.

EyeQ: EyeQ uses rate limiters to provide bandwidth allocation to tenants in a distributed data center network. It is implemented at the edge to achieve minimum bandwidth guarantee.

HULL: HULL uses token bucket rate-limiter to configure rate, ensuring high bandwidth utilization and maintains queue length at zero, resulting in a zero queuing latency.

One of the solutions to provide isolation in the network is with a virtual queue. However, many currently available fixed-function, as well as state-of-the-art programmable switches, are not capable of supporting a large number of virtual queues. They support around ten virtual queues [15, 39, 40]. Recent advances in programmable switches scheduling [29, 41] could only support 1K rate limiters. Today's IXP network may have more than 500 participants with many applications from each. Network operators may not have control over all these machines, resulting in the inefficiency of rate-limiting at the end-host. Rate limiting in switches provides performance isolation. However, there is no significant active research in switch rate-limiting. This raises the need for scalable rate limiting in switches. We need new programming abstractions that allow participating networks to create and run these applications and a runtime that both behaves correctly when interacting and ensures that applications do not interfere with each other.

CHAPTER 3

NIMBLE

This chapter introduces the design and implementation of a scalable rate limiter system on programmable PISA switches. For the purpose of exposition, I will first describe the PISA programming model and then discuss how the features that comes with this model can be utilized in Nimble implementation. Given the limited resources in PISA reconfigurable switches, such as stages and memory, we will plan to make the best use of these limited resources in the proposed implementation. While there were significant contribution made in the rise of programmable networks, the scheduling queue logic is still a fixed-function block in the reconfigurable switch. The contribution of Nimble is to utilize the capabilities of switch such as memory resources to increase scalability, high throughput, and reduce latency. Current Nimble model is implemented on both protocol independent switches, hardware (Barefoot Tofino) and software (Bmv2).

3.1 Design

Figure 3 illustrates the design overview of Nimble. Nimble is a new leaky bucket rate limiter system for programmable data planes that is implementable in the match+action pipeline of a PISA switch. The general approach is to use a leaky bucket meachanism for controlling the global rate on per-port, and per traffic class basis. Figure 4 shows the table dependency graph of a Nimble P4 program. This graph describe the control flow between tables with matched input and output fields. There are three main components to the design of Nimble,

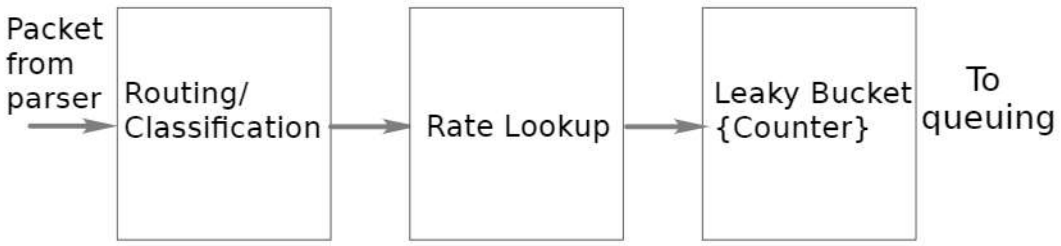


Figure 3: Nimble Design Overview

namely, Routing/Classification, Rate Lookup,and leaky bucket. The first stage in the pipeline, Routing/Classification 3.1.1, is used to identify the traffic class and look up for specific rate-limiter that should be applied to each packet. The second stage, Rate Lookup 3.1.2, uses the traffic class to find the operator configured rate-limit. The third stage, leaky bucket 3.1.3, tracks the occupancy of the leaky bucket and takes the decision whether to drop a packet or not. Importantly, this architecture is a pipeline that only feeds data forward, and only a single piece of state is ever updated at any stage. This is needed to ensure that this design is compatible with PISA switches [37]. The rest of this section describes these components in more detail.

3.1.1 Routing/Classification

Today's cloud multi-tenant networks have to virtualize the network with hundreds of VMs in a physical machine and that will need tens of thousands of flows per end-host [2]. As state-of-

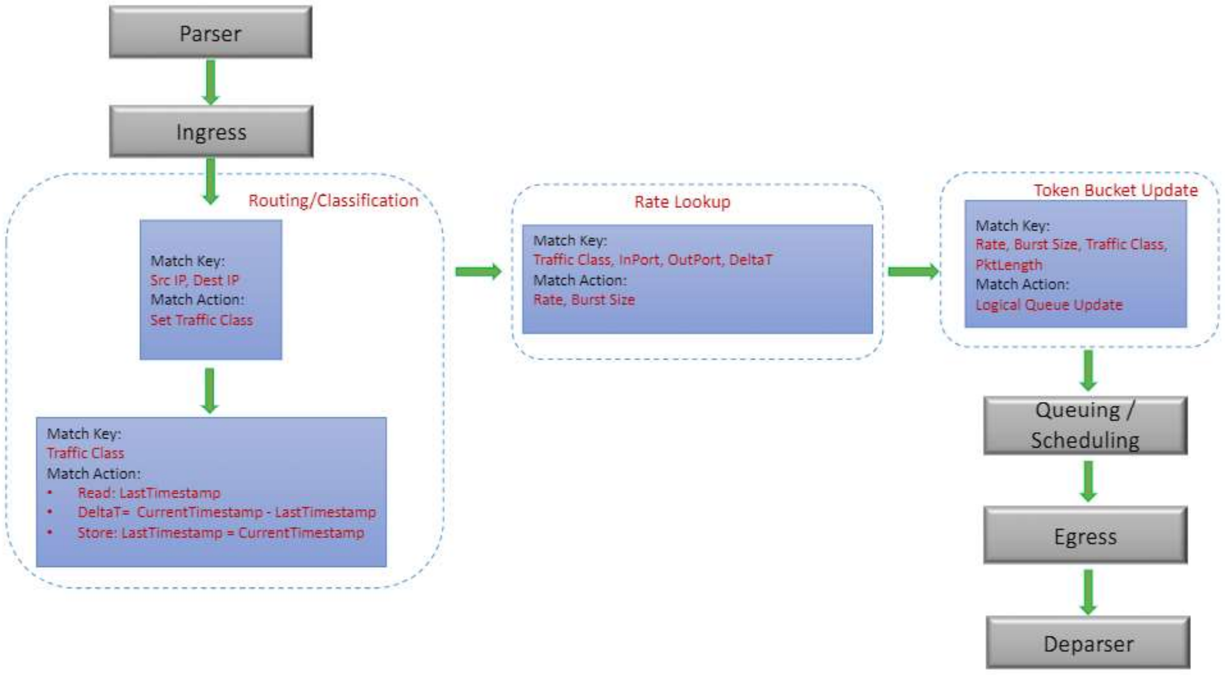


Figure 4: Nimble Table Dependency Graph

the-art packet schedulers like PIFO in programmble switches could support only around 1000K flows, which are not sufficient to scale at data centers. Nimble could programmatically scale to tens of thousands of traffic classes/ flows.

PISA based switches can utilize intrinsic metadata and extern objects that would provide a programmable interface to the pipeline. It matches on metadata or header fields and apply simple packet processing action. Nimble operates independently of however routing is performed on the switch. Figure 5 shows the first block of the model where it is possible to identify the

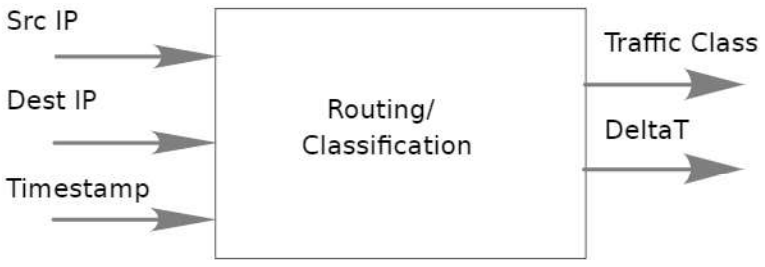


Figure 5: Routing/Classification

traffic classes of a packet from its headers including source address and destination address. Additionally, this block also tracks the timestamp of the last incoming packet for each rate-limiter, which can also be seen in Classification block of Figure 4. Upon receiving a packet at ingress, this block invokes the in-built timers to calculates the time difference of the last observed packet with the current time. This time delta is then saved as metadata for use in the 3.1.2 Leaky Bucket block. Finally,this block updates the time stamp register with the current time and forwards the packet to the next block along with their traffic class.

3.1.2 Rate Lookup

As shown in Figure 6, Rate lookup module is responsible for finding the rate limit for a packet based on its traffic class from previous block 3.1.1. This requires a table to store the necessary metadata for rate-limit value and burst size. Specifically, the rate lookup module will take traffic class, input port, output port, and deltaT as inputs and return a tuple (R,B), where

R is a four byte long rate, and B is a four byte long bucket size/burst allowance. These rates

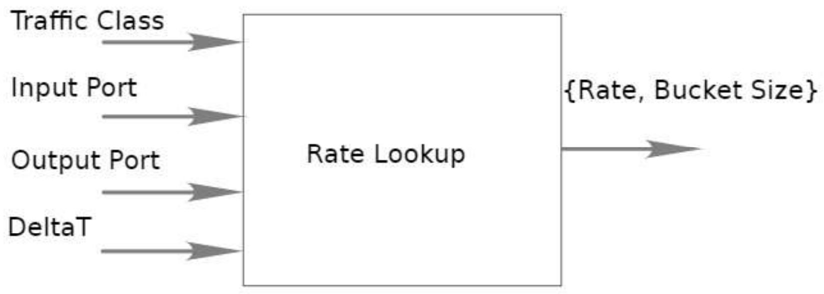


Figure 6: Rate Lookup

and bucket sizes are precomputed and store in tables for lookup. After key match, appropriate lookup, packets are then sent to the next module along with their rate limits.

3.1.3 Leaky Bucket

Nimble implements a leaky bucket algorithm with traffic policing to build scalable rate limiters. This allows for rate-limiters with a configurable burst tolerance in addition to a maximum rate. Nimble enforces the configured rates through policing where packets that exceed their allocated bandwidth will be dropped. To perform rate-limiting, this module keeps track of the current occupancy of each rate-limiters leaky bucket with a counter or register.

Figure 9 well demonstrates this module, for every packet it will update the appropriate rate-limiter based on the size of the current packet and the number of bytes that should be drained from the bucket given the configured rate and the time since the last packet that used this rate-limiter. This module will take rate and burst allowance from previous block 3.1.2,

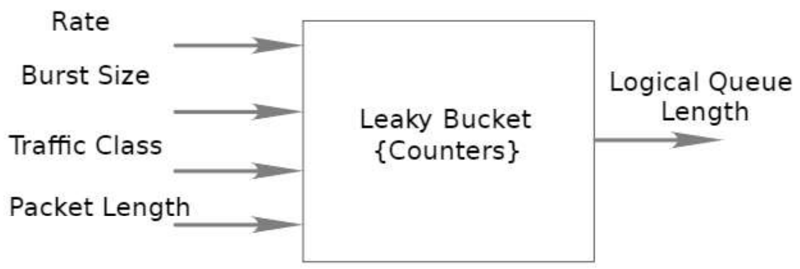


Figure 7: Leaky Bucket

Traffic Class as an index of the counter to update, and size of current packet. The number of bytes to drained from the bucket are precomputed by multiplying the configured rate limit by the time since the last packet and stored in tables. This drained value, the size of the current packet, and the current value of the counter for the bucket are then used to update the counter and decide if a packet is dropped or not. If the packet size minus the drain bytes causes the counter to exceed B, the burst size, the counter will be draining only the drain bytes and packet

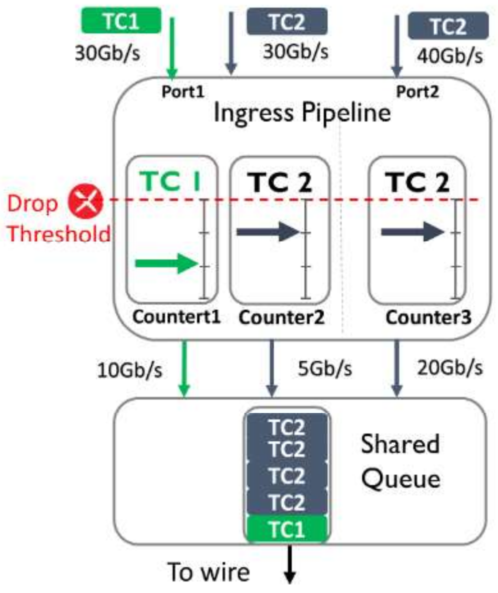


Figure 8: Nimble Illustration

will be dropped. If the counter would decrease below zero, indicating an empty bucket, the counter will be capped at 0, and packets will be allowed to pass. Otherwise, the counter is updated by adding the packet size minus the drain value. Below pseudocode1 will give a better understanding of the queue updates.

Figure 8 gives an illustration of Nimble on a PISA switch. Let's understand the naive approach for packet scheduling with queue. When packet from traffic class arrives at switch, global queue accepts and updates the queue. Then packets from traffic class 2 destined to egress port 1 arrives, global queue accepts it. Later, packets from TC 2 destined for port 2 gets accepted by global queue as it has the capacity of 5 packets. It will update and gets filled.

Now, when packet from TC 1 arrives, global queue drops the packet since it has no space. This creates unfair distribution of packets. Nimble overcome this limitation and avoid packet drop by assigning a seperate counter per-port and per-traffic class. These counters/registers from match+action pipeline of a PISA switch proves to approximate the behavior of virtual queues and resolves the scalability problem in a network by making available 64K rate limiters by utilizing in-built memory like TCAMs and SRAMs.

Algorithm 1: Token Bucket Update

```
Input  : Packet Length, Drain Bytes
Output: Updated queue length
foreach Arriving packet at Ingress do
    if QueueLength + PktLength - DrainBytes > Drop Threshold then
        QueueLength=QueueLength - DrainBytes
        Drop Packet
    else
        QueueLength=QueueLength + PktLength - DrainBytes
    end
    if QueueLength > ECN Threshold then
        Mark ECN bit indicating congestion
    end
end
```

3.2 Implementation

In this section, we describe an implementation of the Nimble as described by the design in 3.1. Nimble is implemented as a P416 program on two PISA switches: the BMV2 software switch, a behavioral model [42] ; and the Barefoot Wedge 100B-65X (Tofino) [37]. Table I gives information on total lines of P416 source code required to implement Nimble on both the PISA switches. It required ~ 300 lines of P416 source code [36] on Bmv2 Software switch, and around ~ 500 on Tofino Barefoot switch. Since programmable switches already have tens of megabytes of memory resources (ex. SRAM) in form of counters, registers, and meters for flow monitoring purposes, these registers can be repurposed to implement logical queues with low overhead. The aim of this implemented is to show that any funtionality that is performed with virtual queue can be approximated by logical queue with low resource overhead.

Target	Lines of code
BMV2	~ 300
Barefoot Tofino	~ 500

TABLE I: P416 LINES OF CODE

CHAPTER 4

EVALUATION

This chapter will describe the evaluation method used to validate the performance of both of our Nimble based implementation described in the previous chapter. At first, I will describe the methodology of the experiments and the measurements. In the following, we will quantify how our approach makes efficient use of counters and registers to implement rate limiters achieving high throughput, low latency, and high performance in terms of queue length.

4.1 Methodology

After having analyzed the theory behind Nimble, now we can look at its practical implications. There will be an evaluation on two implementations i.e. Nimble with meters, and Nimble with logical queues. At last, we will observe the key difference between both the rate-limiting approaches. Evaluation of Nimble is done on both software (BMV2) as well as hardware switch (Barefoot Tofino). For software, I used BMV2 [42], a P4 software switch behavioral model, and mininet [43], an emulation environment. After having implemented and analyzed the performance of our approach on software switch, I performed tests and demonstrated the feasibility of Nimble on the hardware switch. This solution can be implemented in real hardware. For evaluation, I used Barefoot Tofino, a high-speed P4 switch running the latest P4-16 specification program defined with TNA architecture.

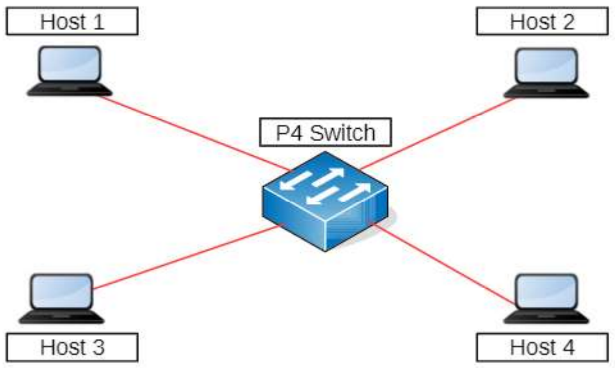


Figure 9: Network Topology

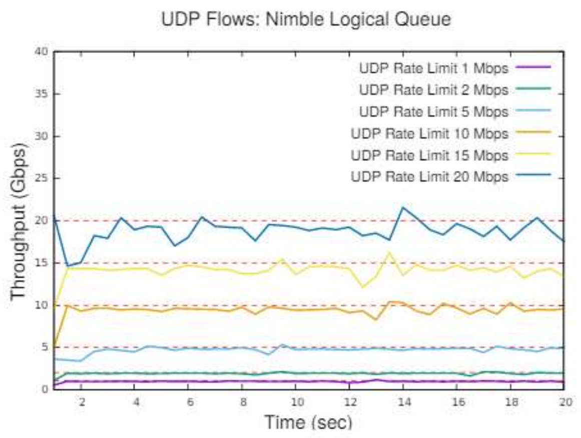
4.2 Testbed

Software switch experiments are conducted on Linux Virtual Machines (2-core Intel Core i7-8550U CPU @ 1.80GHz, 2000MiB memory) running p4c-bmv2 compiler and mininet with 4 virtual hosts connected to BMV2 switch. Bmv2, a P4 software switch, is a behavioral model switch defined with V1Model architecture. To deploy the evaluation setup, I used a network emulator, mininet. This system is implemented and tested on an Ubuntu 16.04 virtual machine. To generate flows, I used the iperf [44] network benchmark tool. Because bmv2 is not designed to be a high-throughput switch, in our experiments, it does not achieve more than 20 Mbps throughput. Our hardware switch, experiments are conducted on 4 servers (8-core Intel Xeon 1.80GHz, 62GiB memory), Barefoot Tofino switch with 22MB buffer shared by 32 100G ports.

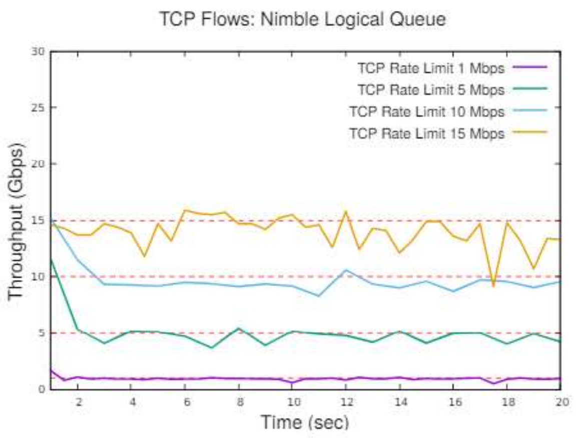
Figure 9 shows the network topology for Nimble. It has 2 hosts and 1 P4 switch. Different traffic rates with various data sizes were transferred in experiments using iperf tool to measure network metrics such as throughput, packet losses, latency, and RTT. Iperf server and client were used to establish TCP and UDP connection.

4.3 Bechmarks

4.3.1 Correctness/Precision



(a) UDP Flows



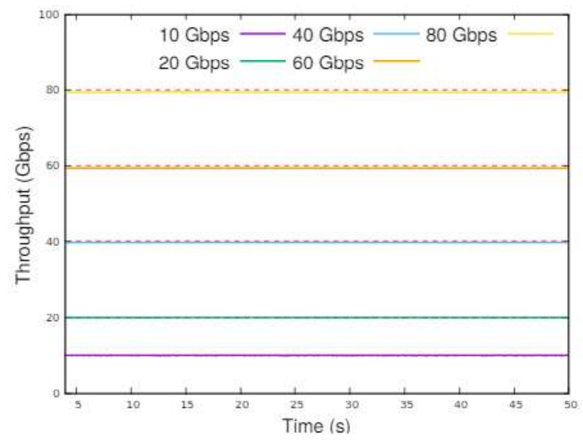
(b) DCTCP Flows

Figure 10: Bmv2 Rate Limiting

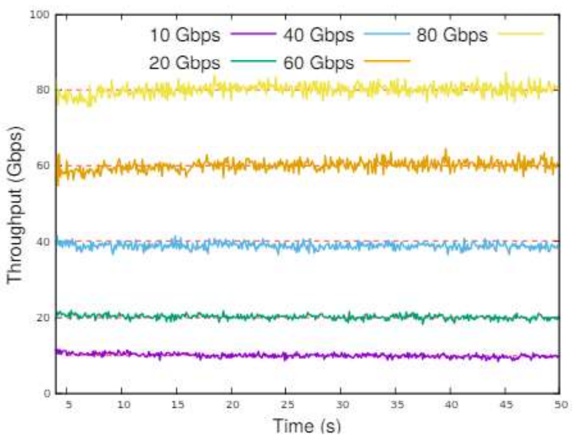
There are sets of different metrics used for the experiments: Throughput, latency, and queue length. Throughput is measure by iperf for both UDP, and DCTCP flows. Tail latency is measured by sockperf, and logical queue length is monitored with and without ECN marking to show the efficiency of congestion control algorithms. DCTCP is enabled on servers which will set ECE bit if the logical queue length crosses ECN threshold to avoid packet drop. To show the correctness and precision in Nimble, two different experiments on Bmv2 Software switch were performed using UDP and DCTCP flows. These experiments show rate limiting using logical queues perform equivalent to having static rate limits on dedicated physical queues. Figure10(a) shows the achieved throughput of UDP flows with a ranging rate limit set (1Mbps, 2Mbps, 5Mbps, 10Mbps, and 15Mbps). Dotted red lines indicate an ideal rate limit value to match. It was observed that the flow achieves close to their desired rate limits.

To evaluate DCTCP, I set up an experiment with hosts having ECN capabilities. Figure10(b) shows the achieved throughput of DCTCP flows with a ranging rate limit set (1Mbps, 5Mbps, 10Mbps, and 15Mbps). DCTCP is implemented with ECN threshold set as 200KB considering 1MB queue size. ECN marking avoids packet loss by drop and maintains queue depth below ECN threshold. Dotted red lines indicate an ideal rate limit value to match. It was observed that the flow achieves close to their desired rate limits.

Evaluation on Barefoot Tofino switch is divided into two sets namely, Nimble using logical queues and Nimble using meters. For both implementations, UDP and DCTCP flows were used for analyzing network throughput. To avoid flow slow start and end, all graphs of Nimble implementation will begin from $4s$ and end at $50s$. Figure 11(a), demonstrates the achieved throughput of UDP flows with ranging rate limit set (10Gbps, 20Gbps, 40Gbps, 60Gbps, and 80Gbps). Line rate for tofino switch is 100Gbps. Throughput average accuracy error for UDP



(a) UDP Flows



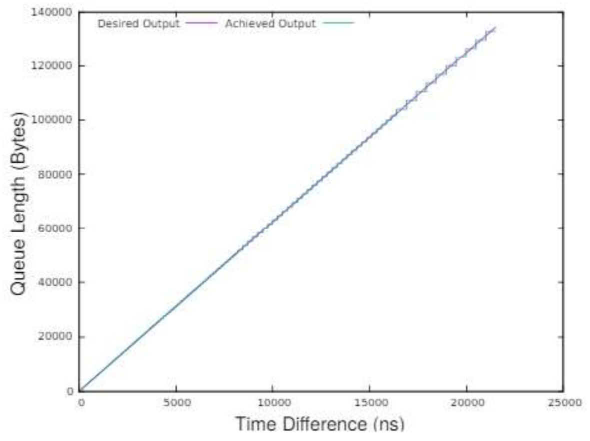
(b) DCTCP Flows

Figure 11: Nimble with Logical Queue

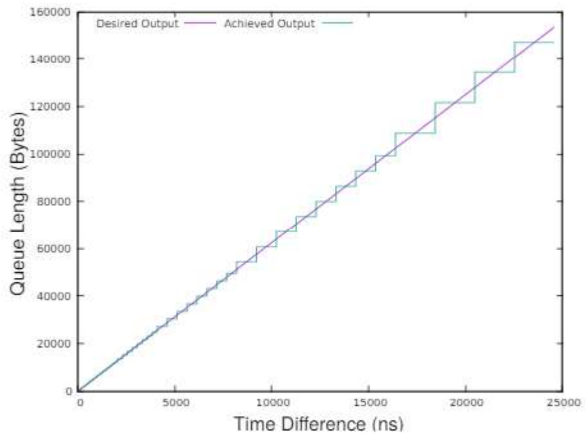
flows is observed to be less than 0.5%. It can be depicted from above figures that the achieve throughput to be equivalent to rate limit set by a physical queue.

For DCTCP flows, hosts are ensured to have ECN capabilities and ECN threshold is set to be 150KB. Different rate limit values are set (10Gbps, 20Gbps, 40Gbps, 60Gbps, and 80Gbps). Figure11(b) shows the achieved throughput using logical queue is close enough to desired rate limit value. Maximum error of 1%, 6%, and 12% in table entries for the rates are shown in 12(a),12(b),12(c), and 12(d) respectively. Thus, these experiments show the usefulness of logical queues and that they can be implemented to approximate the behavior of virtual queues.

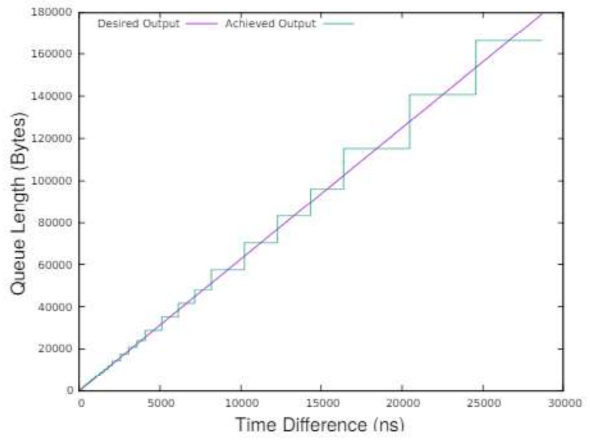
Meters in PISA switch are utilized for rate-limiting incoming traffic. Meter keeps a track of packet and return the output in color. Meters are 3 state markers with two thresholds, namely,



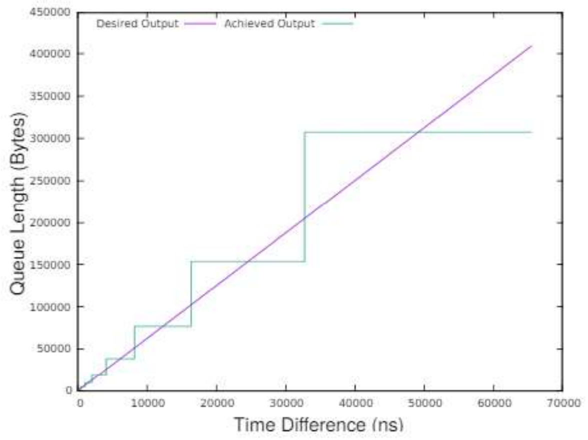
(a) 1% Percent



(b) 6% Percent



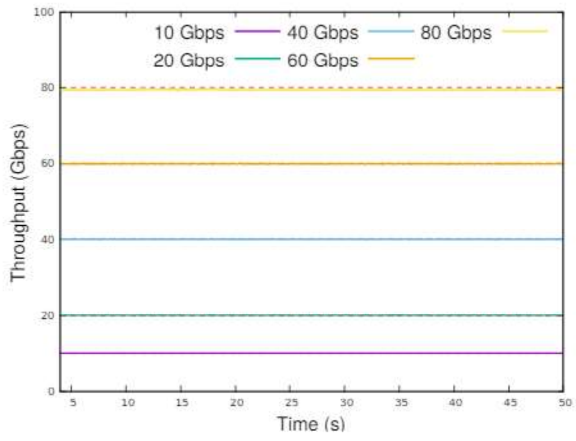
(c) 12% Percent



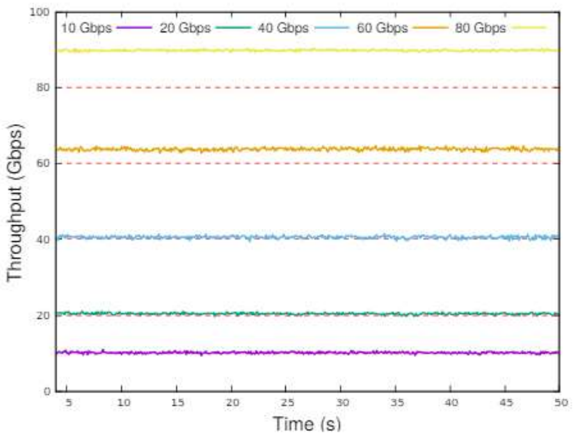
(d) 50% Percent

Figure 12: Maximum Percent Error

minimum threshold and maximum threshold. Meter outputs color green if the rate is below the minimum threhold, color yellow if rate is in between minimum and maximum threshold, and color red when it exceeds maximum thresold. In Nimble with meters, experiments were



(a) UDP Flows



(b) DCTCP Flows

Figure 13: Nimble with Meters

performed using both UDP and DCTCP flows. The set of different rates applied were 10Gbps, 20Gbps, 40Gbps, 60Gbps, and 80Gbps, silimar to rates while testing using logical queues. It can be concluded from figure13(a) and figure13(b) that using meter can achieve throughput equal to desired rate. For DCTCP flows, if the rate exceeds allocated bandwidth, meter will mark packet with ECN bit to indicate congestion.

4.3.2 TCP Friendliness

Monitoring the queue length in a rate limiter is another important metric to analyze the behavior of network system. As we saw in latency experiment how congestion control algorithm like DCTCP has reduced the latency in the network, similarly, queue length can determine the congestion in a network by its depth. In this setup, I have monitored logical queue of size of 1MB with ECN threshold of 20KB. Two experiments, one with ECN marking and another without ECN marking in packets were performed. Figure 14 shows that marking packets with ECN indicating congestion in a network can maintain low queue length.

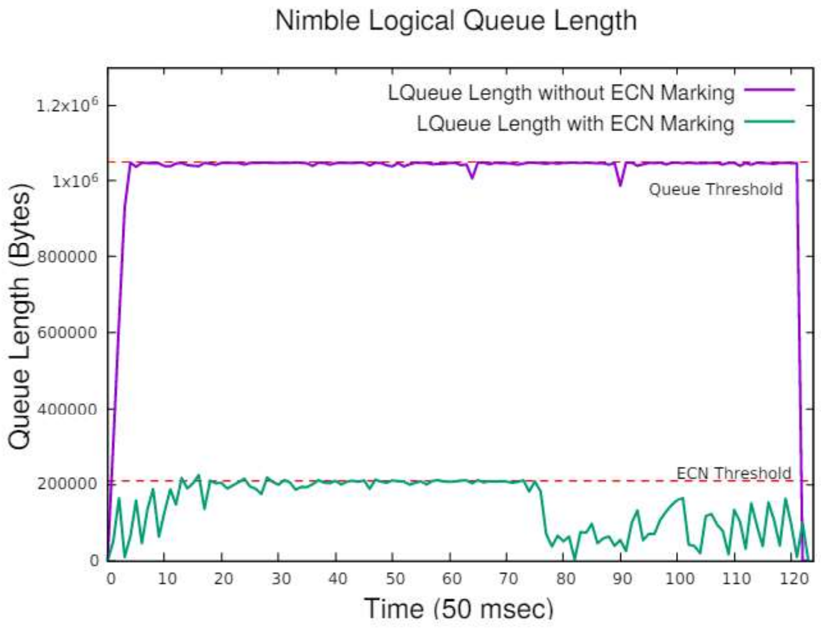
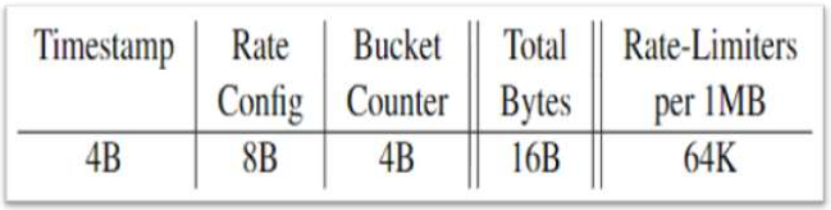


Figure 14: Nimble Logical Queue Analysis

4.3.3 Scalability

Figure 15 presents an analysis of the state overheads of implementing rate-limiters in Nimble. There are three contributors to the state overheads of Nimble: the timestamp that is stored in the Routing/Classification stage, the rate that is configured in the Rate Lookup, and the



Timestamp	Rate Config	Bucket Counter	Total Bytes	Rate-Limiters per 1MB
4B	8B	4B	16B	64K

Figure 15: Nimble Scalability

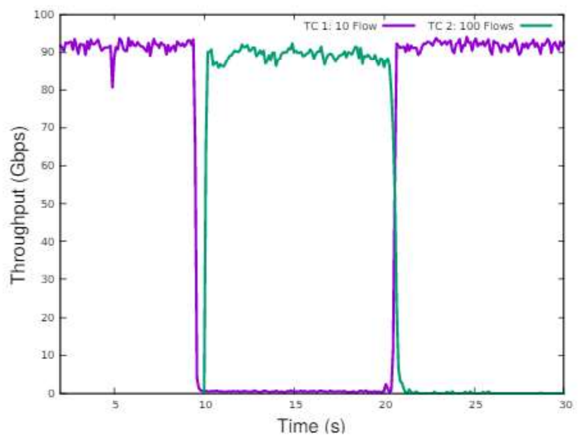
bucket that is stored in the Leaky Bucket stage. The timestamp is stored in a 4B register. The configured rate and bucket size are 4B, and the bucket counter is also 4B. This means that if Nimble were allocated 1MB of state, which is roughly 10% of the amount of state available on todays programmable switches, then it would be able to support 64K rate-limiters.

4.4 Application Level Benefits

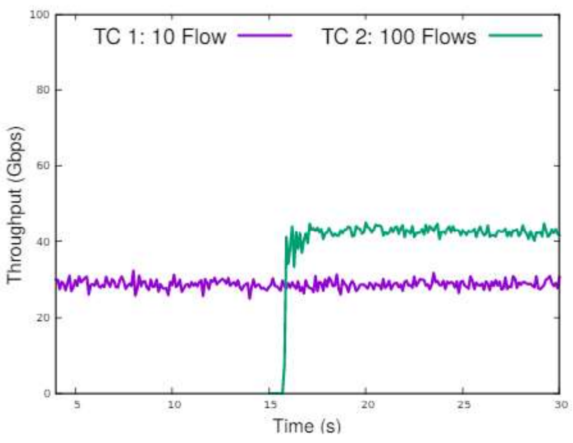
Different sets of experiments were performed to show the application level throughput and latency benefits from Nimble.

4.4.1 Application Level Throughput Benefits

TCP guarantees only per-flow fairness, independent of the numbers of flows that each tenant generates. Because, TCP can only guarantee per-flow and not per-tenant fairness, tenants can game the system by initiating more flows. In this section, I will demonstrate with the experiments that TCP on its own provides bad isolation when bandwidth-hungry application with multiple flows compete with genuine application sending traffic at allocated speed.



(a) Without Nimble



(b) With Nimble

Figure 16: DCTCP Flows: Per-Tenant Isolation

Figure 16(a) shows how TCP fails to provide guaranteed per-tenant isolation. When two tenants, namely TC1, and TC2, are competing with each other and since TCP guarantees per-flow fairness, TC2 (malicious tenant) with 100 flows gets more allocated bandwidth than TC1 with 10 flow. This proves that the performance of a genuine tenant (TC1) can easily get impacted if other tenant sends more flows.

To demonstrate per-tenant isolation in Nimble, a series of flows is generated which belongs to both TC1 and TC2. TC1 starts sending 10 flows traffic from $0s$, then at $15s$, after which TC2 starts sending 100 flows. Nimble has proved to achieve per-tenant isolation, independent of numbers of flows from each tenant. Figure16(b) shows even when traffic class 2 (TC2) sends traffic more than allocated to them, it will be rate limited to 50Gbps in figure16(b)) throughput limit pre-computed by Nimble rate limiters without impacting each others performance.

4.4.2 Application Level Latency Benefits

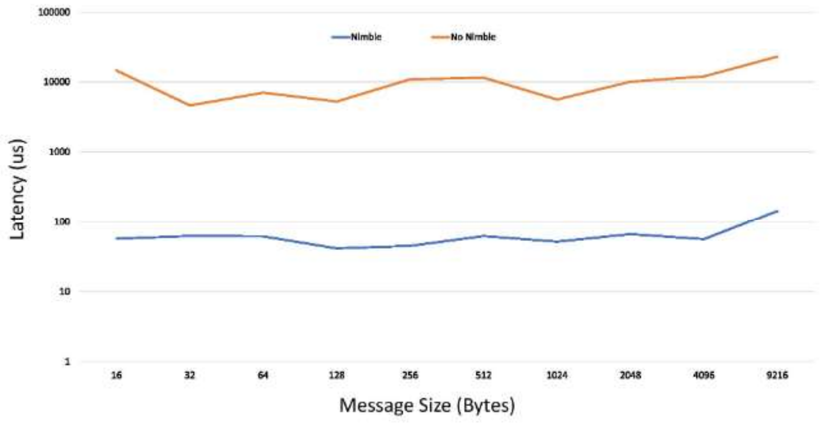


Figure 17: TCP Latency

Latency is an important metric that determines the system behavior. In the setup, I have used sockperf [45] to run between client and server to measure TCP RTT. Traffic from two tenants sharing the same output port is analyzed for latency. There are two sets of experiments: one with DCTCP without Nimble, and other with DCTCP with Nimble implementation.

In the first set of experiments, TC1 traffic with 100 flows is sent to destination port using iperf, while on the other hand, TC2 running sockperf test on client machine is destined to same output port.

In the second set of experiments, TC1 traffic with 100 flows is sent to destination port using iperf in the Nimble rate limiting implementation using logical queue. While, on other hand, sockperf test on TC2 measures the TCP RTT destined to same output port. Figure 17 demonstrates the result from the sockperf test. We can draw the conclusion that rate limiting traffic using logical queue has proved to reduce the latency to about $\sim 60us$ compared to without rate-limiting which is in tens of thousands.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, I present Nimble, a scalable system to perform rate-limiting in high-speed programmable switches. In this research, it was observed that TCP fails to provide isolation in a network. This impacts the performance of competing tenants or applications. Later, demonstration through experiments showed that rate limiters can be utilized for performance isolation. In a multi-tenant network, traffic sharing introduces high latency for competing traffic. With experiments, it was shown that Nimble can achieve its goal of high throughput and low latency. This thesis introduces a fundamentally new approach to implementing rate-limiting on a programmable switch. This thesis demonstrated the usefulness of registers and meters, already available as switch resources and utilize for network isolation without any hardware costs. Nimble implements performance isolation policy by dropping or marking packets with ECN bits indicating congestion. From the analysis on Bmv2 [42], and Barefoot switch [37], it Nimble is expected to provide per-tenant isolation in a network. It proves that Nimble can be scalable and support around 64K rate limiters provided 1MB SRAM. Nimble has achieved precision with maximum error in table entries to be less than 1%. Overall, Nimble provides application level benefits. Below are some key takeaways from this thesis:

- Nimble is able to scale up to 64K rate limiters given 1MB SRAM. It approximates the behavior of virtual queues with fewer resources. Nimble provides scalability in the network by implementing in-switch rate-limiters without requiring any queuing or scheduling resources.

- Nimble provides performance isolation with high throughput. All the ports in a pipeline can forward minimum sized packets in a switch.
- Nimble reduces dyanamic rate limiter updates from data plane.
- Experiments performed showed the accuracy of rate enforcement in the current prototype of Nimble. Nimble can enforce rate limits with less than 1% error across a range of rates measured from 1Gbps to 100Gbps.
- Nimble introduces a novel TCP friendly rate-limiting mechanism that uses ECN marking. This allows Nimble to be more TCP friendly than any existing approaches for in-network rate-limiting.

While Nimble resolves the performnace isolation problem, there are still substantial advances to be made in future work. I plan on implementing more complex policies such as weighted fairness, strict prioritization, and priority-based queuing. Next, I would also like to investigate different congestion control algorithms like RCP and PERC and its behavior with logical queues and meters. Additionally, as a part of future work on this project, there is a need to perform testbed experiments with multiple traffic classes and end hosts.

CITED LITERATURE

1. Radhakrishnan, S., Geng, Y., Jeyakumar, V., Kabbani, A., Porter, G., and Vahdat, A.: Senic: Scalable nic for end-host rate limiting. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* , NSDI14, page 475488, USA, 2014. USENIX Association.

2. Saeed, A., Dukkipati, N., Valancius, V., The Lam, V., Contavalli, C., and Vahdat, A.: Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* , SIGCOMM 17, page 404417, New York, NY, USA, 2017. Association for Computing Machinery.

3. Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A.: Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM* , pages 183–197, 2015.

4. Hong, C.-Y., Mandal, S., Al-Fares, M., Zhu, M., Alimi, R., B., K. N., Bhagat, C., Jain, S., Kaimal, J., Liang, S., Mendelev, K., Padgett, S., Rabe, F., Ray, S., Tewari, M., Tierney, M., Zahn, M., Zolla, J., Ong, J., and Vahdat, A.: B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in googles software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* , SIGCOMM 18, page 7487, New York, NY, USA, 2018. Association for Computing Machinery.

5. Jang, K., Sherry, J., Ballani, H., and Moncaster, T.: Silo: Predictable message latency in the cloud. In *SIGCOMM* , 2015.

6. Jeyakumar, V., Alizadeh, M., Mazieres, D., Prabhakar, B., Kim, C., and Greenberg, A.: EyeQ: Practical network performance isolation at the edge. In *NSDI* , 2013.

7. Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I.: Faircloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* , SIGCOMM 12, page 187198, New York, NY, USA, 2012. Association for Computing Machinery.

8. Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A.: Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* , SIGCOMM 11, page 242253, New York, NY, USA, 2011. Association for Computing Machinery.

9. Kumar, A., Jain, S., Naik, U., Kasinadhuni, N., Zermeno, E. C., Gunn, C. S., Ai, J., Carlin, B., Amarandei-Stavila, M., Robin, M., Siganporia, A., Stuart, S., and Vahdat, A.: BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM* , 2015.

10. Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C.: Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* , pages 450 462, 2015.

11. Shieh, A., Kandula, S., Greenberg, A., Kim, C., and Saha, B.: Sharing the data center network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* , NSDI11, page 309322, USA, 2011. USENIX Association.

12. Saeed, A., Zhao, Y., Dukkipati, N., Zegura, E., Ammar, M., Harras, K., and Vahdat, A.: Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* , pages 17 32, Boston, MA, February 2019. USENIX Association.

13. Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M.: Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* , NSDI12, page 19, USA, 2012. USENIX Association.

14. Brent Stephens, A. A. and Swift, M. M.: Loom: Flexible and efficient nic packet scheduling. In *NSDI* , 2019.

15. Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M.: Congestion control for large-scale RDMA deployments. In *SIGCOMM* . ACM, August 2015.

16. Mittal, R., Lam, T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D.: TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM* , 2015.

17. Dalton, M., Schultz, D., Adriaens, J., Arefin, A., Gupta, A., Fahs, B., Rubinstein, D., Zermeno, E. C., Rubow, E., Docauer, J. A., Alpert, J., Ai, J., Olson, J., DeCabooter,

K., De Kruijf, M., Hua, N., Lewis, N., Kasinadhuni, N., Crepaldi, R., Krishnan, S., Venkata, S., Richter, Y., Naik, U., and Vahdat, A.: Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* , NSDI18, page 373387, USA, 2018. USENIX Association.

18. Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M.: Data Center TCP (DCTCP). In *SIGCOMM* , 2010.

19. Yu, M., Greenberg, A., Maltz, D., Rexford, J., Yuan, L., Kandula, S., and Kim, C.: Profiling network performance for multi-tier data center applications. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* , NSDI11, page 5770, USA, 2011. USENIX Association.

20. Barroso, L., Marty, M., Patterson, D., and Ranganathan, P.: Attack of the killer microseconds. *Commun. ACM* , 60(4):4854, March 2017.

21. Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., Chandrappa, H. K., Chaturmohta, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D. A., and Greenberg, A.: Azure accelerated networking: Smartnics in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* , NSDI18, page 5164, USA, 2018. USENIX Association.

22. Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A.: High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference* , IMC 17, page 7885, New York, NY, USA, 2017. Association for Computing Machinery.

23. Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S. P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R., and Katz-Bassett, E.: Sdx: A software defined internet exchange. In *Proceedings of the 2014 ACM Conference on SIGCOMM* , SIGCOMM 14, page 551562, New York, NY, USA, 2014. Association for Computing Machinery.

24. Cisco. comparing traffic policing and traffic shaping for bandwidth limiting.

25. Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R.: Detail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM*

2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication , SIGCOMM 12, page 139150, New York, NY, USA, 2012. Association for Computing Machinery.

26. Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M.: Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* , SIGCOMM 13, page 99110, New York, NY, USA, 2013. Association for Computing Machinery.

27. Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S.: Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* , SIGCOMM 16, page 1528, New York, NY, USA, 2016. Association for Computing Machinery.

28. Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R.: Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* , SIGCOMM 13, page 1526, New York, NY, USA, 2013. Association for Computing Machinery.

29. Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N.: Programmable packet scheduling at line rate. In *SIGCOMM* , 2016.

30. Data Center Bridging Task Group: `http://www.ieee802.org/1/pages/dcbridges.html`.

31. Floyd, S. and Jacobson, V.: Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking* , 1(4):397–413, 1993.

32. Ramakrishnan, K., Floyd, S., and Black, D.: Rfc3168: The addition of explicit congestion notification (ecn) to ip. Technical report, USA, 2001.

33. Zhu, Y., Ghobadi, M., Misra, V., and Padhye, J.: Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* , CoNEXT 16, page 313327, New York, NY, USA, 2016. Association for Computing Machinery.

34. Shin, M., Nam, K., and Kim, H.: Software-defined networking (sdn): A reference architecture and open apis. In *2012 International Conference on ICT Convergence (ICTC)* , pages 360–361, 2012.

35. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J.: Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* , 38(2):6974, March 2008.

36. Budiu, M. and Dodd, C.: The p416 programming language. *SIGOPS Oper. Syst. Rev.* , 51(1):514, September 2017.

37. Tofino, B.: Worlds fastest p4-programmable ethernet switch asics, 2018.

38. Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A.: B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* , SIGCOMM 13, page 314, New York, NY, USA, 2013. Association for Computing Machinery.

39. Goyal, P., Shah, P., Sharma, N. K., Alizadeh, M., and Anderson, T. E.: Backpressure flow control. *Proceedings of the 2019 Workshop on Buffer Sizing* , Dec 2019.

40. Stephens, B., Cox, A. L., Singla, A., Carter, J., Dixon, C., and Felter, W.: Practical DCB for improved data center networks. In *INFOCOM* , 2014.

41. Shrivastav, V.: Fast, scalable, and programmable packet scheduler in hardware. In *SIGCOMM* . ACM, 2019.

42. p4lang/behavioral-model. `https://github.com/p4lang/behavioral-model`.

43. Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N.: Reproducible network experiments using container-based emulation. In *CoNEXT* , 2012.

44. iperf3: Documentation. `http://software.es.net/iperf/`.

45. Sockperf: A network benchmarking utility over socket api.

VITA

NAME:	KOMAL SHINDE
EDUCATION:	M.S., Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, Illinois, 2020.
	B.Tech., Electronics and Telecommunication, Dr. Babasaheb Ambedkar Technological University, Raigad, India, 2016.
ACADEMIC EXPERIENCE:	Graduate Research Assistant, University of Illinois at Chicago, 2018 - 2020.