MITIGATING THE BUFFERBLOAT PROBLEM TO REDUCE INTERNET TRANSPORT LATENCY

Thesis

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Sachin Dattatraya Patil

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA,

SURATHKAL, MANGALORE - 575025

June 2020

DECLARATION

by the Ph.D. Research Scholar

I hereby **declare** that the Research Thesis entitled **Mitigating The Bufferbloat Problem To Reduce Internet Transport Latency** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfilment of the requirements for the award of the Degree of **Doctor of Philosophy** in **Computer Science and Engineering** is a **bonafide report of the research work carried out by me**. The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.

(**138039 CS13F07, Sachin Dattatraya Patil**)
(Register Number, Name & Signature of Research Scholar)
Department of Computer Science and Engineering

Place: NITK, Surathkal.
Date: June 9, 2020

CERTIFICATE

This is to *certify* that the Research Thesis entitled **Mitigating The Bufferbloat Problem To Reduce Internet Transport Latency** submitted by **Sachin Dattatraya Patil**, (Register Number: **138039 CS13F07**) as the record of the research work carried out by him, is *accepted as the Research Thesis submission* in partial fulfillment of the requirements for the award of degree of **Doctor of Philosophy**.

Dr. Mohit P. Tahiliani
Research Supervisor
(Name and Signature with Date and Seal)

Chairman - DRPC
(Signature with Date and Seal)

Acknowledgements

Firstly, I would like to express my sincere gratitude to my guide Dr. Mohit P. Tahiliani (Assistant Professor, Department of CSE) for his continuous guidance and motivation, because of which I have been able to learn in handling the issues in terms of technical, personal and professional career during the journey of my PhD at NITK, Surathkal.

Besides my guide I would like to thank Prof K. Chandrasekharan (CSE Department and my RPAC member) and Prof. K. V. Gangadharan (Mechanical Department and another my RPAC member), who have been a continuous source of motivation and encouragement with their constructive suggestions to assist me during my PhD journey.

My sincere thanks also goes to Dr. Alwyn Roshan Pais (HoD CSE and chairman, DRPC) who has been supported me all the time of my PhD.

Place: Surathkal **Sachin Dattatraya Patil**

Date: June 9, 2020

Abstract

There has been a proliferation of high capacity routers on the Internet to appropriately utilize the high-speed data links. These routers typically consist of very large buffers because the memory prices have fallen sharply. Due to surplus buffering, packets experience an excessive queuing delay which leads to a significant degradation in the overall performance and largely reduces the quality of service for time-critical applications. The above-mentioned problem is known as *bufferbloat*. Active Queue Management (AQM) is a promising technique to minimize the impact of Bufferbloat and improve the quality of service for time-critical applications. Several AQM algorithms have been designed to monitor and limit the growth of the queue at routers. Controlled Delay (CoDel) and Proportional Integral controller Enhanced (PIE) are two popular AQM algorithms which are designed to address the problem of bufferbloat.

One of the vital characteristics of AQM algorithms is to maintain a proper trade-off between queue delay and bottleneck link utilization. However, maintaining this trade-off becomes challenging when unresponsive flows crossing the router do not respond to congestion notifications, e.g., congestion agnostic UDP flows. Unresponsive flows increase queue delay, affect the queue stability and lead to more packet losses. Additionally, AQM algorithms do not provide adequate fairness when responsive flows and unresponsive flows share the same bottleneck link. Unresponsive flows tend to dominate the bandwidth consumption due to lack of congestion control. This leads to fairness problems and subsequently, the performance of responsive flows degrades significantly. One of the potential approaches to resolve the problem of unfairness between responsive and unresponsive flows is to provide flow protection by integrating AQM algorithms with packet scheduling algorithms, such as Deficit Round Robin (DRR).

The main goal of this work is to enhance the robustness of AQM algorithms against unresponsive traffic and provide fairness when responsive and unresponsive flows coexist. Along these lines, this thesis makes the following contributions: we propose Modified

CoDel and Minstrel PIE as enhancements to CoDel and PIE, respectively to increase their robustness against unresponsive flows. Subsequently, we propose Flow Queue Minstrel PIE (FQ-Minstrel PIE) to address the concern of fairness among responsive and unresponsive flows. Besides these primary contributions, we have developed a fluid model to obtain an in-depth understanding of working of CoDel and Modified CoDel, aligned the implementation of PIE in Linux kernel to RFC 8033 and implemented a new model for Flow Queue PIE (FQ-PIE) in the Linux kernel.

Extensive evaluations conducted through mathematical modeling, simulations and real-time experiments show that Modified CoDel achieves better performance against unresponsive flows. However, we note that Modified CoDel is not scalable and fails to perform due to inherent limitations in the design of CoDel. Conversely, it is observed that Minstrel PIE, a minor enhancement of PIE, offers significant performance improvements against unresponsive traffic, and FQ-Minstrel PIE resolves the fairness problem between responsive and unresponsive flows. The work on aligning the PIE implementation in Linux kernel with RFC 8033 is merged in the mainline of Linux kernel since v5.1 and the FQ-PIE algorithm is merged in the mainline of Linux kernel since v5.5.

Keywords: Bufferbloat, Active Queue Management, CoDel, PIE.

Table of Contents

List of Figures

List of Tables

Abbreviations and Nomenclature

Abbreviations

AIMD	Additive Increase Multiplicatve Decrease
AQM	Active Queue Management
ARED	Adaptive Random Early Detection
BDP	Bandwidth Delay Product
CDF	Cumulative Distribution Function
COBALT	COntrolled Delay and Blue ALTernate
CoDel	Controlled Delay
DRR	Deficit Round Robin
ECN	Explicit Congestion Notification
EWMA	Exponential Weighted Moving Average
FQ	Flow Queue
FQ-CoDel	Flow Queue CoDel
FQ-PIE	Flow Queue Proportional Integral controller Enhanced
Flent	Flexible Network Tester
Flent	Flexible network tester
IP	Internet Protocol

MTU	Maximum Transmission Unit
PDF	Probability Distribution Function
PI	Proportional Integral controller
PIE	Proportional Integral controller Enhanced
QUIC	Quick User Datagram Protocol Internet Connection
RED	Random Early Detection
REM	Random Exponential Marking
RFC	Request For Comments
RTT	Round Trip Time
SFB	Stochastic Fair BLUE
SFQ	Stochastic Fair Queuing
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
VoIP	Voice over Internet Protocol

Nomenclature

avg_dq_rate	average dequeue rate
cur_delay	current delay
interval	interval period
maxavg_dq_rate	maximum average dequeue rate
ns-2	Network Simulator 2
ns-3	Network Simulator 3
old_delay	previous inerval delay
p	packet drop probability
p	drop probability
pdrop	drop probability in BLUE algorithm
qdelay_ref	reference queue delay
qdisc	queue discipline
target	target queue delay
tupdate	probability update period
α	constant used in PIE algorithm
β	constant used in PIE algorithm
PI^2	Proportional Integral controller Square
avg	average queue length in ARED
max_thr	maximum threshold constant in ARED
min_thr	minimum threshold constant in ARED
$p_{(i)}$	drop probability in the currrent interval

$p_{(i-1)}$	drop probability in the previous interval

Chapter 1

Introduction

Internet has been experiencing a rapid growth in terms of penetration rate and diversity of applications that range from huge file transfers and online gaming to remote health monitoring. It has transformed from a communication system into a massive and decentralized source of information. According to 2018 study[1], the average Internet speed in wired networks is 46.12 Mbps and 22.44 Mbps for download and upload, respectively, whereas the same for cellular networks is 22.82 Mbps and 9.19 Mbps for download and upload, respectively. Although the Internet usage is rapidly increasing and enhancing the connectivity worldwide, new applications have placed stringent performance requirements. The primary needs of today's Internet applications are low latency and high throughput. High throughput can be achieved by provisioning more bandwidth, but reducing latency is a significant challenge.

The present day Internet applications can be classified into three categories (Kennedy et al., 2017): (i) *latency sensitive applications* such as web browsing, instant messaging, remote monitoring, online gaming, Voice over IP (VoIP), video conferencing, DNS queries, IoT applications and others, (ii) *latency tolerant or elastic applications* such as photo and video sharing, emails with large attachments, peer to peer file transfer, offsite backup systems, updates for operating systems and mobile apps, downloading movies and TV shows, offline playback, messaging with multimedia attachments and (iii) *streaming content traffic* such as Netflix, Amazon prime videos, Youtube and others. Table 1.1 presents different types of Internet traffic separated into three latency categories with their tolerance time (Boerlage and Collom, 2016).

The performance of latency sensitive applications becomes worse when routers have

[1] https://www.speedtest.net/insights/blog/2018-internet-speeds-global/

Table 1.1: Latency tolerance limits for different Internet applications

Applications	**Latency Category**	**Latency Tolerance**
VoIP or video	Sensitive	0 - 150 ms
Gaming applications	Sensitive	0 - 200 ms
Web browsing	Sensitive	1 - 2 s
Message applications	Elastic	5 - 10 s
Bulk downloads	Elastic	>10 s
Cloud uploads (e.g., photo)	Elastic	>10 s
Media streaming	Streaming	5 - 10 s

large buffering capacity and the default queuing mechanism is passive (e.g., *droptail*). (Gettys and Nichols, 2011). Typically, the minimum buffer size required for fully utilizing the bottleneck link to a particular bandwidth scenario is equal to the *Bandwidth Delay Product* ($BDP = \frac{B * RTT}{\sqrt{N}}$) where B is bottleneck bandwidth, RTT is average Round Trip Time of a flow crossing through a router and N is the number of flows sharing the bottleneck router (Appenzeller et al., 2004)). However, manufacturers tend to provide high buffering capacity in routers due to lesser memory prices. *Droptail* with large buffer capacity has the following drawbacks:

- **Global synchronization:** Routers that employ Passive Queue Management (PQM), like droptail do not drop packets before the queue fills up. This leads to packets getting dropped from all the TCP flows traversing through that router. Consequently, all senders reduce their congestion window (*cwnd*) at the same time, and subsequently, tend to gradually increase the *cwnd*. Thus, this phenomenon is called *global synchronization* (Floyd and Jacobson, 1993). Global synchronization results in alternate epochs of overutilization and underutilization of the bottleneck link capacity, and it also adds jitter due to large oscillations in the queue occupancy.

- **Lock out:** PQM allows early comer flows to dominate the buffer space at the router. This results in an unfair sharing of network resources among flows (Hassan and Jain, 2003).

- **Bufferbloat:** Since memory costs reduced in the past, modern Internet routers are designed with extremely large buffers. As a result, today's Internet suffers from poor

network performance because TCP congestion control mechanisms implemented in modern operating systems follow an end-to-end approach and hence do not reduce the sending rate unless a packet drop is encountered. Since the packet drop occurs only when these large buffers overflow, queuing delay experienced by each packet increases significantly, consequently the Quality of Service (QoS) for latency sensitive applications. This problem is known as *bufferbloat*. (Gettys and Nichols, 2011).

Active Queue Management (AQM) mechanisms have been extensively studied to monitor and limit the growth of the queue at routers. Controlling queue length/delay at routers plays a vital role to tackle the problems that occur with droptail. In this regard, several AQM mechanisms have been designed: Random Early Detection (RED) (Floyd and Jacobson, 1993), Random Exponential Marking (REM) (Athuraliya et al., 2001), Adaptive RED (ARED) (Floyd et al., 2001), BLUE (Feng et al., 2002), Proportional Integral controller (PI) (Hollot et al., 2001), Controlled Delay (CoDel) (Nichols and Jacobson, 2012), Proportional Integral controller Enhanced (PIE) (Pan et al., 2013) and PI^2 (De Schepper et al., 2016) are some of the well-known AQM mechanisms.

The classification of AQM mechanisms is often based on congestion indicator(s) such as: queue length (e.g., RED, PI, etc), input rate (e.g., BLUE, etc), queue delay (e.g., CoDel, PIE, etc) and the combination of input rate with queue length/delay. The ones that use similar congestion indicator(s) differ in a way they arrive at a decision to enqueue/drop packets e.g., in ARED, the packet drop probability to enqueue/drop packets is a function of an *average queue length* whereas PI measures the packet drop probability on the basis of *instantaneous queue length*. Figure 1.1 depicts the classification of AQM mechanisms based on their congestion indicators.

This work focuses on the queuing mechanisms which are popularly known for solving the bufferbloat problem, namely ARED, CoDel and PIE (Kuhn et al., 2014; Järvinen and Kojo, 2014; Kuhn et al., 2017).

- **ARED:** ARED is one of the oldest and most popular AQM mechanisms which is a variant of RED mechanism (Floyd and Jacobson, 1993; Floyd et al., 2001). Although ARED predates bufferbloat, studies have sown that it is a promising mechanism to address the bufferbloat problem (Kuhn et al., 2014; Khademi et al., 2013). ARED works on the basis of *average queue length* which uses a low pass filter technique for averaging. Depending on the *average queue length*, it calculates the drop probability to decide to enqueue/drop the incoming packets.

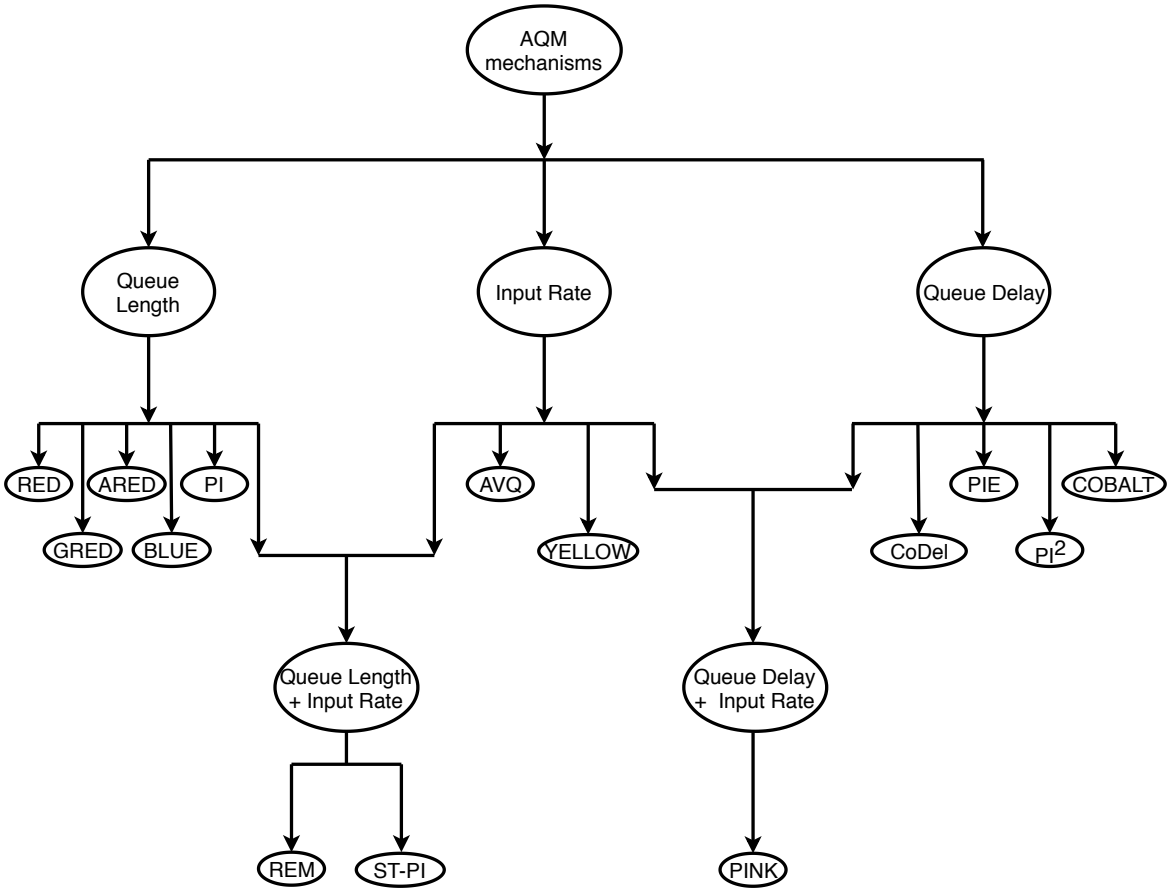


Figure 1.1: Classification of AQM mechanisms

- **CoDel:** CoDel is designed to solve the problem of bufferbloat (Nichols and Jacobson, 2012). It is easy to implement and requires setting of two parameters namely *target* and *interval*. During dequeuing, CoDel decides whether to drop/dequeue a packet based on the time spent by that packet in the queue (sojourn time).

- **PIE:** PIE is a lightweight controller which is designed to solve the problem of bufferbloat (Pan et al., 2013). PIE combines the benefits of both CoDel and RED in PI controller (Hollot et al., 2001). It detects the congestion on the basis of queuing delay like CoDel, and updates the drop probability during enqueue time like RED. It efficiently controls the queuing delay around the *reference queue delay*.

1.1 The problem

Although many AQM mechanisms are available in the relevant literature to solve the bufferbloat problem, they are not thoroughly tested and evaluated against the unresponsive flows. This section highlights the performance problems that arise due to unresponsive

traffic.

1.1.1 Issues with unresponsive traffic

One of the vital characteristics of AQM mechanisms is to maintain a proper trade-off between queue delay and bottleneck link utilization. But maintaining this trade-off is challenging when some of the flows crossing the router do not respond to the congestion feedback provided by AQM mechanisms, e.g., congestion agnostic UDP flows do not respond to congestion notifications. The following are the major concerns in such cases:

- Tackling the problem of bufferbloat becomes challenging because unresponsive flows affect the stability of the queue.
- Unresponsive behavior of congestion agnostic UDP flows defeats the purpose of deploying AQM mechanisms and leads to excessive queuing delays.
- The problem further aggravates when unresponsive flows arrive in bursts, leading to variations in RTT. The burst may also lead to bulk packet losses due to queue overflow.
- The network resource allocation is unfair because TCP flows respond to congestion signals whereas unresponsive flows do not. Hence, the major share of the bandwidth gets utilized by unresponsive flows, leaving a negligible room for TCP flows.

Addressing these concerns is becoming significant due to the widespread adoption of UDP as a primary transport protocol. Popular applications of UDP include: DNS, routing updates, SNMP and NAT traversal techniques such as STUN (RFC 5389) and TURN (RFC 5766). Emerging applications of UDP include: video streaming, online gaming, peer to peer applications, Quick UDP Internet Connection (QUIC) (Langley et al., 2017) and browser frameworks such as Web Real-Time Communication (WebRTC)[2].

Although the emerging applications of UDP are obliged to implement congestion control at the application level and respond to congestion (e.g., QUIC and WebRTC), a significant share of the UDP traffic comprises short-lived unresponsive flows. For example, the recent analysis of 4G LTE traffic from a Tier-1 wireless carrier in the south-central US reveals that DNS dominates UDP traffic in terms of the number of flows (39% of the

[2]WebRTC is a framework of protocols and JavaScript APIs that enables peer-to-peer audio, video, and data sharing between browsers. It is supported by popular browsers such as Google Chrome and Mozilla Firefox, and it uses UDP for data transfers (Bergkvist et al., 2012).

total flows captured in the network) (Li et al., 2018). Despite the fact that DNS traffic is thin in terms of volume and might not lead to long-lasting congestion, studies have shown that such a small fraction of unresponsive UDP flows can affect the queue stability and deteriorate the performance of AQM mechanisms (Hollot et al., 2003; Li et al., 2005; Cai et al., 2010). Consequently, some of the researchers have proposed AQM mechanisms with dual functionality (e.g., COBALT uses CoDel and BLUE (Morton, 2016)) to control the queue delay against unresponsive flows. However, these mechanisms do not guarantee the fairness among the responsive and unresponsive flows. Through extensive evaluations by means of simulation and real time experiments, this works confirms that controlling queue occupancy is a non-trivial task when responsive and unresponsive flows coexist.

1.1.2 Fairness issues

Recently, there has been more emphasis on using multiple logical queues at a router for mixed traffic to avoid the starvation of TCP flows against unresponsive UDP flows. To accommodate this, many new hybrid AQM mechanisms are developed by researchers with the help of Flow Queuing (FQ) schedulers that include: Stochastic Fair Queuing-CoDel (SFQ-CoDel) (Jain et al., 2014), Flow Queue-CoDel (FQ-CoDel RFC 8290), Flow Queue-PIE (FQ-PIE) (Al-Saadi and Armitage, 2016), etc. The results analysis of FQ-PIE and FQ-Minstrel PIE presented in this work proves that the adequate fairness can be achieved when responsive and unresponsive flows coexist.

1.2 Contributions of this thesis

1.2.1 Primary contributions

A Modified CoDel

The performance of CoDel is largely affected due to a fixed parameter called *interval* (100 ms), and the conservative adaptation of packet drop rate based on the square root law. To overcome these limitations, it is proposed to reduce the interval value to 30 ms based on the experimental observations and modify the control law of CoDel with harmonic series instead square root law to adapt the packet drop rate quickly. This work proposes both these modifications while considering the fact that the performance of CoDel should remain unaffected for responsive flows.

B Minstrel PIE

This study proposes a variant of the PIE mechanism called Minstrel PIE to increase its robustness when unresponsive flows coexist with TCP flows. It was decided to base the work on PIE mechanism because: (i) it is a defacto AQM used by CableLabs since DOCSIS 3.1 cable modems (RFC 8034), and (ii) studies have shown that PIE is likely to provide better control on queue delay than other mechanisms in the case of extreme overload (Järvinen and Kojo, 2014; Kuhn et al., 2017).

Minstrel PIE adapts itself and timely drops packets to control the queue delay when the traffic load increases, otherwise operates similar to PIE. Minstrel PIE regulates the reference queue delay ($qdelay\_ref$) parameter of PIE to adjust the drop probability in Minstrel PIE. Despite gaining significant attention, the implications of keeping $qdelay\_ref$ fixed in PIE are not widely studied. This work highlights the need to adapt $qdelay\_ref$ in PIE.

C FQ-Minstrel PIE

PIE and Minstrel PIE being AQM mechanisms do not provide flow protection. Combining these mechanisms with flow queuing mechanism is a promising approach to solve the problem of fairness. This work provides the insights of working of FQ-PIE and FQ-Minstrel PIE. Subsequently, this work also evaluates the FQ-Minstrel PIE by comparing its performance with PIE, FQ-PIE and FQ-CoDel.

1.2.2 Secondary contributions

A Fluid model for CoDel

The fluid models for ARED and PIE are exists but not for CoDel[3]. ARED and PIE work on the basis of packet drop probability, hence, the model proposed in (Misra et al., 2000) can be used directly with ARED and PIE. However, CoDel works on the basis of deterministic drop strategy, thus, there is to modify the existing fluid model to work with CoDel. To overcome this limitation, we designed the fluid model for CoDel. This work has been verified by comparing the results obtained with to those obtained from ns-2.

[3] According to 2017 survey, when we proposed the fluid model for CoDel

B PIE alignment with RFC 8033

Internet Engineering Task Force (IETF) has described the working of PIE in RFC 8033 for deployment in the Internet. Although Linux kernel has a PIE queue discipline (qdisc), its working is not same as described in RFC 8033 because the implementation of PIE qdisc in Linux preceded the publication of RFC 8033. We align Linux PIE qdisc with RFC 8033, and evaluated in the real time testbed. The evaluation shows that certain features recommended in RFC 8033 offer significant benefits, whereas others are not worth adopting in real systems. Consequently, only a few features are considered from RFC 8033 to be implemented in the Linux PIE qdisc. The proposed modifications are merged and published in the mainline of Linux kernel v5.1.

C Design and implementation of FQ-PIE

FQ-CoDel (RFC 8290) and FQ-PIE (Al-Saadi and Armitage, 2016) are two attempts towards combining Flow queuing with AQM mechanisms. Although FQ-CoDel is available in the latest versions of the Linux kernel, an implementation of FQ-PIE is missing. The present study therefore implemented FQ-PIE because it forms the basis for FQ-Minstrel PIE. Moreover, the researcher believed that adding the support of FQ-PIE in Linux kernel would be useful because in our study it was shown that PIE, even without FQ, was likely to provide better control on queue delay than CoDel when unresponsive flows existed. This implementation is currently in the Linux kernel since v5.6.

1.3 Outline of the thesis

Chapter 2 discusses the relevant literature of all the popular AQM mechanisms which are specifically designed to solve the problem of bufferbloat. Further, the open issues existing in AQM mechanisms due to which they are not able to perform well in some of the network scenarios are discussed. Specifically it highlights the problem of AQM mechanisms against unresponsive flows and provides the background of flow queuing.

Chapter 3 first discusses the design of Modified CoDel. Subsequently, it presents the design of the mathematical model for CoDel and verifies the implementation by comparing it with ns-2 implementation of CoDel. The analysis of CoDel and Modified CoDel in fluid model and real time testbed is presented in detail. Next, it highlights the limitations of CoDel in terms of scalability and why the focus is shifted to PIE.

Chapter 4 presents the design of Minstrel PIE to improve the performance of PIE against unresponsive flows. Originally, PIE has a better control against unresponsive flows than CoDel mechanism in terms of controlling queuing delay. In addition to this, Minstrel PIE gives optimized trade-off between bottleneck link utilization and queue delay than PIE against unresponsive flows. The results show that Minstrel PIE reduces the queue delay significantly against unresponsive flows without affecting link utilization.

The prerequisite to implement FQ-Minstrel PIE is FQ-PIE. But FQ-PIE is not available in the Linux kernel. Chapter 5 discusses about contributions made to the Linux kernel by implementing FQ-PIE. Later, the analysis of FQ-PIE, FQ-CoDel and FQ-Minstrel PIE is discussed in this chapter. The results show that FQ-Minstrel PIE achieves same fairness with reduced queue delay than FQ-PIE and FQ-CoDel.

Chapter 6, summarizes the contribution made in this thesis and the potential limitations of the work. The directions for future work are also discussed in this chapter.

Chapter 2

Literature Review

Although high-speed routers and switches are being used with high bandwidth links to improve the overall network performance, latency continues to remain a big issue in today's Internet. It is believed that users tend to get frustrated when they perceive a delay of 300 milliseconds (Grigorik, 2013). There are four major components of delay: propagation delay, queuing delay, transmission delay and processing delay. Besides others, the major concern today is queuing delay due to bloated buffers. Recently, bufferbloat has received a major attention from the research community. Along with increased queuing delay, bufferbloat gives rise to problems such as variations in RTT and defeats the purpose of TCP's congestion control mechanisms (Groenewegen and Kleppe, 2011).

2.1 Background

RED (Floyd and Jacobson, 1993), REM (Athuraliya et al., 2001), ARED (Floyd et al., 2001), BLUE (Feng et al., 2002), PI (Hollot et al., 2001), etc are some of the widely studied AQM mechanisms. Recently, new AQM mechanisms have been designed to solve the problem of bufferbloat: Controlled Delay (CoDel) (Nichols and Jacobson, 2012) and Proportional Integral controller Enhanced (PIE) (Pan et al., 2013) are two popular ones.

AQM algorithms that predate bufferbloat do not use the *queue delay* as a congestion indicator. Queue delay has a direct implication on the user's perception of the application behavior. CoDel and PIE have been designed to tackle the problem of queue delay arising due to the bufferbloat problem, and thus, both use queue delay as an indicator of congestion. This work considers to build on the success of CoDel and PIE in terms of controlling queue delay. Despite the fact that ARED predates bufferbloat, some stud-

ies have shown that it has the potential to control queue delay within specified bound (Järvinen and Kojo, 2014; Kuhn et al., 2014, 2017). However, we do not consider it for further investigations because it depends on RED and suffers from the same limitations that are highlighted in Misra et al. (2000). Nevertheless, we provide background of RED and ARED in the next section for completeness.

2.1.1 RED and Adaptive RED

RED (Floyd and Jacobson, 1993) is one of the oldest and widely known AQM mechanisms. It has been extensively studied and is known to have sensitivity to parameter settings. Past work (Feng et al., 2001; Floyd et al., 2001; Misra et al., 2000) has shown that an incorrectly configured RED gateway can deteriorate network performance. Adaptive RED (ARED) (Floyd et al., 2001) addresses a few limitations of RED and eliminates the need to manually configure some of its parameters. This section describes the working of RED in brief, followed by ARED.

RED uses Exponential Weighted Moving Average (EWMA) to calculate the average queue length (avg) on arrival of every packet. If avg exceeds maximum threshold (max_{th}), the incoming packet is dropped and if it is lesser than minimum threshold (min_{th}), the packet is enqueued. Otherwise, if avg is in between max_{th} and min_{th}, then the packet is dropped/enqueued depending on the random drop probability (p). Eq. (2.1) and (2.2) show equations for avg and p, respectively.

$$avg = cur\_qlen \times w_q + old\_avg \times (1 - w_q) \tag{2.1}$$

$$p = \begin{cases} 1 & \text{if } avg \geq max_{th} \\ max_p \times \dfrac{avg - min_{th}}{max_{th} - min_{th}} & \text{if } max_{th} > avg \geq min_{th} \\ 0 & \text{if } avg < min_{th} \end{cases} \tag{2.2}$$

where

- old_avg is the average queue length during previous sample;
- cur_qlen is the current queue length;

- w_q is EWMA constant;
- max_p is the maximum drop probability;

Algorithm 1: RED mechanism

Initialization: *count = -1*
avg = 0

On every packet arrival
Calculate new average queue length
if *Queue is nonempty* **then**
  $avg = cur\_qlen \times w_q + old\_avg \times (1 - w_q)$
else
  $avg = (1 - w_q)^{\frac{cur\_time - old\_time}{s}} \times avg$
if $avg \geq max_{th}$ **then**
  Drop the packet
  $count = -1$
else if $max_{th} > avg \geq min_{th}$ **then**
  Increment *count*
  $p_b = max_p \times \dfrac{avg - min_{th}}{max_{th} - min_{th}}$
  $p_a = \dfrac{p_b}{1 - (count * p_b)}$
  Drop the packet with probability p_a
  $count = 0$
else
  $count = -1$
When Queue becomes empty
$old\_time = cur\_time$

Parameters
R = Random variable
s = packet transmission time
p_a = Actual packet drop probability

The detailed working of RED is shown in Algorithm 1. In Eq. (2.2) it can be noted that drop probability becomes 1 after avg crosses above max_{th}. This sharp increase in drop probability leads to aggressive dropping of incoming packets at the router and hence, affects the throughput. To resolve this issue, a new variant of RED was proposed by Sally Floyd, named Gentle RED (GRED). In GRED, when avg rises from max_{th} to $2 \times max_{th}$, GRED gradually increases drop probability from max_p to 1 so that large number of packets are not dropped.

RED's performance and effectiveness significantly depends on the appropriate setting of the four parameters (Feng et al., 1999). To avoid manually setting these parameters, Self Configuring RED (SCRED) (Feng et al., 1999) is proposed which adapt the max_p on

the basis of average queue length. The value of max_p is adapted in SCRED to keep the avg in between min_{th} and max_{th}.

ARED is in fact an extension to SCRED. ARED aims to maintain the avg within the $target$ range of min_{th} and max_{th}. Moreover, ARED follows the Additive Increase Multiplicative Decrease (AIMD) approach to adapt the value of max_p, so that it can handle the rise in the traffic load gradually.

Several other variants of RED exist in the literature e.g., Stabilized ARED (SARED) (Javam and Analoui, 2006), Self Tuning RED (STRED) (Chen et al., 2011), Refined ARED (Re-ARED) (Kim and Lee, 2006) are a few. Although these mechanisms offer performance improvements, they add new parameters which further complicates the design of RED and hinders its deployment in the real network.

Misra et al. (2000) presented an analysis of RED with the help of a fluid model. The authors show that the performance of RED largely depends on the appropriate setting of w_q. RED works on the basis of average queue length and for averaging, it uses the parameter w_q which affects its behavior against incipient congestion because w_q is dependent on the duration and size of the incoming burst at the queue. A large value of w_q suppresses the oscillations in the queue and shows the stability in the average queue length, and in turn, reduces jitter. These oscillations depend on many factors, such as packet size, bandwidth, and incoming traffic load. However, value of w_q cannot be too large, otherwise it affects the calculation of average queue length and leads to an increase in the initial spikes in the instantaneous queue length.

To overcome this problem, (Hollot et al., 2001) present a new AQM mechanism called Proportional Integral (PI), which works on the basis of instantaneous queue length. PIE is a variant of the PI controller.

2.1.2 CoDel

CoDel is easy to implement and uses two important parameters namely *target* and *interval* for managing buffers. It directly deals with queuing delay and is specially designed to solve the bufferbloat problem. Further, based on *per packet queue delay (sojourn time)*, CoDel decides whether a packet should be dropped or sent out during dequeue. Algorithm 2 shows the brief skeleton of the CoDel mechanism. On every packet arrival in the queue, CoDel marks the current timestamp to the packet header. Subsequently, while dequeuing the packet from the queue, CoDel calculates the queuing delay (packet sojourn time)

Algorithm 2: CoDel mechanism

Initialization: $target = 5ms$
$interval = 100ms$
$count = 0$
$Dropping = 0$
$next\_drop\_time = 0$

On every packet arrival
$pkt\_timestamp = current\_time$
On every packet departure
$sojourn\_time = current\_time - pkt\_timestamp$
if $Dropping == 1$ & $next\_drop\_time \geq current\_time$ **then**
  if $sojourn\_time \geq target$ **then**
    $count+ = 1$
    $next\_drop\_time = current\_time + \frac{interval}{\sqrt{count}}$
    Drop the Packet
  else if $sojourn\_time < target$ **then**
    $Dropping = 0$
    $count = 0$
    Dequeue the packet
else if $sojourn\_time > target$ & $Dropping == 0$ **then**
  $Dropping = 1$
  $count+ = 1$
  $next\_drop\_time = current\_time + \frac{interval}{\sqrt{count}}$
  Drop the packet

experienced by the packet by subtracting the timestamp from the current time. CoDel uses this sojourn time to compare with *target* (5 ms) and decides whether to enter into packet *dropping state* or not. Once CoDel enters into packet *dropping state* it sets control law to calculate the next drop time at which CoDel decides to drop the next packet based on the sojourn time.

2.1.3 PIE

The functionality of PIE is based on the original PI that attempts to control the *queue occupancy* around a desired *reference queue length*. Unlike PI, PIE aims to control the *queue delay* around a desired *reference queue delay*. The detailed working of PIE is explained in the following sub-sections.

A Random dropping

PIE randomly drops or enqueues incoming packets based on the drop probability (p) which is calculated PI. A random number generated between 0 and 1 is compared to p. The incoming packet is enqueued if p is less than the random number, otherwise dropped. However, the current implementation of PIE in the Linux kernel uses de-randomized dropping as suggested in Section 5.4 of RFC 8033.

B Drop probability calculation

PIE uses the rate of change of queuing delay to calculate the drop probability, which differentiates it from other mechanisms. Drop probability is calculated at regular *tupdate* interval (16ms in the Linux kernel) as shown in Eq. (2.3)

$$p(i) = p(i-1) + alpha * (qdelay - ref\_qdelay) + beta * (qdelay - qdelay\_old) \quad (2.3)$$

where

- *p(x)* is the drop probability calculated at time x
- *alpha* and *beta* are control parameters that help in drop probability calculation
- *ref_ qdelay* is the reference queuing delay (15ms in Linux kernel)
- *qdelay* is the current queuing delay
- *qdelay_ old* is the queuing delay in the previous sample

C Queue latency calculation

The following two approaches are used to calculate queuing delay in PIE:
a) Little's Law: This method calculates queuing latency from current queue length and average dequeue rate (avg_dq_rate) (Little and Graves, 2008) as shown in Eq. (2.4)

$$qdelay = \frac{qlen}{avg\_dq\_rate} \quad (2.4)$$

and avg_dq_rate is given by Eq. (2.5)

$$avg\_dq\_rate = 0.125 * \frac{count}{dtime} + 0.875 * avg\_dq\_rate \tag{2.5}$$

where,

- *qlen* is instantaneous queue length
- *dtime* is the time difference between the current and previous sample
- *count* is the amount of data dequeued in dtime
- *qlen* is instantaneous queue length
- *count* is the size of data dequeued between two dequeue periods
- *dtime* is the time difference between the current and previous dequeue

b) Packet timestamping: This method uses timestamping of packets to calculate the instantaneous per packet queuing latency at dequeue time. The current queuing delay is given by the queuing delay experienced by the most recently dequeued packet and is given by Eq. (2.6). This approach is adopted in the PIE implementation of FreeBSD.

$$qdelay = dequeue\_time - enqueue\_time \tag{2.6}$$

D Burst Tolerance

Burst tolerance is a mechanism to detect transient congestion and ensure that short bursts of packets are not subjected to random dropping behaviour of PIE. It sets a maximum interval (`max_burst`) of time for which a burst of packets are permitted to pass without dropping. When this interval is exceeded, packets may be subjected to random dropping.

2.1.4 Other solutions for bufferbloat

Table 2.1 shows the summary of the AQM mechanisms which are designed to solve the problem of bufferbloat. The key points highlighted in this table are i) approaches adopted to detect bufferbloat ii) advantages and iii) tools used.

Table 2.1: Soultions for bufferbloat

Author(s) and Year	Publication title	Approach to detect bufferbloat		Advantages	Tools used
		AQM abbreviations	Bufferbloat measures		
Francini (2012)	Periodic early detection for improved TCP performance and energy efficiency	PED	Instantaneous and average queue length with periodic time scale	Buffer size reduction up to 95%, TCP throughput and fairness	ns-2
Nichols and Jacobson (2012)	Controlling Queue Delay	CoDel	Queuing delay	Maintains less queuing delay	ns-2
Chen et al. (2012)	Statistical Adapting RED in Dynamic Networks	SARED	Standard deviation of instantaneous queue length for controlling RED parameter	Controls the oscillation of queue size for stability of TCP/RED systems.	ns-2, TCP eval suite.
Jiang et al. (2012)	Tackling Bufferbloat in 3G/4G Mobile Networks	DRWA	Receiver window adjustment	Reduces delay by 25% in cellular network and increase TCP throughput by 51%.	Real time analysis in mobile network.
Xue et al. (2013)	AFCD:An Approximated-Fair and Controlled-Delay Queuing for High Speed Networks	AFCD	Uses state information of flows to approximately estimate the flow's sending rate when packets are enqueued	Approximates fairness of TCP flows and very low queuing delay for high speed networks	Networking testbed CRON
Tahiliani and Shet (2013)	Analysis of Cautious Adaptive RED (CARED)	CARED	average queue length	Increases the throughput, minimizes packet drop and maintain queue length	ns-2
Showail et al. (2014a)	An Empirical Evaluation of Bufferbloat in IEEE 802.11n Wireless Networks	AMPDU	Aggregate MAC Protocol Data Unit (A-MPDU) MAC-layer frame	Reduce RTT delays with simultaneously increasing network throughput.	Real time testbed analysis

Palaniappan et al. (2013)	Bufferfloat Mitigation for Real-time Video Streaming using Adaptive Controlled Delay Mechanism	Adaptive CoDel	Queuing delay	Mitigate bufferbloat and improve the QoS parameters of real-time video stream.	SITL module of Opnet
Pan et al. (2013)	PIE: A Lightweight Control Scheme to Address the, Bufferbloat Problem	PIE	Current queuing delay and the trend of queue delay variations	Ensure low latency under various congestion situations	ns-2
Showail et al. (2014b)	WQM: An Aggregation-aware Queue Management Scheme,for IEEE 802.11n based Networks	WQM	Network load, channel condition, and frame aggregation level	Achieves the lesser queuing delay than CoDel and default Linux AQM pfifo in wireless network	Real time testbed
Ghoreishi et al. (2015)	Active Queue Management for Congestion Avoidance, in Multimedia Streaming	DC AQM	Distortion constrained in Video	Improves QoS for Video and maintain queue length in the buffer	Numerical analysis
Kuhn and Ros (2016)	Improving PIE's performance over high-delay paths	MAD-PIE	Queuing Delay over 30 ms	Deterministic drops are dominant when RTT increases, which results in lower queuing delays and better performance for VoIP traffic and small file downloads, with no major impact on bulk transfers.	ns-2
Kobayashi (2015)	LAWIN : a Latency-AWare InterNet Architecture for Latency Support on Best-Effort Networks	LAWIN	Applications specify latency itself	Advantages over other QoS approaches because it does not require any flow state, more than one queue, or any latency target	Real-time testbed
De Schepper et al. (2016)	PI^2: A Linearized AQM for both Classic and Scalable TCP	PI^2	same as PIE 15ms queue delay	It's a variant of PIE which auto-tunes and scales the parameters of PIE instead of heuristic approach	ns-2 and Real time testbed

Wang et al. (2017)	Active queue management algorithm based on data-driven predictive control	DATA-AQM	Data-driven predictive control	Superior than RED in terms of stability and robustness.	Simulator
Casoni et al. (2017)	How to avoid TCP congestion without dropping packets: An effective AQM called PINK	PINK	Number of active flows, flow's RTT and bottleneck bandwidth	Improves efficiency for multiplexed channels with low queue delay and flow fairness and without forced packet drop	ns-3
Høiland-Jørgensen et al. (2018)	Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways	CAKE and COBALT	Queuing delay	Provides bandwidth shaping based on goodput instead of throughput, handling DiffServ reasonably, improved flow hashing and filtration of TCP ACKs.	Real time testbed

These mechanisms are proposed by considering of responsive traffic only and very few mechanisms discuss about the fairness problem within the responsive flows, except COBALT and CAKE. However, none of the mechanisms deal with mixed traffic environment that consists of responsive and unresponsive flows together. Therefore, this literature gives the motivation to base our work when responsive and unresponsive flows coexist. Primarily, our focus is on two main issues i) excessive queue delay when unresponsive flows exist, and ii) fairness between responsive and unresponsive flows.

2.2 Evaluation Methodologies

The following methodologies have been used in this thesis for evaluating the AQM mechanisms:

2.2.1 ns-2

ns-2 is one of the most popular open-source network simulators (McCanne and Floyd, 1997) and widely used for the analysis of AQM mechanisms due to the availability of all AQM mechanisms. In our work, we have used ns-2 for the verification of the fluid model for CoDel and for the preliminary analysis of Minstrel PIE.

2.2.2 ns-3

Recently, ns-3 has become popular due to the availability of a large number of protocols. Additionally, ns-3 supports Direct Code Execution (DCE) to install and analyse the protocols of Linux TCP/IP stack (Henderson et al., 2008).

Thorough evaluation of an AQM mechanism requires significant time and effort. Hence, the AQM and Packet Scheduling Working Group at IETF published RFC 7928 which provides the guidelines to evaluate AQM mechanisms. An automated evaluation framework complying with the guidelines of RFC 7928 has been recently developed for ns-3 (Deepak et al., 2017). We use this evaluation suite to verify the desired behavior of Minstrel PIE and its relative performance against PIE under various traffic conditions. While RFC 7928 recommends several test scenarios to evaluate AQM mechanisms, we use the ones that are most relevant to the objective of designing Minstrel PIE.

2.2.3 Fluid modeling

The fluid model helps to capture the interaction between TCP flows and the RED mechanism (Misra et al., 2000). It forms the basis to analyze AQM mechanisms mathematically. This in turn also helps to understand the stability of AQM mechanisms and TCP flows. The basic fluid model presented in (Misra et al., 2000) highlights the sensitivity of the parameters in the RED mechanism. This fluid model has a great ability to understand and analyze the different congestion control mechanisms. Consequently, we have modified the same fluid model to analyze the performance of the CoDel mechanism mathematically. The mathematical analysis of the CoDel mechanism is discussed in Chapter 3.

2.2.4 Real time testbed

Network simulators give the abstract of the performance for the network protocols. The network simulators like ns-2 and ns-3 do not use the TCP/IP stack similar to the Linux kernel and traffic generators like iperf or netperf which might hides the real-time performance of the protocols. Consequently, along with simulation studies, verifying the effectiveness of the congestion control mechanisms in the real time environment is important. To overcome these limitations we have setup the real-time testbed for the real-time analysis of AQM mechanisms. Flexible Network Tester (Flent) (Høiland-Jørgensen, 2015) tool has been used for the traffic generation and collecting the stats from the AQM mech-

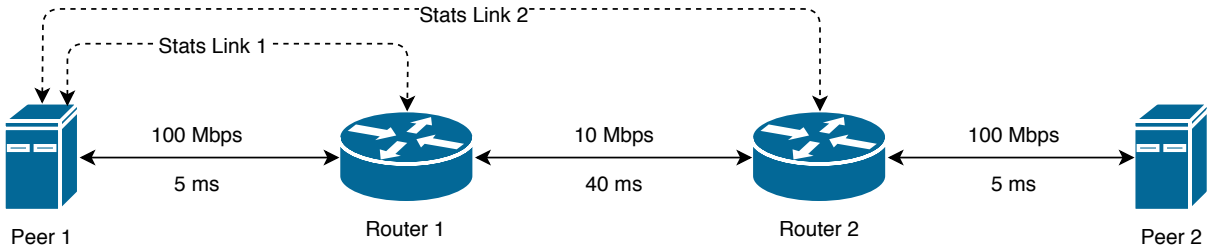


Figure 2.1: Example topology used for Virtual Flent

anisms.

2.2.5 Virtual Flent

AQM schemes need to be extensively evaluated before they can be deployed and used at scale. However, such extensive evaluation requires a lot of physical resources which are expensive, hard to maintain, and require a lot of time to set up correctly. Simulators such as ns-2 and ns-3 do not require a lot of hardware but, they neither process real-world network flows nor evaluate the code that actually processes them. Virtual machines and containers can overcome the above issues but they use a lot of system resources, limiting the scalability of the evaluation.

In order to accurately evaluate AQM mechanisms that are implemented in the Linux kernel without using a lot of additional hardware, we have designed a lightweight virtual testbed. The virtual testbed makes use of network namespaces, veth (virtual ethernet) devices, and various qdiscs (queuing disciplines) to emulate a real-world network topology. The iproute2 utility is used to set up and manages the virtual testbed. We further integrate this testbed with the Flent tool to evaluate AQM mechanisms.

The virtual testbed can be set up and/or modified as needed with ease, allowing the user to automate and run multiple tests over a long period of time. The following topology is shown in Figure 2.1 has been used for the experimentation.

2.3 Related Work

The main focus of this work is to explore the performance of popular bufferbloat solutions against unresponsive flows when they coexist with responsive flows. Very few studies have shown the impact of unresponsive flows on the performance of AQM mechanisms, such as CoDel and PIE. There are two major concerns in the AQM mechanisms when they face the unresponsive flows: i) uncontrolled queuing delay ii) unfairness between responsive

and unresponsive flows. We discuss the related work for both the problems in the following subsections.

2.3.1 Uncontrolled Queue Delay

Specifically, CoDel is more prone to unresponsive flows than PIE due to its inherent design issues, and hence, latency-sensitive applications face the problem of excessive queuing delay when they coexist with the unresponsive flows. To deal with this problem in the case of CoDel, the researchers have developed a new variant of CoDel namely, *CoDel BLUE alternate* (COBALT) (Morton, 2016).

A COBALT

Algorithm 3: Pseudocode of BLUE part in COBALT Queue Discipline

```
   Initialization: freeze_time =100 ms,
                   Increment_in_pdrop = 0.0025,
                   Decrement_in_pdrop = 0.00025
1  Enqueue: if q_len > limit then
2      Drop the packets before enqueue if prev_pdrop_update >= freeze_time then
3          Increment pdrop
4
   Dequeue: if queue is empty then
5      if prev_pdrop_update >= freeze_time then
6          Decrement pdrop
7  if pdrop > rand_number then
8      Drop this packet
       Retry dequeue with the next packet
9  else
10     Deliver packet
```

CoDel mechanism faces the two main limitations against unresponsive traffic: a) Once CoDel leaves dropping state after queue delay reaches below *target*, it resets the *count* variable in control law to zero. Later, when router gets congested again, CoDel needs the same amount of time to control the congestion as a previous congestion occurrence. This limitation of control law allows CoDel to increase queue delay which is unsatisfactory in real-time scenarios for latency-sensitive applications, and b) CoDel has no alternative mechanism against unresponsive flows to adapt the queuing delay rapidly. It works only

on the basis of control law to increase the drop frequency linearly, which is not sufficient to control queue delay against unresponsive traffic.

By considering the above design issue in the CoDel mechanism, COBALT gives an alternative of the BLUE mechanism which can adapt easily against unresponsive traffic by increasing packet drop probability periodically as mentioned in Algorithm 3.

COBALT uses two mechanisms CoDel and BLUE (Morton, 2016) that independently operate on the basis of queue delay and packet loss, respectively. As discussed above in Section 2.1.2, CoDel enters into the dropping state or non-dropping state depending on the basis of comparison of queue delay and *target* whereas, BLUE operates independently on the basis of packet loss and link utilization. The operation of BLUE is explained in the Algorithm 3. BLUE mechanism updates its drop probability *pdrop* only during two times if i) queue becomes full, and if ii) queue becomes empty, and these updates can happen only after every 100 ms *freeze_time* interval. Subsequently, along with updating *pdrop*, BLUE compare this *pdrop* with random number (*rand_number*) to decide whether packet has to be dropped or not. In COBALT, the BLUE mechanism operates only if CoDel fails to drop the packet.

2.3.2 Unfairness between Responsive and Unresponsive flows

The aggressiveness of unresponsive flows leaves very small space in the queue for responsive flows, which results in the unfairness problem. To avoid such issues, new hybrid AQM mechanisms are being designed by the researchers e.g., FlowQueue-CoDel (FQ-CoDel), FlowQueue-PIE (FQ-PIE), Common Application Kept Enhanced (CAKE), etc. The detailed working of Flow Queuing is explained in Chapter 5.

Chapter 3

Design and Evaluation of Modified CoDel

Controlled Delay (CoDel) is a modern AQM mechanism designed to control the queuing delay, and has some unique operating strategies when compared to other mechanisms like RED, ARED, PIE, PI Improved with square (PI^2) (De Schepper et al., 2016):

- It uses *per packet queue delay* as a measure of queue occupancy instead of instantaneous and average queue length measurements.
- It drops a packet during dequeue as opposed to other mechanisms that act during enqueue.
- It employs deterministic packet drops at regulated intervals rather than random drop probability measurements.

This chapter makes four contributions: first, we modify the fluid model presented in (Misra et al., 2000) to study the queue dynamics of CoDel and verify the correctness of this model by comparing its results with those obtained from *ns-2* (McCanne and Floyd, 1997). Secondly, we use the modified fluid model to evaluate CoDel against the following recommendations from IETF: *Design of AQM mechanisms should be independent of the transport protocol behavior*, *knob-free* and *easy to deploy*. We choose fluid modeling for evaluation since it is an effective and scalable means to capture the dynamics of AQM mechanisms, and their interaction with TCP flows. Thirdly, based on our observations, we confirm that the control law of CoDel requires tweaks to enhance its robustness against varying network traffic load. Subsequently, the fourth contribution is evaluating the CoDel

and performance in the real-time testbed using Flent. The real-time analysis shows that CoDel has limitations while performing in the low bandwidth scenarios. The source code to reproduce the results presented in this chapter has been made openly available[1].

3.1 Fluid Modeling

3.1.1 Genesis

The fluid model proposed in (Misra et al., 2000) captures the interactions between a set of TCP flows and RED. Assuming N long flows traverse through a single *RED enabled* router with transmission capacity C:

$$\frac{dW_i}{dt} = \frac{1}{R_i(q)} - (\frac{W_i}{2})\frac{W_i(t-\tau)}{R_i(q(t-\tau))}p(x(t-\tau)) \tag{3.1}$$

$$\frac{dx}{dt} = \frac{log_e(1-w_q)}{\delta}x(t) - \frac{log_e(1-w_q)}{\delta}q(t) \tag{3.2}$$

$$\frac{dq(t)}{dt} \approx -C + \sum_{i=1}^{N}\frac{W_i}{R_i(q)} \tag{3.3}$$

where, $W_i(t)$ and $R_i(t)$ denote the *cwnd* and RTT at time t of flow i ($1 \leq$ i $\leq$ N), respectively, $q(t)$ denotes the instantaneous queue length at bottleneck router, τ represents the round trip delay for a loss notification to reach the sender, C represents the capacity of the bottleneck router in packets, x represents the average queue length at a RED router, *p(x)* represents the packet drop probability as a function of x, w_q represents the smoothing constant for exponential weighted moving average x, δ represents the sampling interval of x and $R_i(t)$ is given by

$$R_i(t) = \beta_i + \frac{q(t)}{C} \tag{3.4}$$

where β_i is the fixed propagation delay and $\frac{q(t)}{C}$ models the queuing delay.

[1]https://github.com/steps-to-reproduce/codel-scripts

Eq. (3.1) models the Additive Increase Multiplicative Decrease (AIMD) behavior of TCP, where *cwnd* increases by one in every RTT and decreases by half on arrival of a loss notification. Eq. (3.2) and Eq. (3.3) provide an estimate of the average queue length and instantaneous queue length at a RED router, respectively.

This aforementioned model cannot be applied to CoDel in its current form because CoDel is significantly different than RED. Hence, we propose a modified fluid model to capture the queue dynamics of CoDel mechanism.

3.1.2 Proposed fluid model for CoDel

CoDel is easy to implement and does not require configuration settings for managing buffers. It directly deals with queuing delay, and is specially designed to solve the bufferbloat problem. Further, based on *per packet queue delay*, CoDel decides whether a packet should be dropped or sent out during dequeue.

If the *per packet queue delay* remains above 5 ms (*target*) consistently for a duration of 100 ms (*interval*), CoDel mechanism enters into a *dropping state* and remains there until the *per packet queue delay* reduces below the *target*. While in the dropping state, it regulates the packet drop frequency by using the control law shown in Algorithm 4, where t indicates the current time, *ndt* indicates the time when next packet should be dropped and *count* indicates the number of packets dropped thus far:

Algorithm 4: Control law of CoDel

Initialization: *interval* 0.1 seconds

Control_Law (*time_t t*)
    return $t + \frac{interval}{\sqrt{count}}$

Using the aforementioned control law, we can track the packet drop times as:

$$t + \frac{interval}{\sqrt{1}}, t + \frac{interval}{\sqrt{2}}, t + \frac{interval}{\sqrt{3}}, ..., t + \frac{interval}{\sqrt{n}}$$

Thus, the drop time of n^{th} packet ($d_n(t)$) would be:

$$d_n(t) = \sum_{1}^{n} \frac{interval}{\sqrt{count}} = interval \times \sum_{1}^{n} \frac{1}{\sqrt{count}} \tag{3.5}$$

It is known that:

$$\sum_{1}^{n} \frac{1}{\sqrt{count}} < \int_{1}^{n} count^{\frac{-1}{2}}$$

and

$$\int_{1}^{n} count^{\frac{-1}{2}} = 2 \times \sqrt{count}$$

So Eq. (3.5) can be rewritten as:

$$d_n(t) \simeq interval \times (2 \times \sqrt{count} + k) \tag{3.6}$$

where k is a constant used to represent the cumulative difference in the time values obtained from CoDel's control law and Eq. (3.6).

Thus, while in the *dropping state*, whether CoDel drops the packet at time t or not is predictable:

$$p_d(t) = \begin{cases} 1 & \text{if } t = d_i(t), \text{where, } i = 1, 2, 3, \ldots, n \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

where i represents the number of dropped packets, and $p_d(t)$ indicates whether packet is dropped at time t. Thus, the fluid model for CoDel can be given by the following equations:

$$\frac{dW_i}{dt} = \frac{1}{R_i(q)} - (\frac{W_i}{2})\frac{W_i(t-\tau)}{R_i(q(t-\tau))}p_d(t-\tau) \tag{3.8}$$

$$\frac{dq(t)}{dt} \approx -C + \sum_{i=1}^{N} \frac{W_i}{R_i(q)} \tag{3.9}$$

3.1.3 Correctness of the proposed fluid model

We have verified the correctness of CoDel's fluid model in two ways: (i) depending on the experimental observations while varying *count* from 1 to 2000 packets, we note that the cumulative difference between both time values is 43. Hence, we recommend k = -43 for our proposed fluid model. The analytical results show that the determined value of k works well and gives same *interval* values as obtained from manual calculations of square

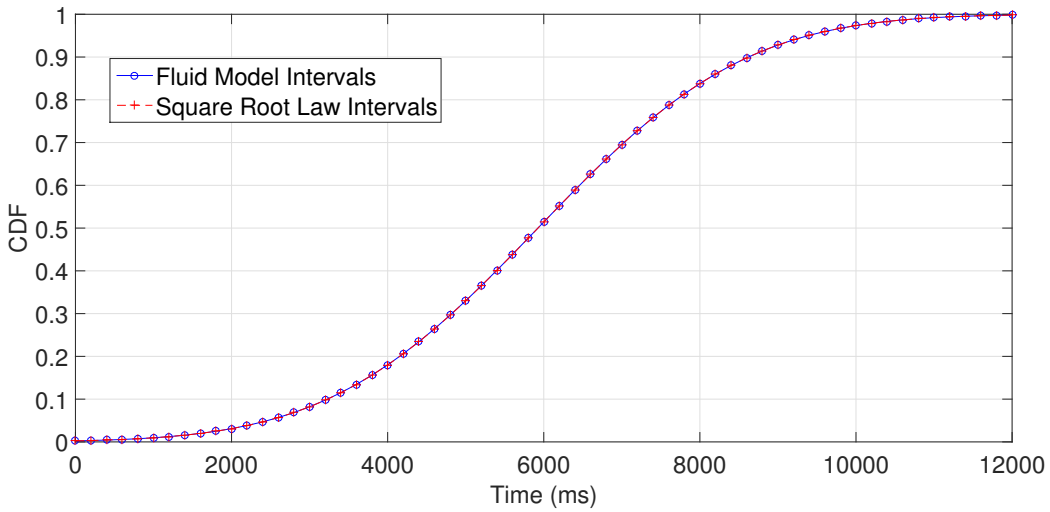


Figure 3.1: Analytical comparison of square root law and proposed fluid model *intervals*

root law. (ii) we show that the proposed fluid model for CoDel works fine and yields same results as the *ns*-2 implementation of CoDel.

We present the Cumulative Distribution Function (CDF) and Probability Density Function (PDF) for the results discussed in this work. CDF provides the percentage distribution of the obtained values in the range of 0 and 1. This helps in understanding whether the mechanism is controlling the values strictly within the specified *target*. PDF provides the probability distribution of the values obtained in the results and helps to understand the variance in the values.

A Verifying the setting of k

In this section, we compare the values of *interval* obtained by manual calculation of square root law and the ones obtained from the proposed fluid model. Figure 3.1 presents the CDF for more than 2000 *interval* values for 12 seconds (CDF is generated by obtaining several values of *interval*, counted in every 200 ms). We observe that the *interval* values obtained from the proposed fluid model *intervals* closely follow the ones obtained from the square root law with k=-43. We further confirm that our selected value of k (-43) works well by comparing the results obtained from the proposed fluid model to those obtained from *ns*-2 in the next section.

B Comparing with ns-2 results

In this section, we compare the results obtained from the proposed fluid model to those obtained from *ns*-2. We simulate a simple dumbbell topology in MATLAB and *ns*-2

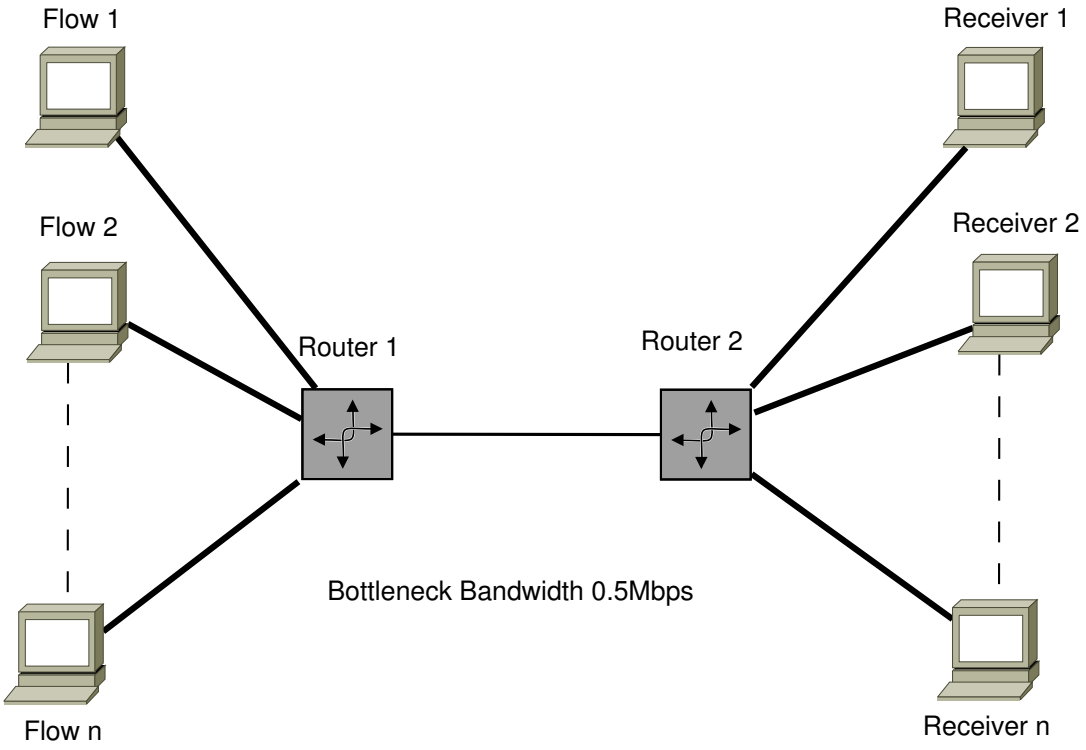


Figure 3.2: Dumbbell Topology used for all experiments

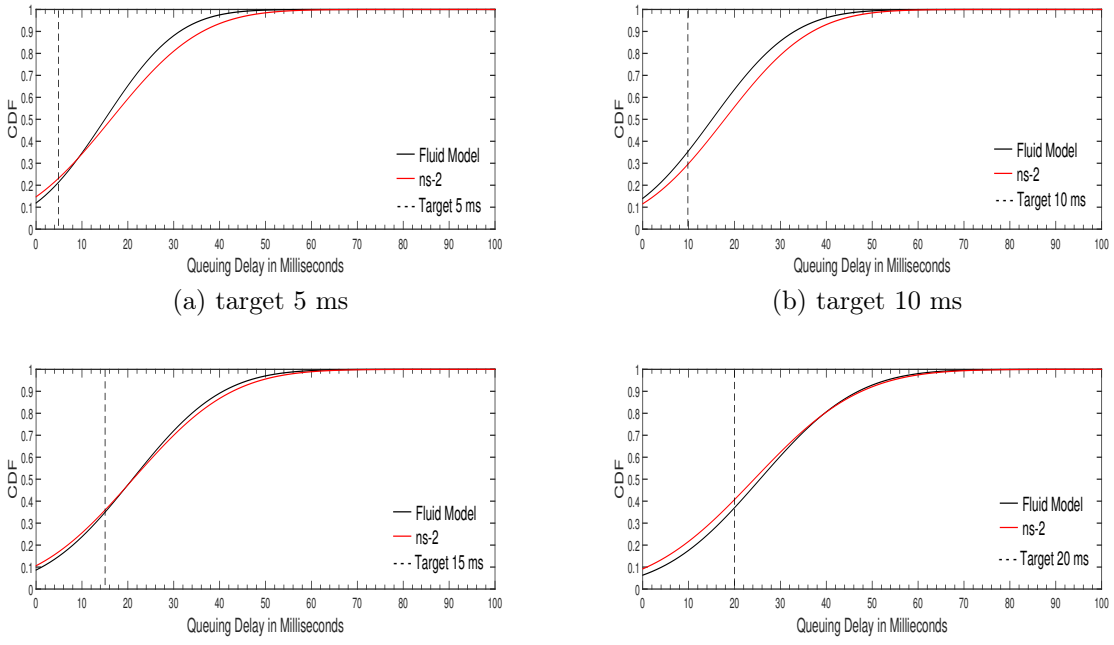
which consists of five long lasting TCP flows passing through a bottleneck bandwidth of 0.5 Mbps as shown in Figure 3.2. RTT is set to 100 ms. Default value of *interval* (100 ms) is used, but *target* is varied from 5 ms to 20 ms in steps of 5ms. The simulation duration is 200 seconds.

Figure 3.3 presents the CDF of queuing delay obtained with our proposed fluid model and *ns*-2 for different *target* values. We see that our proposed model tracks *ns*-2 results very closely. The marginal performance gap between both is because the fluid model does not consider TCP's *Slow Start* phase. We also note that CoDel fails to control the queue delay in all the scenarios and discuss about this issue later in the work. The results shown in this section can be reproduced by using the source code provided[2].

3.2 Control Law Sensitivity of CoDel

Studies have shown that CoDel has self-adaptation issues, and its performance is sensitive to appropriate setting of *target* and *interval* (Kuhn and Ros, 2016; Järvinen and Kojo, 2014; Raghuvanshi et al., 2013; Kulatunga et al., 2015). It has been also noted that CoDel's control law fails to adapt and control the queue length when large num-

[2]https://github.com/steps-to-reproduce/codel-scripts



(a) target 5 ms

(b) target 10 ms

(c) target 15 ms

(d) target 20 ms

Figure 3.3: CDF of queue delay with fluid model and *ns*-2

ber of unresponsive UDP flows coexist with responsive TCP flows. To understand this, we have performed the experiment in ns-2 against a varying numbers of responsive and unresponsive flows.

3.2.1 Impact of control law of CoDel

We verify the functionality of CoDel through an exhaustive evaluation conducted by using the *ns*-2 TCP Evaluation Suite (Hayes et al., 2007). *ns*-2 TCP Evaluation Suite is a test suite developed to gain an initial insight into the working of new TCP extensions. We extend this tool further to use it for evaluating the performance of AQM mechanisms.

To verify the robustness of CoDel against a large number of non-responsive flows, we conducted a set of experiments with dumbbell topology as shown in Figure 3.2. In this topology, sources and sinks are connected to routers through 20 Mbps links. Bottleneck bandwidth between the routers is set to 10 Mbps. AQM mechanisms are deployed on the sender side router. The bottleneck RTT is set to 80 ms and the buffer size used for these simulations is 8 * BDP, which accounts to 1600 packets with an average packet size of 500 bytes.

The evaluation comprises two sets of experiments: the first set of experiments include simulating TCP traffic by varying the number of FTP flows from 1 to 1000; and the

Table 3.1: Parameter settings for varying TCP and UDP flows experiments

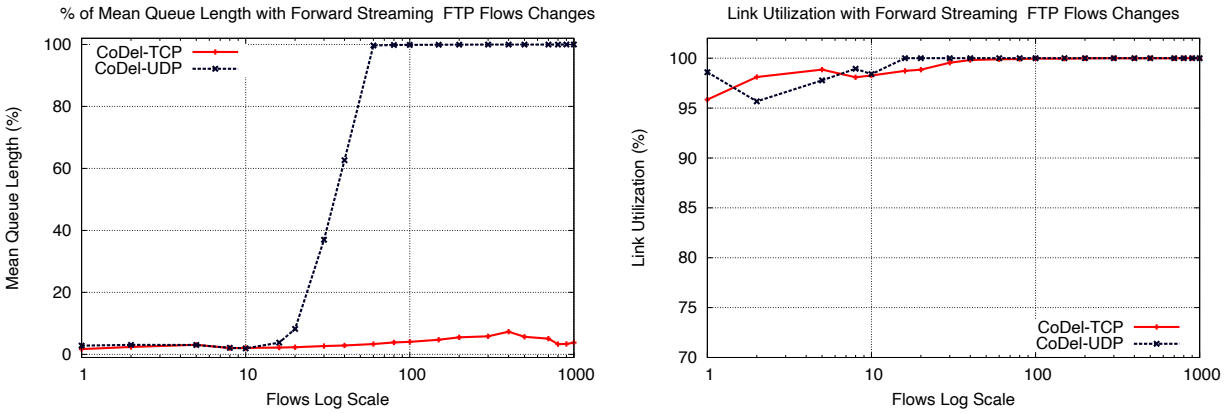
Parameter name	**Traffic Type**	**Changing FTP flows**	**Changing Streaming flows**
Number of Bottleneck	-	1	1
Bandwidth of bottleneck (Mbps)		10	10
RTT (ms)		80	80
Number of forward FTP flows	TCP CUBIC	Changing from 1 to 1000	5
Number of reverse FTP flows		5	5
HTTP connection generation rate (/s)	UDP Traffic	15	15
Number of Voice flows		5	5
Number of forward streaming flows		5	Changing from 1 to 1000
Number of reverse streaming flows		5	5
Simulation time (sec)		100	100

second set of experiments include simulating UDP traffic by varying the number of video streaming flows from 1 to 1000. Additionally, both experiments also contain: 5 voice flows, 5 FTP flows in reverse direction (i.e., from right to left in Figure 3.2), 5 streaming flows in reverse direction and HTTP traffic generated at a rate of 15 connections per second. Table 3.1 provides further details about the simulation parameters.

Figure 3.4 depicts the performance of CoDel against TCP and UDP traffic with number of respective flows varying from 1 to 1000. While the inferences from Figure 3.4(b) are clear, Figure 3.4(a) provides a deeper insight into the working of CoDel. Figure 3.4(a) shows that CoDel fails to maintain the mean queue length within the specified bounds when there is an increase in the number of unresponsive flows.

We infer the following from these results: a value of 100 ms for interval turns out to be quite large when there are more unresponsive flows in the network; and furthermore, the square root law employed by CoDel takes more time to adapt the packet drop rate. In

summary, CoDel takes longer to reach a point where it aggressively reacts to the presence of bursty unresponsive flows.



(a) Queue Length (b) Utilization

Figure 3.4: Performance of CoDel against varying number of TCP and UDP flows

3.2.2 Modified CoDel

The performance of CoDel is largely affected due to an interval of 100 ms, and the conservative adaptation of packet drop rate based on the square root law. To overcome these limitations, we propose to reduce the interval value to 30ms based on our experimental observations, and modify the control law of CoDel, as shown below in Algorithm 5, to adapt the packet drop rate quickly.

We propose both these modifications while considering the fact that the performance of CoDel should remain unaffected for responsive TCP flows. To ensure that this holds true, we repeat the set of experiments which were carried out in above in Figure 3.4 and re-evaluate the performance of CoDel with proposed changes.

Figure 3.5 depicts the performance of CoDel with our proposed changes, against TCP and UDP traffic with number of respective flows varying from 1 to 1000. Comparing Figure 3.5(a) to Figure 3.4(a) indicates that our proposed modifications help CoDel to maintain the mean queue length in the presence of unresponsive traffic, without hurting the performance of responsive traffic. Although Figure 3.5(b) shows a slight reduction in the link utilization for responsive flows initially, the utilization improves with a rise in the number of TCP flows.

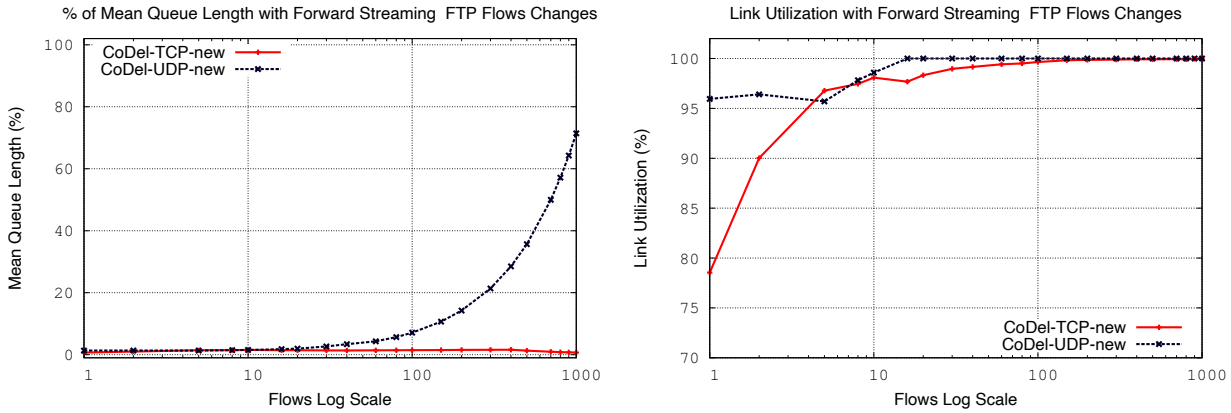
Our proposed fluid model does not require any modifications to account for changes suggested for the new *interval* value. However, to account for changes suggested for

Algorithm 5: Proposed modifications to tune CoDel parameters

Initialization: *interval* 0.03 seconds

Control_Law (*time_t t*)

Return $t + \dfrac{interval}{count}$



(a) Queue Length (b) Utilization

Figure 3.5: Performance of CoDel against varying number of TCP and UDP flows with interval 30 ms and modified control law

control law, our fluid model requires minor modifications to consider the packet drop times as a harmonic series as shown in Eq. (3.10):

$$d_n(t) = \sum_{1}^{n} \frac{interval}{count} = interval \times \sum_{1}^{n} \frac{1}{count} \tag{3.10}$$

where

$$\sum_{1}^{n} \frac{1}{count} < \int_{1}^{n} count^{-1}$$

$$\int_{1}^{n} count^{-1} = \ln(count) + \gamma + \sigma_n$$

where γ is a Euler-Mascheroni constant fixed at $= 0.577$, and $\sigma_n \sim \dfrac{1}{2n}$.

Eq. (3.10) can be rewritten as:

$$d_n(t) \simeq interval \times (\ln(count) + \gamma + \sigma_n) \tag{3.11}$$

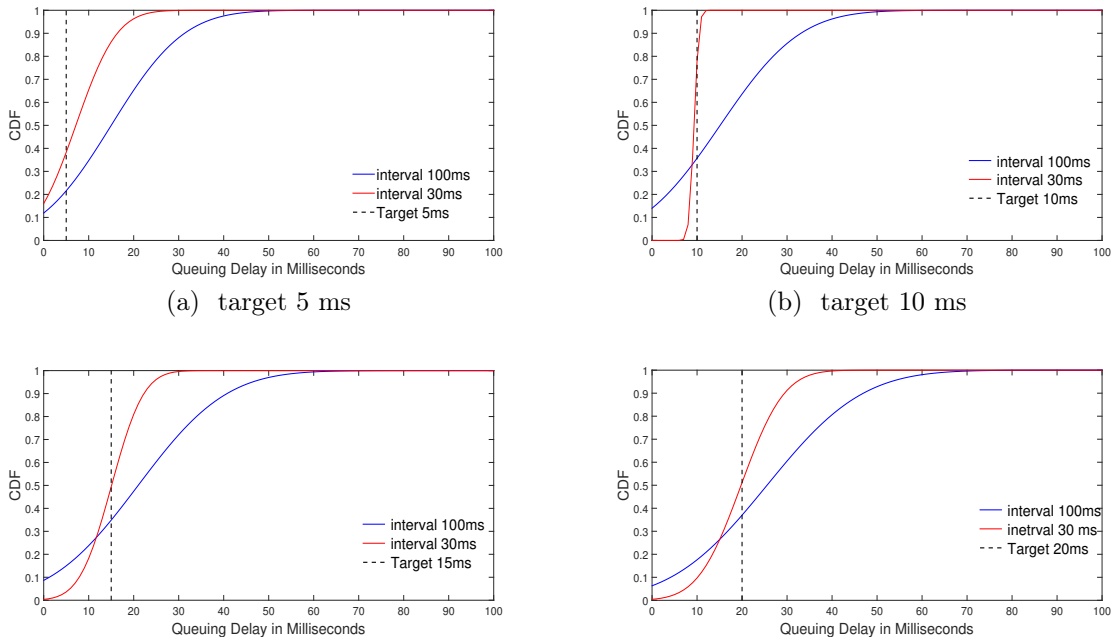
Thus, for the modified CoDel, Eq. (3.11) can be used instead of Eq. (3.6) in Eq.

(3.8) and Eq. (3.9). We use Eq. (3.11) in our fluid model to verify the correctness of the modified control law proposed in Eq. (3.10).

We compare the performance of original CoDel with modified CoDel using three different cases:

- **Case 1:** CoDel with *interval* = 30ms,
- **Case 2:** CoDel with a *modified control law*, and
- **Case 3:** original CoDel vs modified CoDel (i.e., *interval* = 30 ms and a *modified control law*).

We consider the same simulation setup as explained in subsection B of Section 3.1.3 for every case.



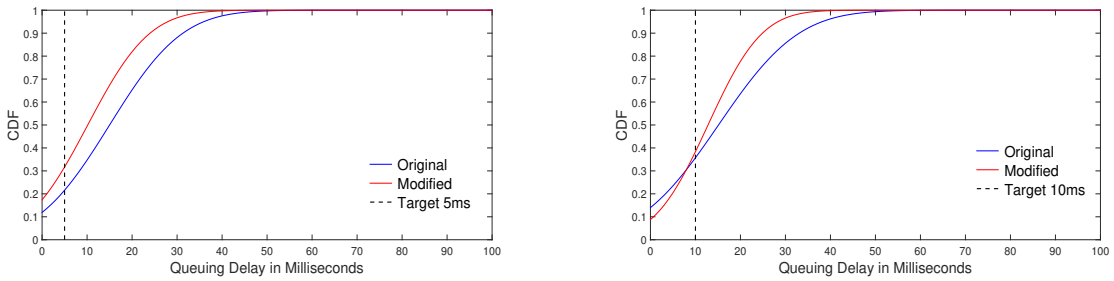
(a) target 5 ms (b) target 10 ms

(c) target 15 ms (d) target 20 ms

Figure 3.6: Case 1: CDF of Queue delay for Original CoDel vs CoDel with *interval* = 30 ms

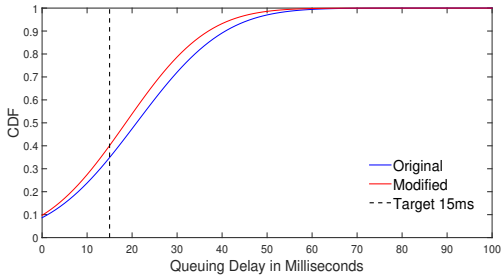
3.2.3 Case 1: CoDel with interval 30 ms

Figure 3.6 shows the CDF of queuing delay obtained from original CoDel and CoDel with the new value of *interval*. Figure 3.6 (a) - (d) present the results for different *target* values (5 ms to 20 ms). We see that with the new value of *interval*, CoDel has better control on

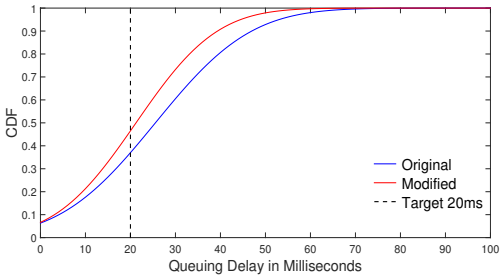


(a) target 5 ms

(b) target 10 ms



(c) target 15 ms



(d) target 20 ms

Figure 3.7: Case 2: CDF of Queue delay for Original CoDel vs CoDel with a *modified control law*

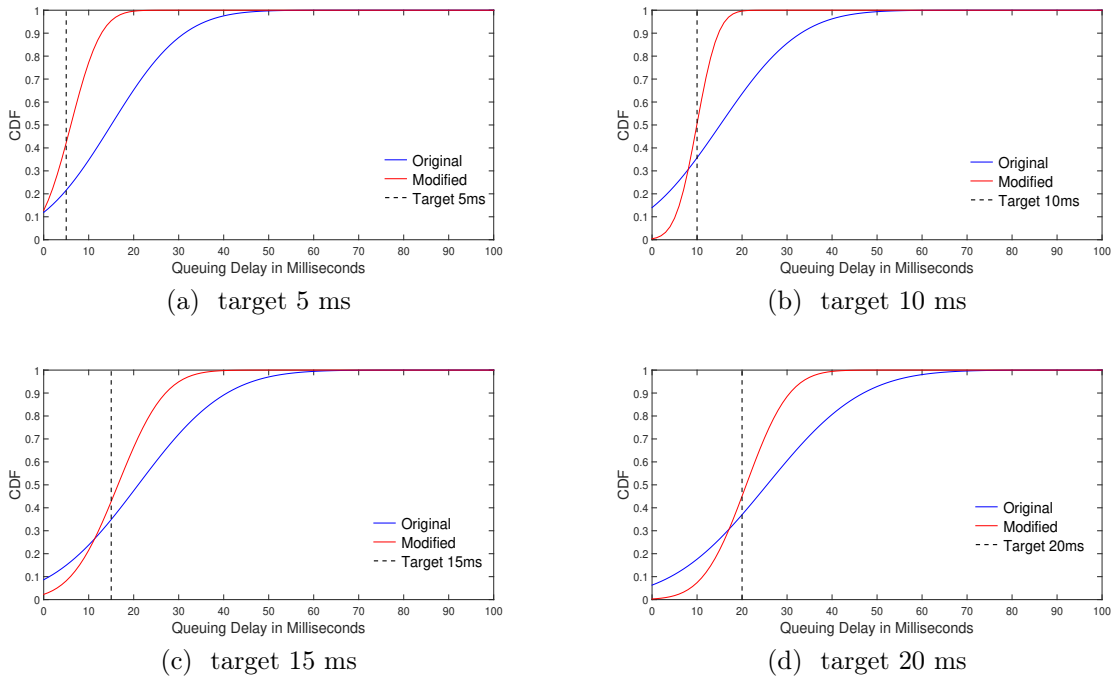
the queuing delay for all four *target* values. Reducing the value of *interval* increases the frequency of proactive packet drops by CoDel, thus allowing it to maintain lower queuing delays.

3.2.4 Case 2: CoDel with modified control law

Figure 3.7 shows the CDF of queuing delay obtained from original CoDel and CoDel with a modified control law. Figure 3.7 (a) - (d) present the results for different *target* values (5 ms to 20 ms). We observe that the modified mechanism performs better than the original mechanism. Using the actual value of *count* in the control law instead of $\sqrt{count}$ implies shorter *intervals* between packet drop times and thus, provides a tighter grip on the queuing delays. However, when compared to Case 1, we note that the magnitude of improvements in Case 2 is less. Hence, in the next case, we highlight the benefits of combining both approaches in the modified CoDel as discussed in Case 1 and 2, respectively.

3.2.5 Case 3: Original CoDel vs Modified CoDel

In this case, the main aim is to understand the benefits of applying two changes suggested above to improve the performance of CoDel. Figure 3.8 shows the CDF of queuing delay



(a) target 5 ms (b) target 10 ms

(c) target 15 ms (d) target 20 ms

Figure 3.8: Case 3: CDF of Queue delay for Original CoDel vs Modified CoDel (combined Case 1 and Case 2)

obtained from the original CoDel and modified CoDel. Figure 3.8 (a) - (d) present the results for different *target* values (5ms to 20ms). We see that the modified CoDel controls the queue delay within *target* more often than the original mechanism. A collective change made in the value of interval and control law yields more promising results than the individual changes made in the previous two cases, which is in line with the observations made in Figure 3.5.

However, to verify whether this performance improvement in queuing delay does not reduce the link utilization, we capture the congestion window evolution (aggregate) for all five senders in Figure 3.9. In the PDF plots[3] shown in Figure 3.9, the width of the bar is decided by choosing the bin size as 15 which is used more commonly (Forbes et al., 2011). It is noted that PDF of both original and modified control law is similar, despite some variations in the density. Thus, we confirm that the control law proposed in Algorithm 5 does not affect the performance when only TCP flows share the bottleneck link. From the point of stability analysis, we observe that the original CoDel suffers from large variations in the *cwnd*. On the other hand, we see that modified CoDel is more stable and suffers from fewer variations. This is confirmed by Table 3.2 which depicts the variance of *cwnd* observed with the original CoDel and modified CoDel. This type of behavior is desirable

[3]Unlike probability, PDF can have values greater than 1 (Abramowitz and Stegun, 1964)

for applications that demand minimal jitter and/or consistent throughput.

Table 3.2: Variance in *cwnd* with CoDel and modified CoDel

target **(in ms)**	**CoDel**	**Modified CoDel**
5	0.5808	0.4105
10	0.6384	0.4973
15	0.7574	0.6325
20	0.7280	0.7072

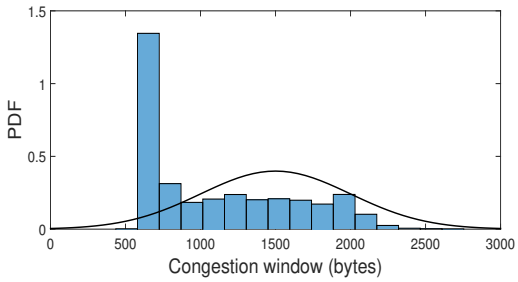
Table 3.3: Throughput with CoDel and modified CoDel

target **(in ms)**	**CoDel (Mbps)**	**Modified CoDel (Mbps)**
5	0.3625	0.3697
10	0.4065	0.4005
15	0.3913	0.3844
20	0.4023	0.3928

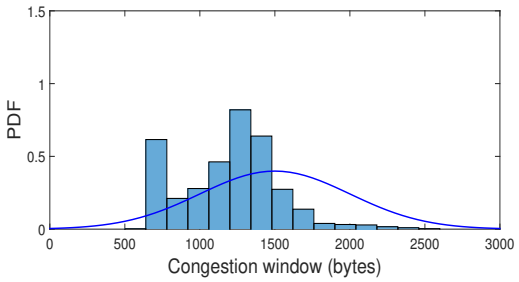
To further analyze the performance of both algorithms, we plotted instantaneous throughput as shown in Figure 3.10 and computed average throughput as shown in Table 3.3. Additionally, we also track the number of packets dropped by CoDel and modified CoDel as shown in Table 3.4. It is observed from Figure 3.10 that modified CoDel provides similar throughput as original CoDel, but with significantly less number of packet drops as shown in Table 3.4. The proactive behavior of modified CoDel provides timely congestion signals to the TCP senders, thus reducing the number of packet drops at a later stage.

3.3 Evaluation using real-time test-bed

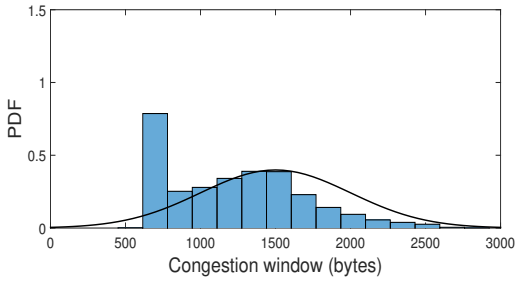
To evaluate the performance of CoDel in real time environment we re-analyze it in a Linux kernel testbed. We setup the testbed using three machines. Flexible network tester (Flent) (Høiland-Jørgensen, 2015) tool has been used to generate the TCP traffic and to



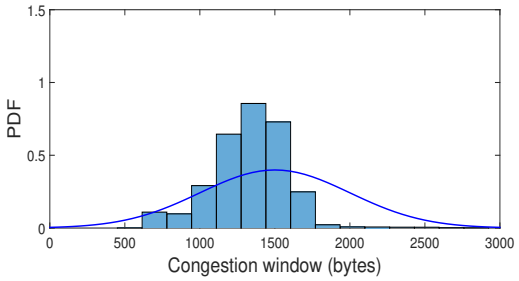
(a) Original control law with target 5 ms



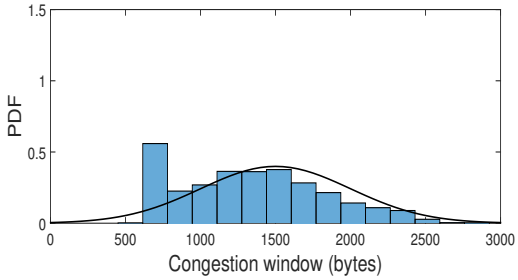
(b) Modified control law with target 5 ms



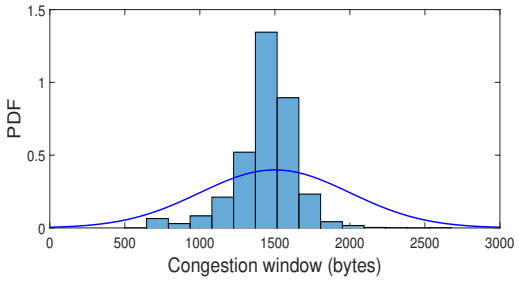
(c) Original control law with target 10 ms



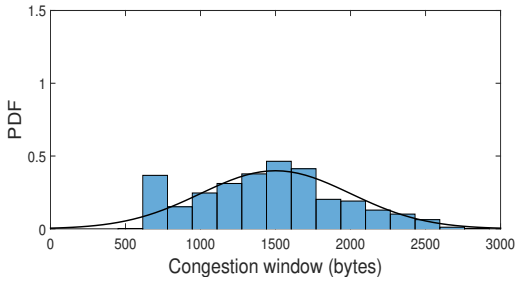
(d) Modified control law with target 10 ms



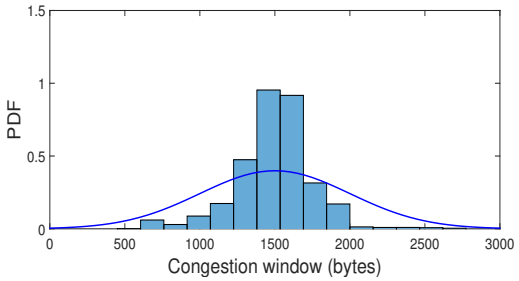
(e) Original control law with target 15 ms



(f) Modified control law with target 15 ms

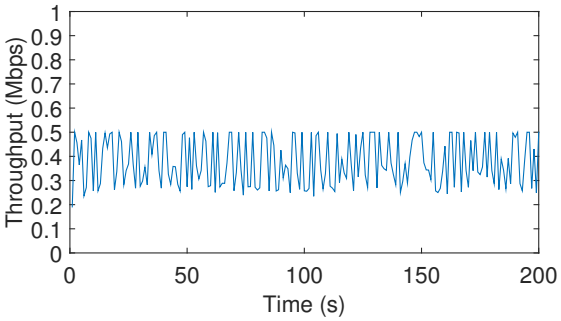


(g) Original control law with target 20 ms

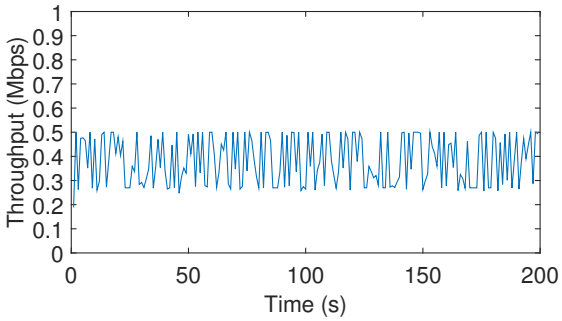


(h) Modified control law with target 20 ms

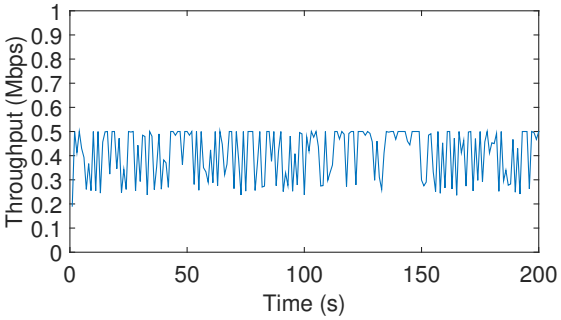
Figure 3.9: Cont... Congestion window evolution with original and modified control law



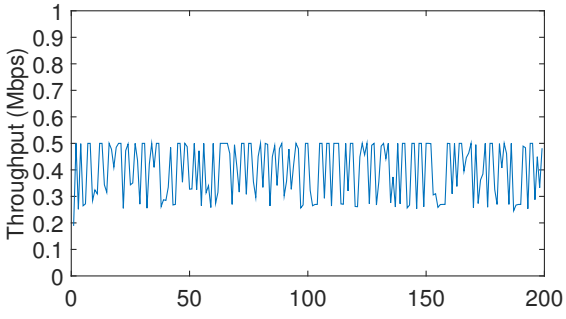
(a) Original control law with target 5 ms



(b) Modified control law with target 5 ms



(c) Original control law with target 10 ms



(d) Modified control law with target 10 ms

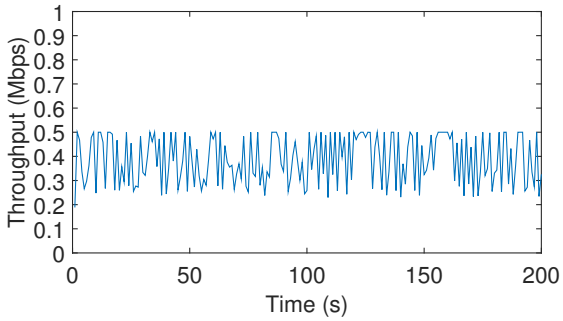
Figure 3.10: Throughput for original and modified control law

Table 3.4: Total number of packets dropped with CoDel and modified CoDel

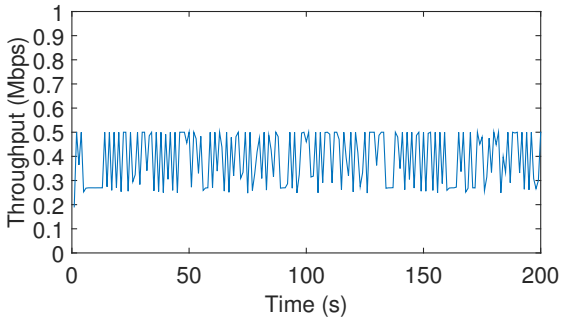
target **(in ms)**	**CoDel**	**Modified CoDel**
5	68875	47360
10	61980	50890
15	61205	40080
20	62110	50415

collect the stats, and iperf has been used to generate the UDP traffic. `tcp_5up` test has been used which configures 5 TCP flows in upload direction along with one UDP flow. The UDP flow is started at 26^{th} second and stopped at 75^{th} second. The total duration of the experiment is 100 seconds.

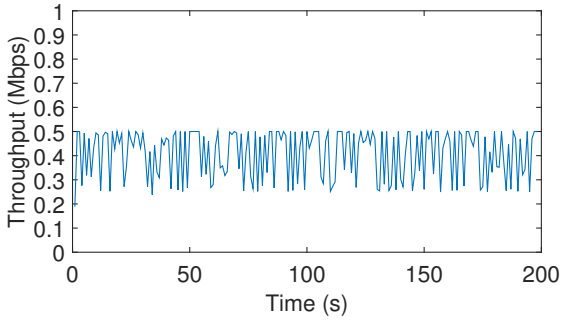
Figure 3.11 (a) and (b) shows the queue delay and link utilization for the CoDel and Modified CoDel with 0.5 Mbps bottleneck bandwidth. It can be observed from Figure 3.11 (a) that Modified CoDel maintains the queuing delay lower than CoDel along with same link utilization as shown in Figure 3.11 (b). However, CoDel is not able to maintain the queue delay under *target* in both the cases. These observations are in line with



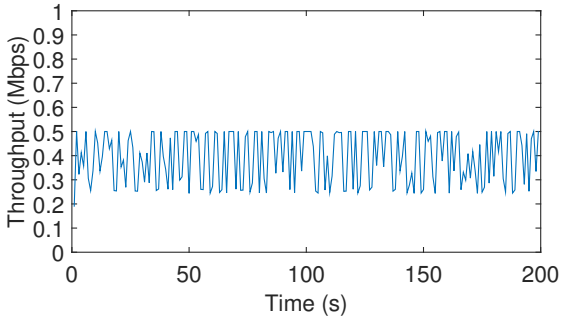
(e) Original control law with target 15 ms



(f) Modified control law with target 15 ms



(g) Original control law with target 20 ms



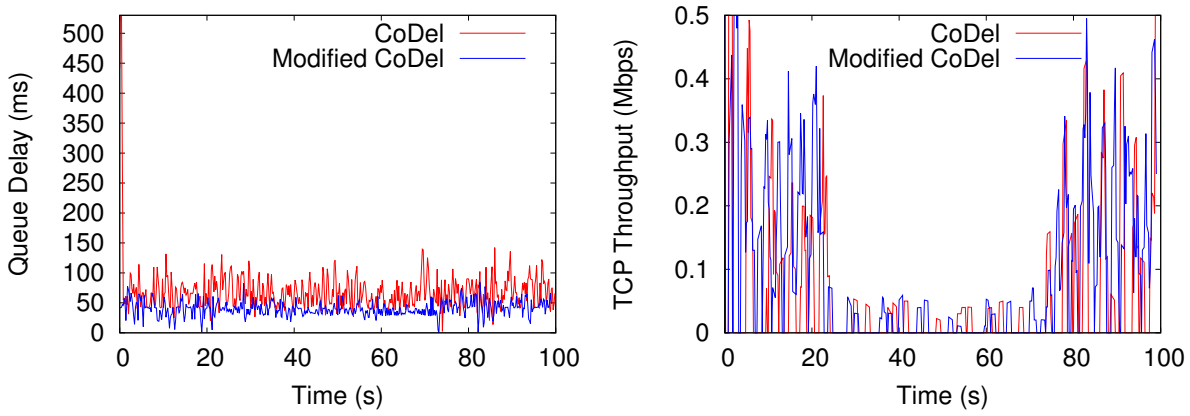
(h) Modified control law with target 20 ms

Figure 3.10: Cont... Throughput for original and modified control law

those mentioned in the official website of bufferbloat community[4], that CoDel does not perform well under low bottleneck bandwidth scenarios (e.g., 0.5 Mbps) in real time testbed whereas, it works fine in simulations (as seen in our results obtained from ns-2 and fluid model). The exact reason for this behavior of CoDel is still unknown, but it has been identified that the value of *target* needs appropriate setting for CoDel to work in such low bandwidth scenarios.

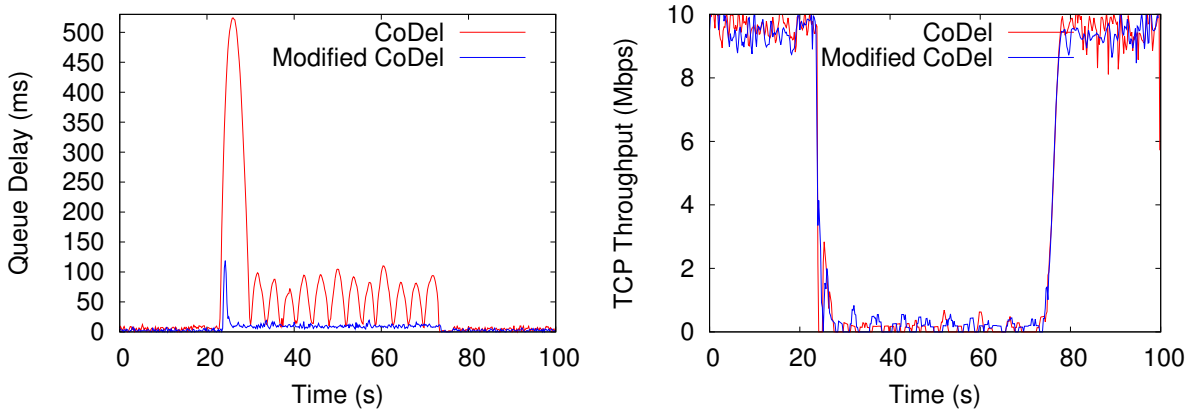
To verify that the problem indeed is only with low bandwidth scenarios, we re-evaluate CoDel and Modified CoDel with 10Mbps bottleneck bandwidth, while using the same experimental setup. Figure 3.12 (a) and (b) show the queue delay and link utilization for CoDel and Modified CoDel, respectively. Figure 3.12 (a) shows that CoDel performs well when only TCP traffic exists from 0 to 25 seconds, and 76 to 100 seconds. It is unable to control the queue delay when UDP traffic is enabled from 26 seconds to 75 seconds. In contrast to CoDel's behavior against UDP traffic, Modified CoDel maintains better control on queue delay. Figure 3.12 (b) shows that both CoDel and Modified CoDel achieve similar link utilization.

[4]https://www.bufferbloat.net/projects/codel/wiki/#known-issues



(a) Queue Delay (b) TCP Throughput

Figure 3.11: Mix TCP and UDP with 0.5 Mbps bottleneck bandwidth



(a) Queue Delay (b) TCP Throughput

Figure 3.12: Mix TCP and UDP with 10 Mbps bottleneck bandwidth

3.4 Constraints of CoDel

There are two fundamental concerns with the functionality of CoDel: (i) the transition from a *non-dropping* state to a *dropping* state requires waiting for at least 100 ms. This affects the behavior when queue delay falls below the desired value and then rises again shortly afterwards, and (ii) in *dropping* state, the control law regulates the packet drop frequency of CoDel. This control law increases the packet drop frequency based on inverse square root law, which controls the queue delay within the specified bound against responsive flows only, and is not suitable against unresponsive flows (Morton, 2016).

Although we show that the modified CoDel successfully increases the packet drop frequency by reducing the value of *interval* to 30 ms and applying a different control law, it is apparent that this is not a reliable solution to address the concerns of CoDel with

unresponsive traffic. While this solution would work for a certain number of unresponsive flows, its scalability is limited. Hence, instead of making any more attempts to improve CoDel, we explored PIE with an aim to resolve the concerns with unresponsive traffic.

Unlike CoDel, PIE employs packet drop probability (p) to enqueue/drop the incoming packets. p is calculated depending on (i) the difference between the current queue delay and reference queue delay, and (ii) whether the current queue delay is increasing or decreasing. PI controllers adjust the drop probability quick enough to react to sudden changes in the queue delay. Hence, we decided to base our work on PIE. Moreover, it is a defacto AQM used by CableLabs since DOCSIS 3.1 cable modems (RFC 8034), and studies have shown that PIE is likely to provide better control on queue delay than other mechanisms in case of extreme overload (Järvinen and Kojo, 2014; Kuhn et al., 2017).

3.5 Inferences

We have proposed a modified fluid model that can be used to obtain an in-depth working of the CoDel mechanism and evaluate its parameter sensitivity. The proposed fluid model has been verified by comparing its results with those obtained from *ns*-2. In addition, this work discusses a potential use case where the modified fluid model is applied to verify the robustness of a new control law designed for CoDel recently. The proposed fluid model can be extended to provide support for modeling unresponsive UDP flows along with responsive TCP flows, but how to measure the impact of the presence of unresponsive flows on responsive flows remains a challenging work.

Further, we identify that CoDel has inherent design issues and it leads to limited performance improvement with modified CoDel. Hence, we switched our focus from CoDel to PIE mechanism because studies have shown that PIE is stable against unresponsive flows.

Chapter 4

Minstrel PIE

This chapter discusses a simple and easy-to-deploy extension to the PIE mechanism, called Minstrel PIE, to increase its robustness when unresponsive flows coexist with TCP flows. Minstrel PIE adapts itself and timely drops (or marks[1]) packets to control the queue delay when the traffic load increases, otherwise operates similar to PIE. The idea is to regulate the *reference queue delay* (*qdelay_ref*)parameter of PIE to adjust the drop probability in Minstrel PIE. Depending on the results obtained through simulations and real-time tests it is observed that Minstrel PIE maintains a better trade-off between queue delay and link utilization when unresponsive flows coexist with TCP flows. It performs similar to PIE in other network settings.

4.1 Impact of fixed *qdelay_ref*

Despite gaining significant attention, the implications of keeping *qdelay_ref* fixed in PIE are not widely studied. This section highlights the need to adapt *qdelay_ref* in PIE. We configure a dumbbell topology as shown in Figure 4.1 in ns-2 with different traffic loads to analyze the impact of having a fixed value of *qdelay_ref*. Three traffic loads are configured in the simulation: (i) Light traffic (5 TCP flows), (ii) Heavy traffic (50 TCP flows) and (iii) Mix traffic (5 TCP and 2 UDP flows). These configurations are identical to the ones used by the authors of PIE (Pan et al., 2013) and PI^2 (De Schepper et al., 2016), except that we have used Compound TCP (Tan and Song, 2006; Tan et al., 2006) for our simulations instead of TCP NewReno (RFC 6582). PIE has been configured with

[1]Marking refers to the setting of a bit in the packet header by using congestion signaling approaches like Explicit Congestion Notification (ECN) (RFC 3168). The words 'drops' and 'marks' are used interchangeably in this chapter.

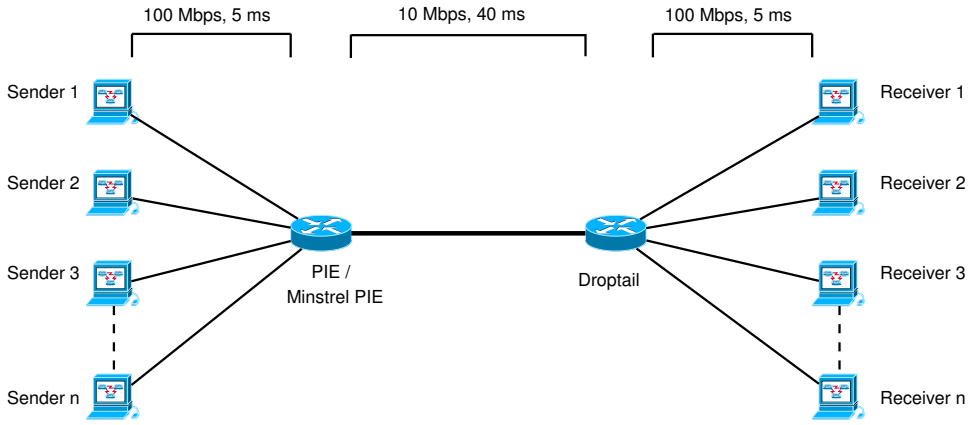
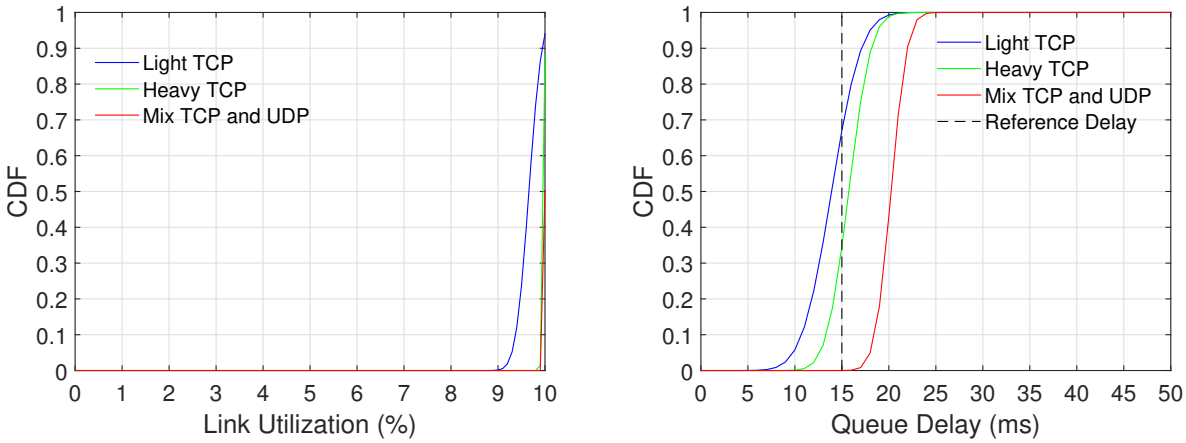


Figure 4.1: Dumbbell Topology used in *ns-2* experiments

the values recommended in RFC 8033. In this paper, all the simulations are run for 100 seconds and the results presented in Figure 4.2 through Figure 4.12 depict the average values obtained by repeating the simulation with 25 different random seed values.



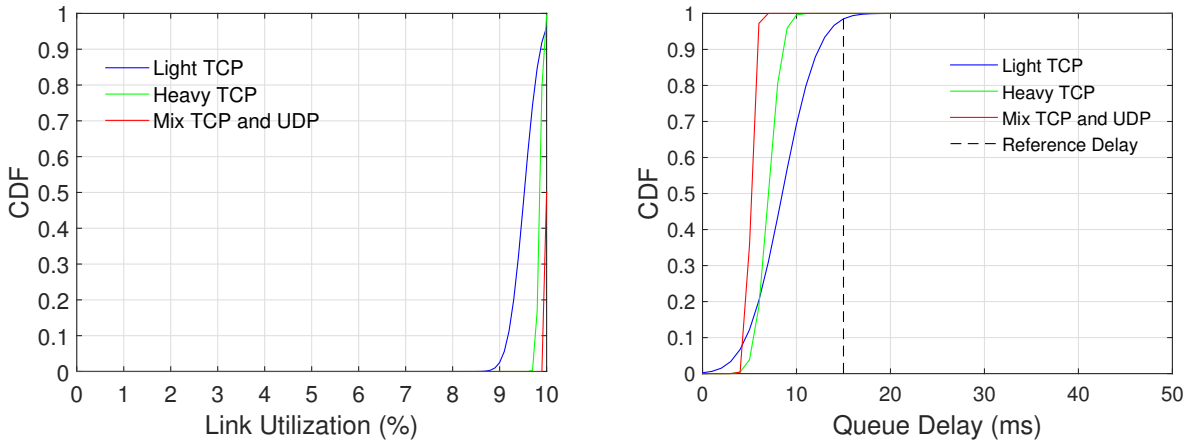
(a) Link Utilization

(b) Queue Delay

Figure 4.2: Link Utilization and Queuing Delay with PIE

Figure 4.2 presents the CDF for link utilization and queue delay obtained with PIE. Increasing the number of responsive flows did not largely impact the performance, but adding a few unresponsive flows led to a significant rise in the queue delay. This happens because PIE does not drop sufficient number of packets to keep the queue delay under control. We believe that this concern can be addressed by adapting *qdelay_ref* (*albeit within acceptable bounds*) and subsequently regulating the packet drop probability (p) to drop appropriate number of packets. This approach forms the basis of Minstrel PIE.

We repeat the simulations described above with Minstrel PIE. Figure 4.3 presents the CDF for link utilization and queue delay. As traffic load increases, Minstrel PIE increases



(a) Link Utilization (b) Queue Delay

Figure 4.3: Link Utilization and Queuing Delay with Minstrel PIE

its packet drop probability based on *qdelay_ref* adjustments and achieves better control on queue delay without hurting the link utilization. The next section describes the working of Minstrel PIE in detail.

4.2 Minstrel PIE

4.2.1 Design

The design of Minstrel PIE is tailored to improve the tradeoff between queue delay and link utilization when responsive and unresponsive flows coexist. It increases *qdelay_ref* when the link is underutilized and decreases *qdelay_ref* when the link is over-utilized. Average dequeue rate (*avg_dq_rate*) measurements in PIE serve as a good approximation of the current link utilization. Minstrel PIE leverages these measurements and accordingly adapts the *qdelay_ref*. The *initial value* of *qdelay_ref* is set to 15 ms which is a *fixed* default value suggested in RFC 8033. Other components in Minstrel PIE are same as PIE. Algorithm 6 presents the pseudo code of the proposed mechanism.

A new parameter called *maximum average dequeue rate* (*maxavg_dq_rate*) is introduced in Minstrel PIE to keep track of the maximum link utilization observed so far. At every *t_update* interval, *avg_dq_rate* is compared with *maxavg_dq_rate* to decide whether *qdelay_ref* should be updated. Subsequently, the updated value of *qdelay_ref* is used to calculate the new drop probability. *maxavg_dq_rate* is internally maintained by Minstrel PIE and does not require settings from the user.

The following subsections provide a detailed insight into the working of Minstrel PIE:

Algorithm 6: Minstrel PIE

Input : avg_dq_rate
Initialization: $qdelay\_ref = 15ms$, $t\_update = 16ms$, $maxavg\_dq\_rate = 0$
Output : updated value of $qdelay\_ref$

On every t_update interval
// Track the highest *avg_dq_rate*
if $avg\_dq\_rate > maxavg\_dq\_rate$ **then**
    $maxavg\_dq\_rate = avg\_dq\_rate$
if $avg\_dq\_rate <= 0.9 * maxavg\_dq\_rate$ **then**
    // Increase *qdelay_ref*
    $qdelay\_ref \mathrel{+}= \frac{|qdelay\_ref - cur\_delay|}{2}$
else
    // Decrease *qdelay_ref*
    if $cur\_delay < qdelay\_ref$ **then**
        $qdelay\_ref = cur\_delay$
    else
        $qdelay\_ref \mathrel{-}= \frac{|qdelay\_ref - cur\_delay|}{2}$
if $qdelay\_ref < 5ms$ **then**
    $qdelay\_ref = 5ms$
else if $qdelay\_ref > 15ms$ **then**
    $qdelay\_ref = 15ms$
Calculate Drop Probability p

A Increase qdelay_ref

Minstrel PIE keeps track of *maxavg_dq_rate* and increases the *qdelay_ref* when the link is not adequately utilized. Specifically, it increases the *qdelay_ref* when the *avg_dq_rate* $\leq$ 90% of the *maxavg_dq_rate*. A binary increment (as shown in Algorithm 6) is applied to the value of *qdelay_ref*. This increment helps Minstrel PIE to adjust the *difference between cur_delay and qdelay_ref* which is tracked by PIE's control parameter α during the drop probability calculation. These adjustments ensure that the overall drop probability is decreased *quickly*, and subsequently, the link utilization increases. The upper bound for *qdelay_ref* is set to 15 ms in Minstrel PIE. The design considerations for choosing this setting and the above mentioned increase/decrease threshold (i.e., 90% of *maxavg_dq_rate*) are discussed later in Section 4.2.2.

B Decrease qdelay_ref

Minstrel PIE decreases the *qdelay_ref* when the link is adequately utilized. Specifically, it decreases the *qdelay_ref* when $avg\_dq\_rate > 90\%$ of the $maxavg\_dq\_rate$. However, decreasing the value of *qdelay_ref* abruptly may hurt the link utilization; so the following approach has been adopted to decrease the value of *qdelay_ref* smoothly:

If $cur\_delay < qdelay\_ref$, decreasing *qdelay_ref* and setting it to *cur_delay* is safe. But if $cur\_delay > qdelay\_ref$, a binary decrement is applied to decrease the value of *qdelay_ref* smoothly. In the former case, Minstrel PIE attempts to stabilize the value of drop probability by setting *qdelay_ref* to *cur_delay* whereas in the latter case, Minstrel PIE attempts to gradually increase the drop probability by increasing the value of (*cur_delay* - *qdelay_ref*). The motivation to use binary decrement is to ensure *quick adaptation* of *qdelay_ref* to achieve the desired trade-off. The lower bound for *qdelay_ref* is set to 5 ms in Minstrel PIE. Section 4.2.2 discusses the design considerations for choosing this value.

4.2.2 Parameter Settings

A Lower and Upper Bound for *qdelay_ref*

The inferences to set lower and upper bound for *qdelay_ref* in Minstrel PIE have been derived from the simulation studies presented in (Kuhn et al., 2017) where the *qdelay_ref* in PIE is set within a range of 5 ms to 20 ms. Hence, the lower bound for *qdelay_ref* in Minstrel PIE is set to 5 ms, which is also the default value used by CoDel. The upper bound for *qdelay_ref* in Minstrel is set to 15 ms instead of 20 ms. The primary reason is that higher values of *qdelay_ref* exhibit more variations in the queue delay (See Figure 7(b) of (Kuhn et al., 2017)). Moreover, RFC 8033 recommends a default value of 15 ms for *qdelay_ref*. Thus, Minstrel PIE does not deviate from the primary goals of PIE.

B Initial value of *qdelay_ref*

Initially, the buffers are empty and it takes a few RTTs for the queue to build up. Setting a low initial value for *qdelay_ref* would be too restrictive and it might not allow the link utilization to grow beyond a certain limit. Hence, we recommend that Minstrel PIE sets the initial value of *qdelay_ref* to its upper bound, i.e., 15 ms.

C Setting the increase/decrease threshold

Quite a few prior works have demonstrated that PIE achieves more than 90% link utilization in most of the scenarios (Kuhn et al., 2017; Kulatunga et al., 2015; Kuhn and Ros, 2016). Based on the observations made in the literature, and our own evaluations, we recommend 90% of *maxavg_dq_rate* to be used as a threshold for Minstrel PIE to increase / decrease the value of *qdelay_ref*.

4.2.3 Support for Explicit Congestion Notification

Explicit Congestion Notification (ECN) (RFC 3168) mechanism marks packet headers (TCP and IP) to notify endpoints of congestion that may be developing in a bottleneck queue, without resorting to packet drops. Minstrel PIE does not require modifications to support ECN style packet marking. It is recommended to use ECN marking with Minstrel PIE to avoid any loss of throughput for TCP flows, especially when *qdelay_ref* is decreased from 15 ms to 5 ms.

4.2.4 Implementation

To verify the correctness of the proposed idea, PIE code in ns-2.36.rc1, ns-3.27 and Linux kernel 4.15 has been modified to support Minstrel PIE. It requires 18, 37 and 33 lines of code change to PIE implementation in ns-2 (McCanne and Floyd, 1997), ns-3 (Henderson et al., 2008) and Linux kernel, respectively. Moreover, Minstrel PIE does not add any new parameter which requires explicit settings from the user.

In the Linux kernel, the `net/sched/sch_pie.c` is modified to implement Minstrel PIE and to enable Minstrel PIE through the `tc` command, `q_pie.c` is modified in the `iproute2` package. To enable Minstrel PIE, passing 'minstrel' as a parameter in the tc-pie command is sufficient[2].

4.3 Evaluation

Our evaluations consisted of three sets of experiments that compare the performance of Minstrel PIE with PIE using (i) preliminary set of simulation scenarios, same as the

[2]Example: `sudo tc qdisc add dev eno1 parent 1:10 handle 1100: pie limit 200 target 15 ms tupdate 16 ms minstrel`

ones used in (Nichols and Jacobson, 2012; Pan et al., 2013; De Schepper et al., 2016) (ii) simulation scenarios described in RFC 7928 for comparing the performance of AQM algorithms and (iii) real time tests provided in Flexible Network Tester (Flent) (Høiland-Jørgensen, 2015) for evaluating AQM algorithms. Besides analyzing the trade-off between latency and link utilization, we compare the fairness achieved with PIE and Minstrel PIE to show that the latter does not deteriorate the fairness among the flows. Fairness with respect to link utilization has been measured using a bounded measuring index called Jain's Fairness Index (Jain et al., 1998). It is defined as in Eq. (4.1):

$$f(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{\sum_{i=1}^{n} x_i^2} \tag{4.1}$$

where,

- x_i is the bandwidth dedicated to flow i
- n is the number of flows sharing the bandwidth

Higher values indicate better fairness. An index of 0.4 means that the network is unfair to 60% of the users sharing the link. The maximum value attainable by the index is 1.

4.3.1 Preliminary Evaluation

The preliminary evaluation consisted of two types of scenarios: (i) simulations with different number of responsive and unresponsive flows, and (ii) simulations in realistic network conditions, with a proper mix of responsive and unresponsive flows. The scenarios mentioned in subsections A to C are same as the ones described in Section 4.1, except that we represent the results in two additional forms: distribution of bottleneck link utilization and queue delay over time, and goodput as a function of queue delay which is recommended in RFC 7928. ns-2.36.rc1 has been used for these simulations.

A Light TCP traffic

This scenario consists of 5 TCP flows starting at the same time in a dumbbell topology as shown in Figure 4.1. Table 4.1 presents the configuration of simulation parameters for this scenario. Figure 4.4 shows the performance of PIE and Minstrel PIE in terms of

Table 4.1: Simulation Configuration for Preliminary Evaluation

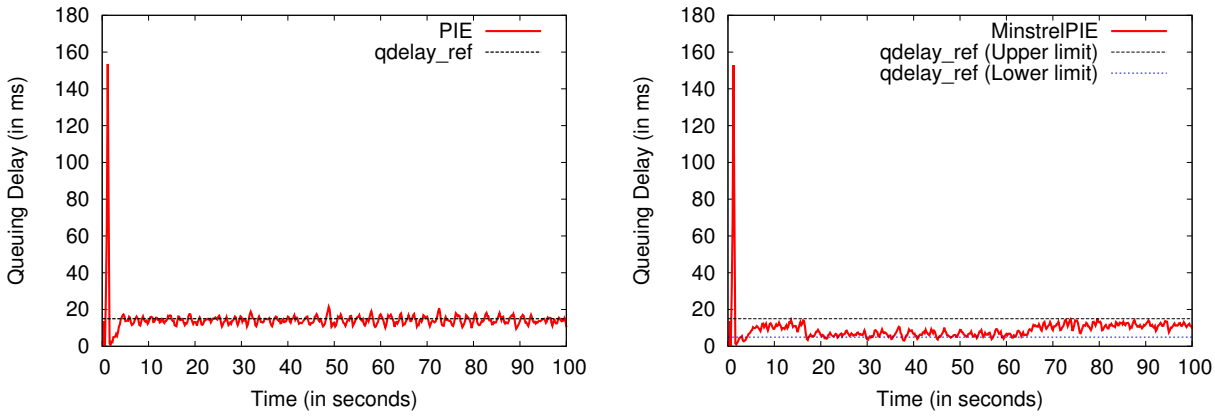
Parameters	**Values**	**Parameters**	**Values**
Bottleneck Propagation Delay	80 ms	Traffic Flows Direction	Forward Only
Non-bottleneck Propagation Delay	20 ms	AQM enabled in Forward path	PIE/Minstrel PIE
Bottleneck Bandwidth	10 Mbps	TCP Application	Long-lived FTP
Non Bottleneck Bandwidth	100 Mbps	UDP Application with rate	Constant Bit Rate at 10Mbps
Buffer Size	200 packets	TCP enabled	Compound
Packet Size	1000 Bytes	Simulation Time	100 Seconds

queue delay and Figure 4.5 shows the performance of PIE and Minstrel PIE in terms of bottleneck link utilization.

Table 4.2: Fairness for Light TCP traffic scenario

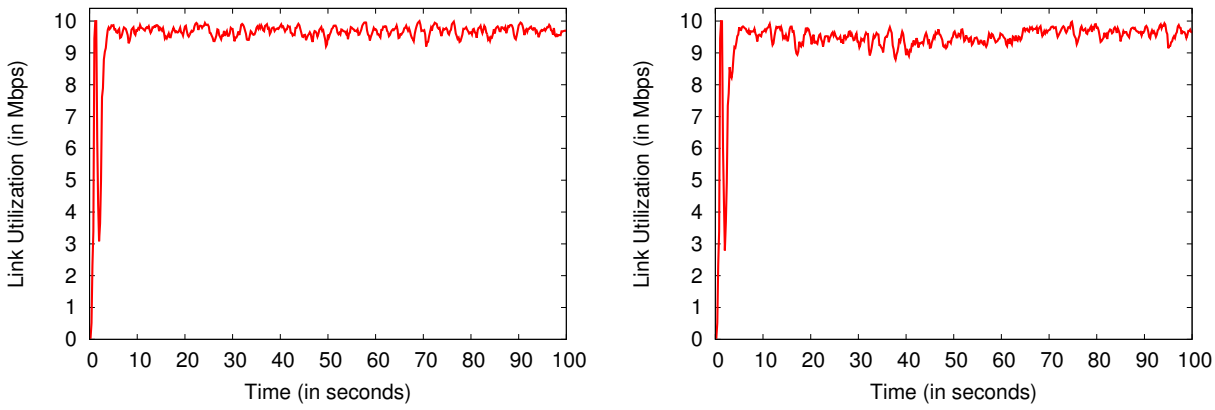
Flow Number	**Throughput (in Mbps)**					**Fairness**
	TCP 1	**TCP 2**	**TCP 3**	**TCP 4**	**TCP 5**	
PIE	1.84	1.83	1.87	1.85	1.85	**0.99**
Minstrel PIE	1.83	1.84	1.80	1.80	1.83	**0.99**

The sharp rise in queue delay in Figure 4.4 and the sudden fall in the link utilization in Figure 4.5 is because all flows start at the same time. This configuration provides an insight into the capabilities of PIE and Minstrel PIE to handle the burst of traffic. Both algorithms handle the burst appropriately and maintain the queue delay around the reference values for the rest of the simulation. Minstrel PIE has marginally better control over the queue delay with same link utilization (Figure 4.4(a) and Figure 4.4(b)) as that of PIE. This is because Minstrel PIE tries to reduce the queue delay to the lower limit (seen around 15 seconds in Figure 4.4(b)) since the link is sufficiently utilized, and allows the queue delay to grow towards the upper limit (seen around 65 seconds in Figure 4.4(b)) when it senses that the link utilization is getting affected (15 seconds to 65 seconds in Figure 4.5(b)). Furthermore, the results shown in Table 4.2 confirm that Minstrel PIE



(a) PIE (b) Minstrel PIE

Figure 4.4: Queuing Delay for Light TCP traffic



(a) PIE (b) Minstrel PIE

Figure 4.5: Link Utilization for Light TCP traffic

does not affect the fairness among the flows.

B Heavy TCP traffic

This scenario sets up 50 TCP flows in a dumbbell topology, but otherwise, it is same as the previous scenario. Figure 4.6(a) and Figure 4.6(b) show the performance of PIE and Minstrel PIE in terms of queue delay, respectively and Figure 4.7(a) and Figure 4.7(b) show the performance of PIE and Minstrel PIE in terms of bottleneck link utilization.

Like Figure 4.4, there is a sharp rise in queue delay in Figure 4.6. Even though the amount of burst in this scenario is much larger than the previous one, it is observed that both algorithms successfully control the queue delay. After the burst period, PIE maintains the queue delay around the *qdelay_ref* whereas Minstrel PIE maintains the queue delay between the lower and upper limit. Since there are sufficient number of TCP

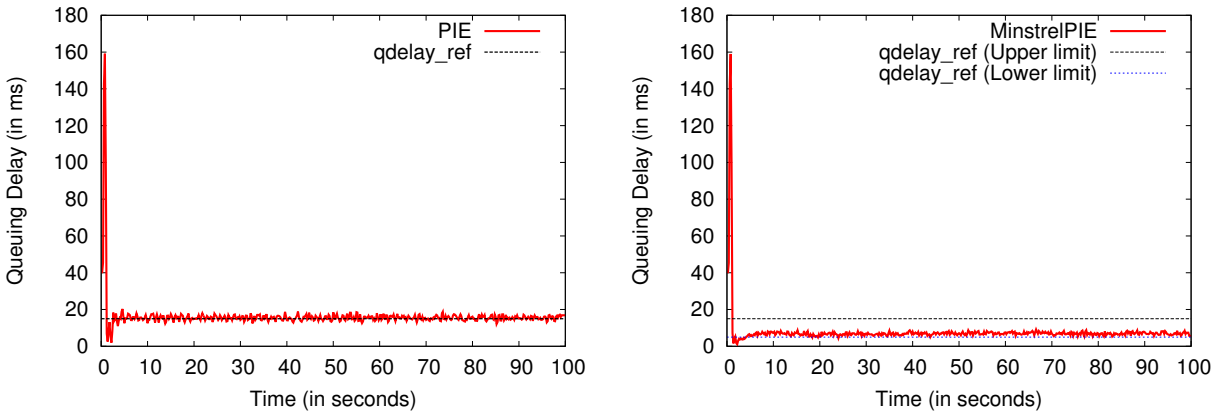
Table 4.3: Fairness for Heavy TCP traffic scenario

Flow Number	**Throughput (in Mbps)**					**Fairness**
	TCP 1-10	**TCP 11-20**	**TCP 21-30**	**TCP 31-40**	**TCP 41-50**	
PIE	0.19	0.19	0.19	0.19	0.18	**0.98**
	0.19	0.18	0.18	0.18	0.19	
	0.19	0.19	0.19	0.19	0.19	
	0.18	0.18	0.19	0.19	0.19	
	0.18	0.19	0.19	0.19	0.19	
	0.19	0.18	0.19	0.19	0.19	
	0.18	0.19	0.18	0.19	0.19	
	0.19	0.19	0.19	0.19	0.19	
	0.18	0.19	0.19	0.19	0.19	
	0.19	0.19	0.19	0.19	0.19	
Minstrel PIE	0.18	0.19	0.17	0.19	0.18	**0.99**
	0.19	0.18	0.19	0.18	0.19	
	0.17	0.19	0.18	0.19	0.18	
	0.19	0.19	0.19	0.19	0.19	
	0.18	0.19	0.19	0.19	0.19	
	0.18	0.18	0.19	0.18	0.19	
	0.18	0.19	0.18	0.19	0.19	
	0.19	0.18	0.19	0.18	0.19	
	0.18	0.19	0.18	0.19	0.18	
	0.19	0.19	0.18	0.19	0.19	

flows to keep the link fully utilized, Minstrel PIE tries to maintain the queue delay near to its lower bound of *qdelay_ref*. In terms of link utilization and fairness among the flows (See Table 4.3), both algorithms have similar performance.

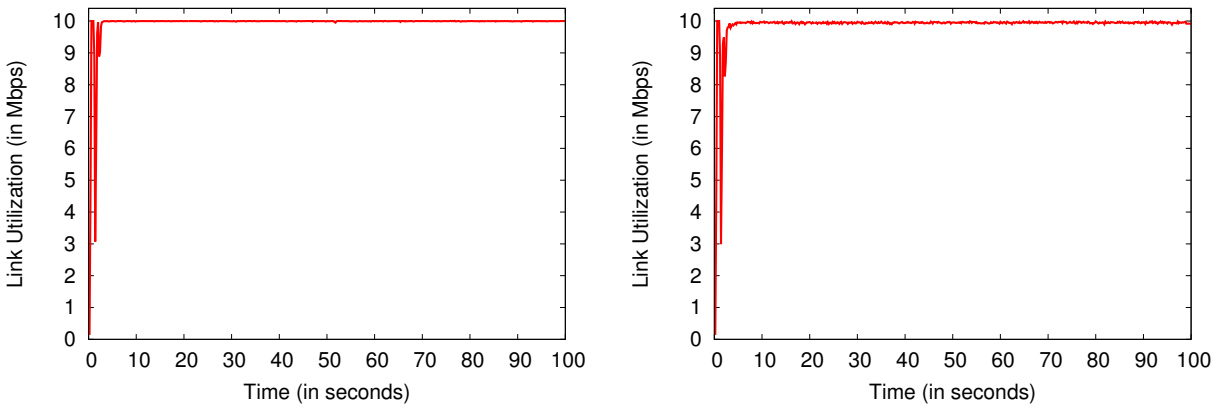
C Mix TCP and UDP traffic

This is an extension of the Light TCP traffic scenario, where 2 UDP flows are added to 5 TCP flows. The goal is to verify the ability of Minstrel PIE mechanism to control queue delay in the presence of unresponsive flows. Figure 4.8(a) and Figure 4.8(b) show the performance of PIE and Minstrel PIE in terms of queue delay, respectively and Figure 4.9(a) and Figure 4.9(b) show the performance of PIE and Minstrel PIE in terms of bottleneck link utilization.



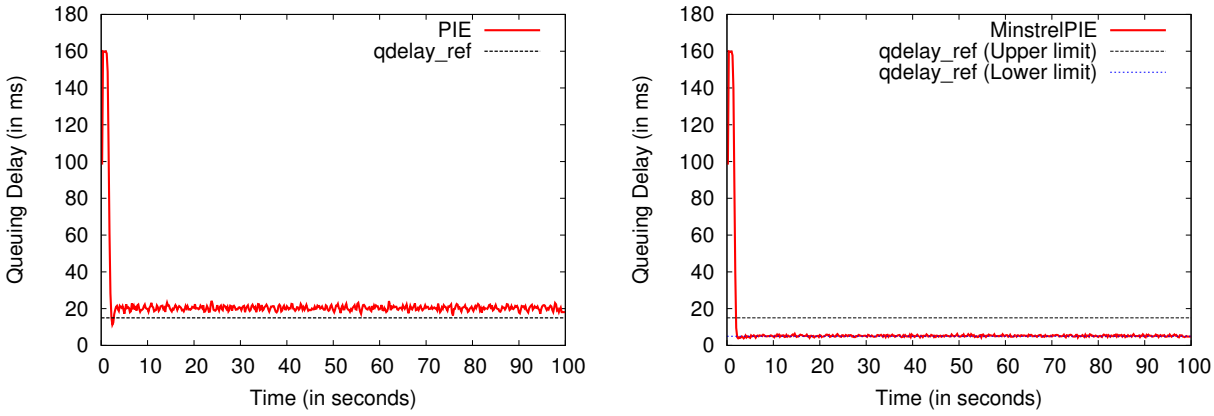
(a) PIE (b) Minstrel PIE

Figure 4.6: Queuing Delay for Heavy TCP traffic



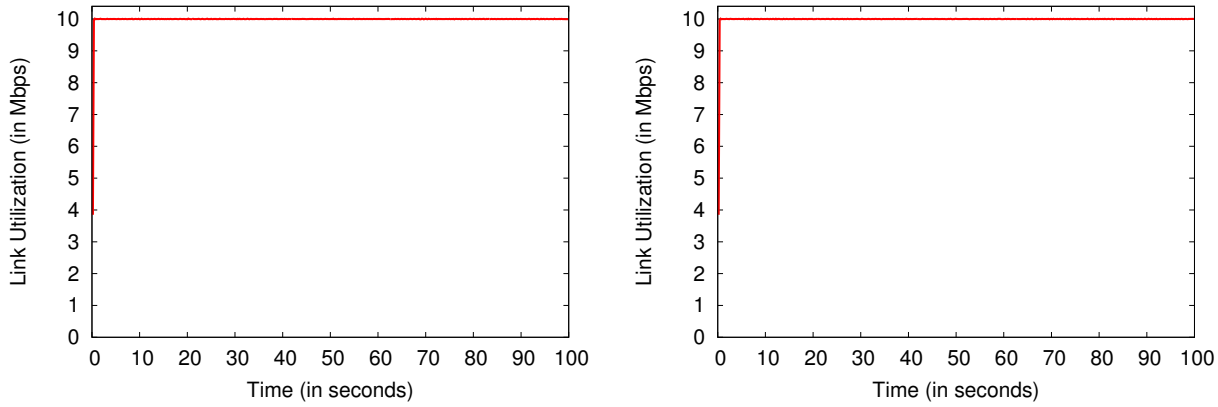
(a) PIE (b) Minstrel PIE

Figure 4.7: Link Utilization for Heavy TCP traffic



(a) PIE (b) Minstrel PIE

Figure 4.8: Queuing Delay for Mix TCP and UDP traffic



(a) PIE (b) Minstrel PIE

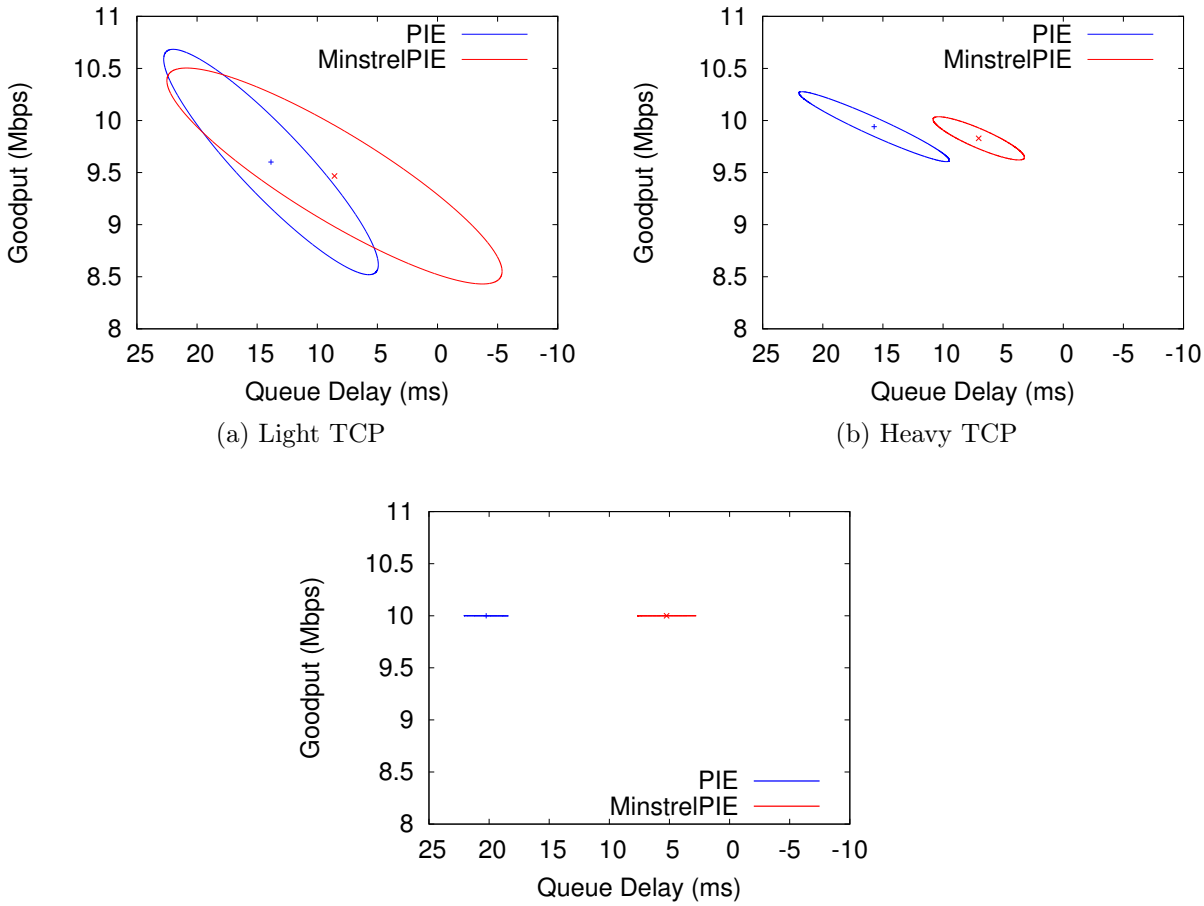
Figure 4.9: Link Utilization for Mix TCP and UDP traffic

Table 4.4: Fairness for Mix TCP and UDP traffic scenario

Flow Number	**Throughput (in Mbps)**							**Fairness**
	TCP 1	**TCP 2**	**TCP 3**	**TCP 4**	**TCP 5**	**UDP 1**	**UDP 2**	
PIE	0.010	0.015	0.006	0.007	0.014	4.790	4.729	0.288
Minstrel PIE	0.010	0.015	0.006	0.007	0.020	4.788	4.729	0.289

Although the initial burst of traffic is successfully handled by both algorithms, PIE is unable to maintain the queue delay within its *qdelay_ref* for the rest of the simulation. On the other hand, Minstrel PIE consistently maintains the queue delay around its lower bound of *qdelay_ref* without hurting the link utilization. Since the link utilization consistently remains at its peak (100%), Minstrel PIE senses an opportunity to minimize the queue delay as much as possible. A low value in the 'Fairness' column in Table 4.4 for both algorithms highlights the unfairness problem due to the unresponsive nature of UDP flows. This problem can be addressed by combining flow queuing, without compromising on the benefits offered by Minstrel PIE. Chapter 5 provides more details on flow queuing.

RFC 7928 recommends plotting the goodput as a function of queue delay to compare the performance of AQM algorithms. Figure 4.10 depicts the results of above three scenarios as per the RFC 7928 guidelines. An ellipse represents the contour with maximum likelihood 2D Gaussian distribution of goodput and delay. The height and width of the ellipse are a function of standard deviation of goodput and delay, respectively. The orientation of an ellipse is based on the covariance between the two. Since the center of



(a) Light TCP

(b) Heavy TCP

(c) Mix TCP and UDP

Figure 4.10: Representation of Figure 4 to Figure 9 as per RFC 7928

an ellipse represents the mean with equal extension of height and width on both sides, it is possible to get part of the contour in the region with negative queue delay or with goodput greater than bottleneck link capacity. The area inside an ellipse can be interpreted as the variance in the values of goodput and delay. The more the variations, the larger is the outline of an ellipse (Winstein and Balakrishnan, 2013). The results show that Minstrel PIE maintains a better tradeoff, especially when responsive and unresponsive traffic coexist.

4.3.2 RFC 7928 based Evaluation

Thorough evaluation of an AQM mechanism requires significant time and efforts. Hence, the AQM and Packet Scheduling Working Group at IETF published RFC 7928 which provides the guidelines to evaluate AQM algorithms. An automated evaluation framework complying with the guidelines of RFC 7928 has been recently developed for ns-3 (Deepak et al., 2017). We use this evaluation suite to verify the desired behavior of Minstrel PIE

Table 4.5: Simulation Configuration for RFC 7928 based Evaluation

Parameters	**Values**	**Parameters**	**Values**
Bottleneck Propagation Delay	80ms	Traffic Flows Direction	Forward Only
Non-bottleneck Propagation Delay	20ms	AQM enabled in Forward path	PIE/Minstrel PIE
Bottleneck Bandwidth	10 Mbps	TCP Application	Long-lived FTP
Non Bottleneck Bandwidth	100 Mbps	UDP Application with rate	Constant Bit Rate at 10Mbps
Buffer Size	200 packets	TCP enabled	NewReno
Packet Size	1000 Bytes	Simulation Time	300 Seconds

and its relative performance against PIE under various traffic conditions. While RFC 7928 recommends several test scenarios to evaluate AQM algorithms, we present the results for the ones that are most relevant to the objective of designing Minstrel PIE. Table 4.5 presents the simulation configuration used by the AQM Evaluation Suite of ns-3. The same has been used for the evaluation of Minstrel PIE.

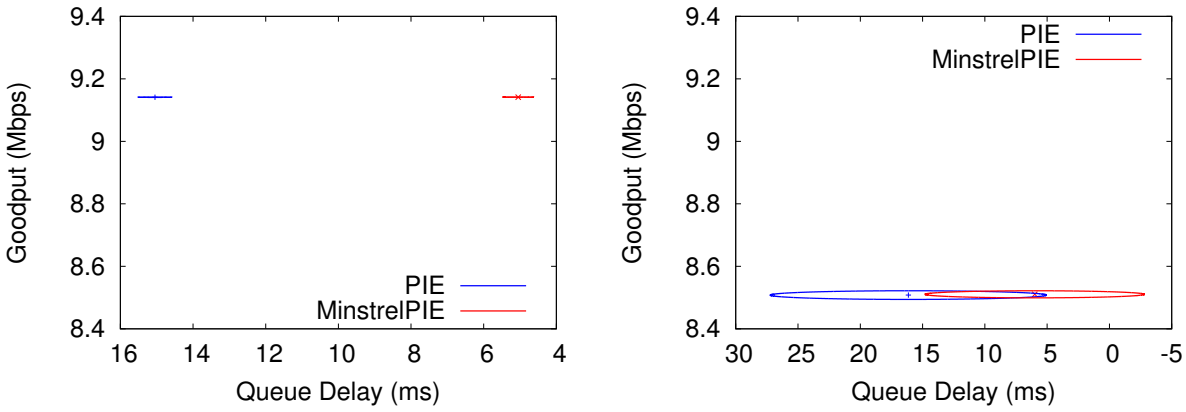
A Unresponsive transport sender

Table 4.6: Scenarios mentioned in Section 5.3 of RFC 7928

Traffic Scenario		**Description**
UDP Transport Sender	Mix TCP and UDP	Consists of a long-lived UDP flow from an unresponsive application with a single long-lived, non-application limited TCP NewReno flow.
	With single UDP Sender	Consists of a non-application limited and long-lived, single UDP flow with sending rate more than bottleneck capacity.

The scenarios mentioned in this section are identical to the ones mentioned in Section 5.3 of RFC 7928. The main idea is to evaluate the performance of AQM algorithms in the presence of unresponsive transport. Table 4.6 presents a summary of the scenarios.

Figure 4.11 presents the results for the traffic scenarios mentioned in Table 4.6. Min-



(a) UDP with TCP Mix

(b) Single UDP Sender

Figure 4.11: Results for Section 5.3 from RFC 7928

Table 4.7: Fairness for Section 5.3 traffic scenario from RFC 7928

Flow Number	**Throughput (in Mbps)**		**Fairness**
	TCP 1	**UDP 1**	
PIE	0.063	8.983	0.50
Minstrel PIE	0.074	9.171	0.50

strel PIE shows a tight control on the queue delay in both the scenarios, more so in the case of a single unresponsive flow. We observe that both algorithms provide a stable performance (i.e., variations in queue delay and link utilization are negligible in Figure 4.11(b)) because a single UDP flow has been configured with Constant Bit Rate (CBR) application, which sends the data at a constant rate. The utilization and fairness (See Table 4.7) obtained with both the algorithms is similar. These results are in line with those discussed in the previous section.

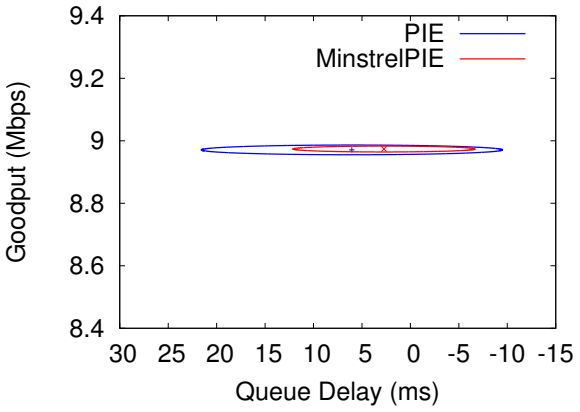
B Levels of network congestion

RFC 7928 specifies three scenarios with different levels of congestion varying from mild, medium to heavy congestion, respectively. The main idea is to evaluate the performance of AQM algorithms in different levels of network congestion. Table 4.8 resents a summary of the scenarios. The details of each scenario configuration can be found in Section 8.2.2, 8.2.3 and 8.2.4 of RFC 7928, respectively.

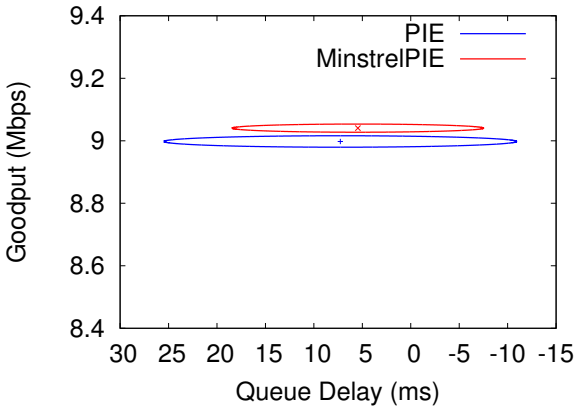
Since these scenarios configure only responsive flows to vary the amount of congestion in the network, Minstrel PIE and PIE are expected to provide similar performance. Figure

Table 4.8: Congestion scenarios mentioned in Section 8 of RFC 7928

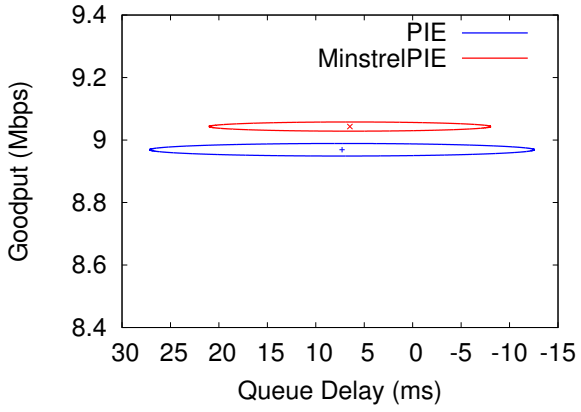
Congestion Scenarios		Description
1	Mild Congestion	Consists of bulk non-application limited TCP flow which can generate packet drop rate of 0.1% at the router.
2	Medium Congestion	Consists of bulk non-application limited TCP flow which can generate packet drop rate of 0.5% at the router.
3	Heavy Congestion	Consists of bulk non-application limited TCP flow which can generate packet drop rate of 1% at the router.



(a) Mild Congestion



(b) Medium Congestion



(c) Heavy Congestion

Figure 4.12: Results for Section 8.2.2 to 8.2.4 from RFC 7928

4.12(a), Figure 4.12(b) and Figure 4.12(c) depict the results for above mentioned scenarios, respectively.

4.3.3 Evaluation using Flent

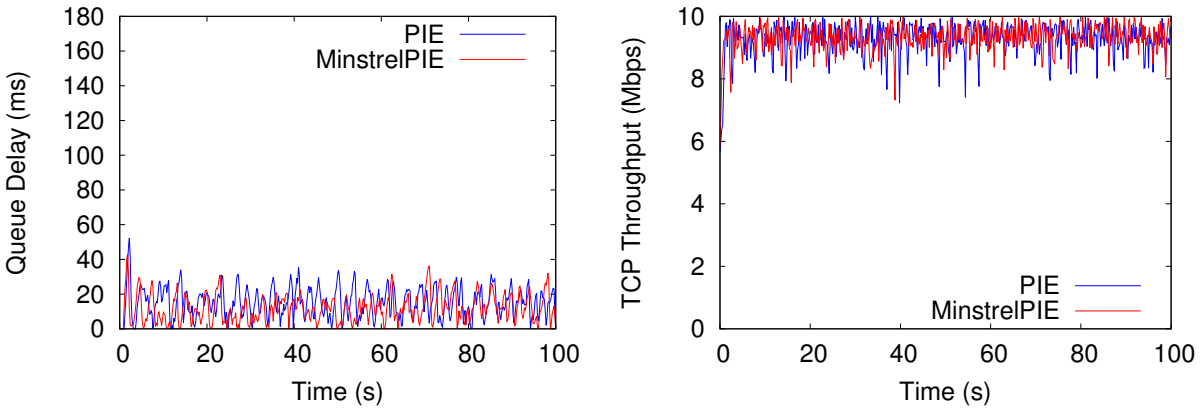
To verify the effectiveness of Minstrel PIE in a real network environment, we conducted experiments on a testbed by using Flent. Flent is developed to overcome the problems of coordination among different testing tools such as *netperf* (Jones et al., 1996) and *iperf* (Tirumala et al., 2005), reproducing testbed results, managing testbed configuration, and storing and analysing measurement data (Høiland-Jørgensen, 2015). It provides test scenarios to evaluate the performance of AQM algorithms. Flent's TCP upload (tcp_up) test has been used to evaluate the performance of Minstrel PIE. The testbed comprises of 1 sender, 1 receiver and 1 AQM machine, all running Ubuntu 16.04 LTS with Linux kernel 4.15. Minstrel PIE is implemented only on the AQM machine. Flent specific parameters settings and traffic loads are listed in Table 4.9. In addition to evaluating Minstrel PIE in basic scenarios like light and heavy TCP traffic, we evaluate it in realistic scenarios like: on-off unresponsive UDP flow, ECN enabled TCP flows and different bottleneck buffer sizes.

Table 4.9: Testbed Setup using ethtool, netem, tc and Flent

Parameters	**Values**	**Parameters**	**Values**
Bottleneck Propagation Delay	80 ms	Traffic Flows Direction	Forward Only
Non-bottleneck Propagation Delay	20 ms	AQM enabled in Forward path	PIE/Minstrel PIE
Bottleneck Bandwidth	10 Mbps	TCP Application	Long-lived FTP
Non Bottleneck Bandwidth	100 Mbps	UDP Application with rate	Constant Bit Rate at 10Mbps
Buffer Size	200 packets	TCP enabled	CUBIC (RFC 8312)
Packet Size	1000 Bytes	Simulation Time	100 Seconds

A Light TCP traffic

This scenario is identical to the one discussed in subsection A of Section 4.3.1 and consists of 5 TCP flows that start at the same time in a dumbbell topology. Figure 4.13 (a) and Figure 4.13 (b) compare the performance of PIE and Minstrel PIE in terms of queue



(a) Queue Delay (b) TCP Throughput

Figure 4.13: Light TCP Traffic

Table 4.10: Fairness for Light TCP traffic scenario in testbed

Flow Number	**Throughput (in Mbps)**					**Fairness**
	TCP 1	**TCP 2**	**TCP 3**	**TCP 4**	**TCP 5**	
PIE	2.13	2.01	1.73	1.88	1.75	**0.99**
Minstrel PIE	1.94	1.94	1.93	1.84	1.83	**0.99**

delay and bottleneck link utilization, respectively. The slight rise initially in queue delay in Figure 4.13 (a) is because all flows start at the same time. Both algorithms have similar performance since all the flows are responsive flows and moreover, the number of flows is less. Additionally, Table 4.10 shows that the fairness among flows remains same with both the algorithms.

B Heavy TCP traffic

This scenario is identical to the one discussed in subsection B of Section 4.3.1 and sets up 50 TCP flows in a dumbbell topology, but otherwise, it is same as the previous scenario. Figure 4.14 (a) and Figure 4.14 (b) compare the performance of PIE and Minstrel PIE in terms of queue delay and bottleneck link utilization, respectively. Queue delay is initially higher than the previous scenario because the number of flows is increased to 50 and all start at the same time. Both algorithms control queue delay appropriately for the rest of the experiment. PIE maintains the queue delay around its *qdelay_ref* and Minstrel PIE maintains it between the lower and upper limits of *qdelay_ref*. The bottleneck link remains fully utilized in Figure 4.14 (b) due to a large number of TCP flows, and both

Table 4.11: Fairness for Heavy TCP traffic scenario in testbed

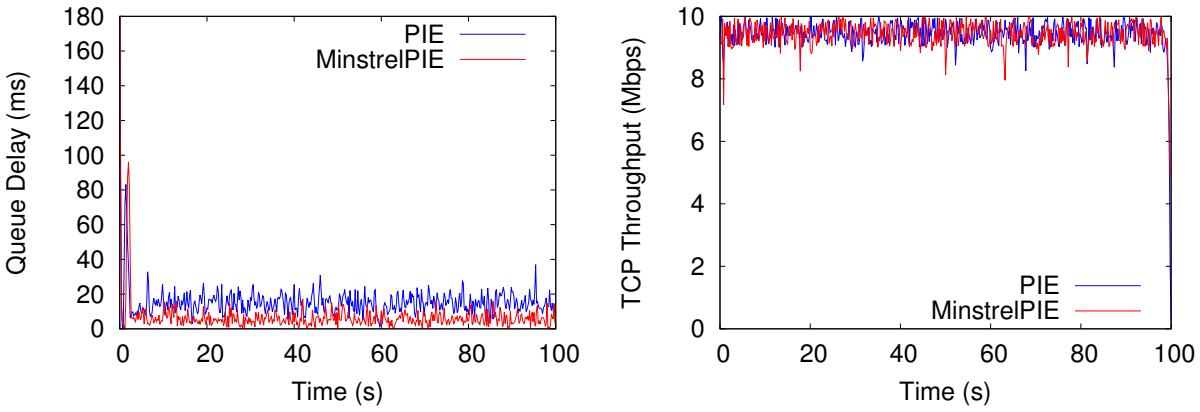
Flow Number	Throughput (in Mbps)					Fairness
	TCP 1-10	TCP 11-20	TCP 21-30	TCP 31-40	TCP 41-50	
PIE	0.18	0.18	0.21	0.17	0.21	0.99
	0.18	0.21	0.18	0.17	0.16	
	0.18	0.20	0.18	0.18	0.19	
	0.19	0.21	0.20	0.18	0.22	
	0.19	0.19	0.19	0.18	0.18	
	0.19	0.21	0.18	0.19	0.20	
	0.20	0.18	0.22	0.18	0.21	
	0.18	0.16	0.20	0.19	0.19	
	0.18	0.20	0.20	0.17	0.21	
	0.19	0.21	0.19	0.20	0.21	
Minstrel PIE	0.21	0.21	0.18	0.2	0.19	0.99
	0.16	0.17	0.22	0.18	0.18	
	0.18	0.19	0.21	0.20	0.21	
	0.19	0.18	0.20	0.19	0.20	
	0.21	0.17	0.21	0.21	0.19	
	0.19	0.16	0.17	0.19	0.19	
	0.19	0.18	0.20	0.19	0.19	
	0.22	0.19	0.17	0.19	0.19	
	0.20	0.18	0.18	0.20	0.20	
	0.19	0.18	0.20	0.17	0.19	

algorithms have a similar performance. Furthermore, Table 4.11 confirms that Minstrel PIE does not deteriorate the fairness among flows.

C Benefits of ECN

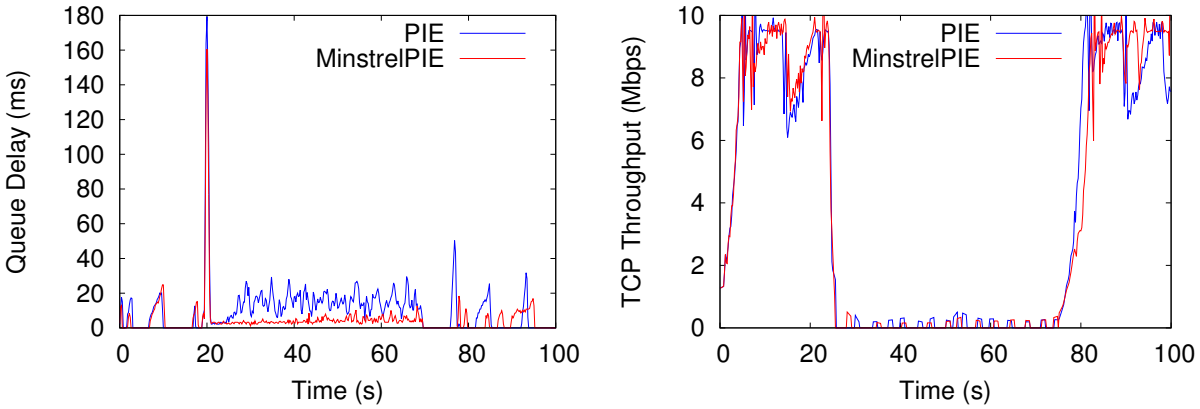
This section highlights the benefits of using ECN with Minstrel PIE. The experiments consist of two traffic configurations: 1-TCP-1-UDP (a single TCP flow with a single UDP flow), and 5-TCP-1-UDP (five TCP flows with a single UDP flow). TCP flows start at 1 second and run until the end of the experiment. UDP flow starts at 25 seconds and stops at 75 seconds. These configurations help us to evaluate the performance of Minstrel PIE when unresponsive flows join and leave the network.

Figure 4.15 (a) and Figure 4.15 (b) compare PIE and Minstrel PIE in terms of queue



(a) Queue Delay (b) TCP Throughput

Figure 4.14: Heavy TCP Traffic



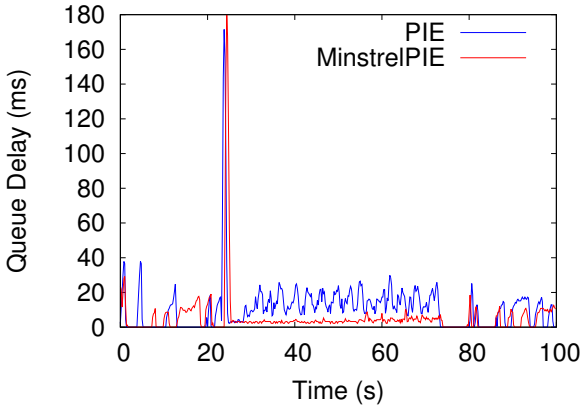
(a) Queue Delay (b) TCP Throughput

Figure 4.15: Mix TCP and UDP Traffic with tcp_1up

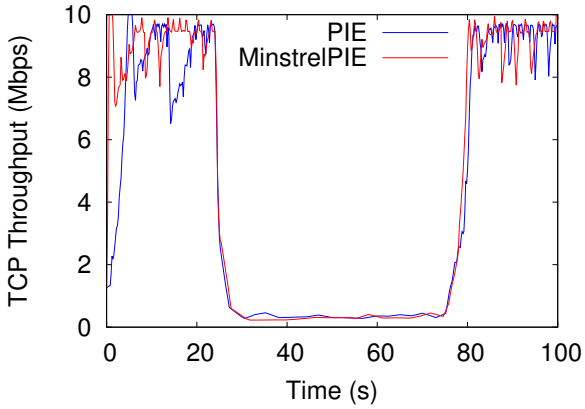
delay and TCP throughput without enabling ECN in a 1-TCP-1-UDP configuration, respectively. Large oscillation in queue delay around 25 seconds is due to the start of UDP flow. Minstrel PIE observes the increase in the *avg_dq_rate* and accordingly reduces the *qdelay_ref* to achieve a tight control on queue delay. Later, when UDP flow leaves the network at 75 seconds, Minstrel PIE increases the *qdelay_ref*. Figure 4.15 (b) shows the impact on the throughput of TCP flow when UDP flow starts. TCP throughput reduces significantly when UDP flow starts, and gets back to normal when UDP flow stops. The results in Table 4.12 show the throughput share obtained by TCP and UDP flows. We repeat this experiment by enabling PIE and Minstrel PIE to use ECN, and the results are depicted in Figure 4.16 (a) and Figure 4.16 (b). It is observed that TCP throughput improves marginally (See Table 4.13), but still gets largely affected when UDP flow is on.

Table 4.12: Fairness in tcp_1up test without ECN

Flow Number	Throughput (in Mbps)		Fairness
	TCP 1	UDP 1	
PIE	0.083	9.60	0.50
Minstrel PIE	0.064	9.61	0.50

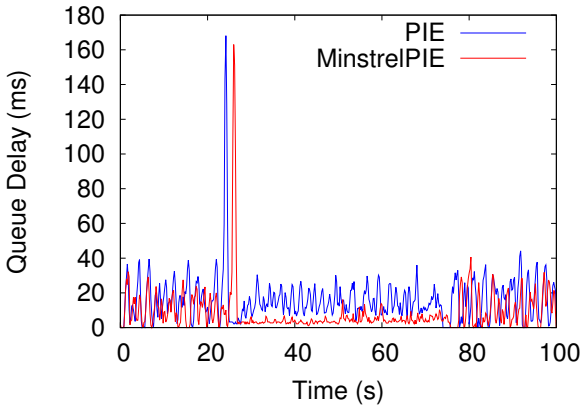


(a) Queue Delay with ECN

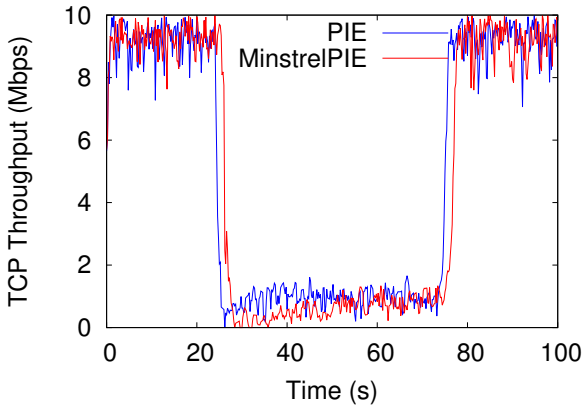


(b) TCP Throughput with ECN

Figure 4.16: Mix TCP and UDP Traffic with tcp_1up



(a) Queue Delay



(b) TCP Throughput

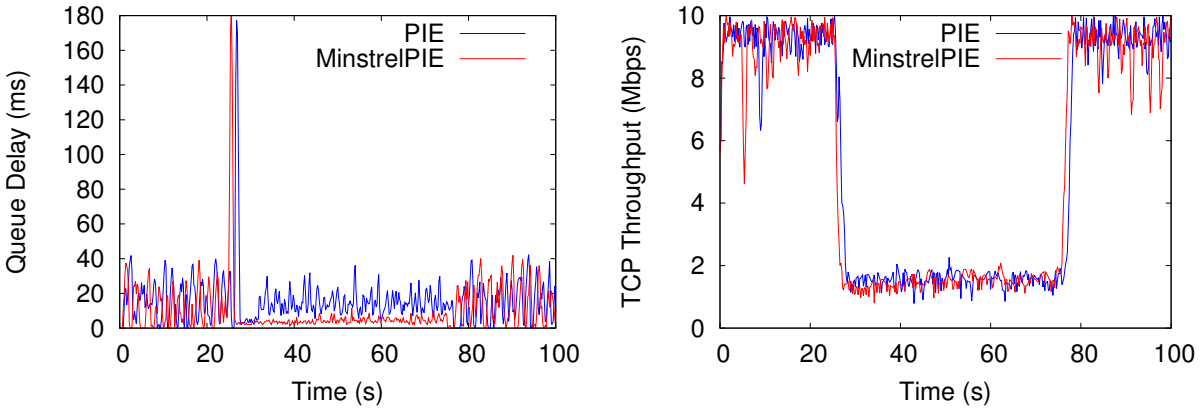
Figure 4.17: Mix TCP and UDP Traffic

Table 4.13: Fairness in tcp_1up test with ECN

Flow Number	Throughput (in Mbps)		Fairness
	TCP 1	UDP 1	
PIE	0.13	9.54	0.51
Minstrel PIE	0.11	9.56	0.51

Table 4.14: Fairness in tcp_5up test without ECN

Flow Number	Throughput (in Mbps)						Fairness
	TCP 1	TCP 2	TCP 3	TCP 4	TCP 5	UDP 1	
PIE	0.19	0.16	0.18	0.18	0.22	8.50	**0.24**
Minstrel PIE	0.17	0.17	0.18	0.16	0.17	8.60	**0.24**



(a) Queue Delay with ECN

(b) TCP Throughput with ECN

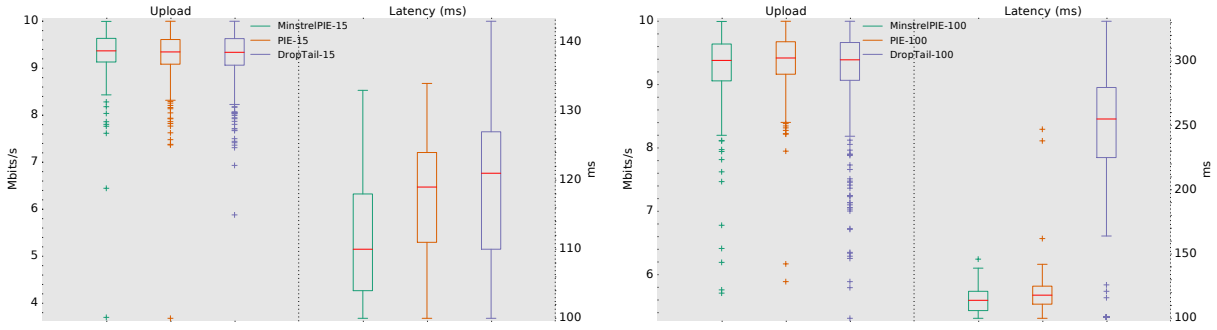
Figure 4.18: Mix TCP and UDP Traffic with ECN

Table 4.15: Fairness in tcp_5up test with ECN

Flow Number	Throughput (in Mbps)						Fairness
	TCP 1	TCP 2	TCP 3	TCP 4	TCP 5	UDP 1	
PIE	0.32	0.29	0.30	0.34	0.31	8.30	**0.28**
Minstrel PIE	0.31	0.30	0.31	0.32	0.32	8.32	**0.28**

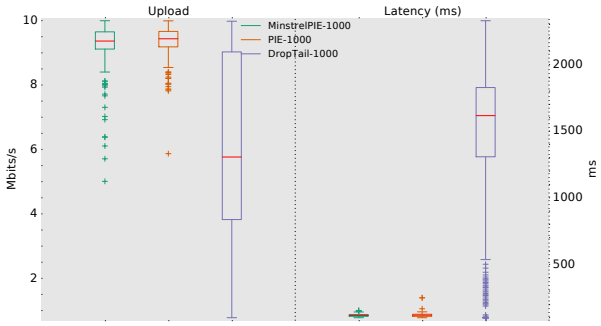
by ECN can be observed by comparing the results shown in Table 4.14 and 4.15.

D Evaluation with different buffer sizes



(a) Buffer 15

(b) Buffer 100



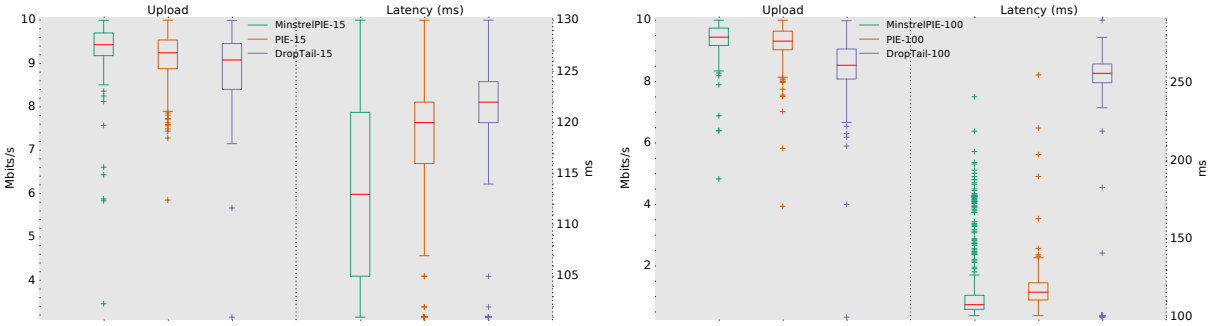
(c) Buffer 1000

Figure 4.19: Light TCP Traffic with TCP CUBIC

A significant amount of work has been done to study the impact of bottleneck buffer size on queue delay and link utilization (Wischik and McKeown, 2005; Raina et al., 2005; Enachescu et al., 2005). In this scenario, we repeat the experiments described in Section 4.3.1 and 4.3.2 by configuring different buffer sizes at the bottleneck. We use the standard buffer size of 15, 100 and 1000 packets (Raina and Wischik, 2005; Raina et al., 2016) in both the cases. In addition, we also compare the results of PIE and Minstrel PIE with the traditional tail drop queues. Figure 4.19 and Figure 4.20 depict the results obtained with light TCP traffic and heavy TCP traffic, respectively.

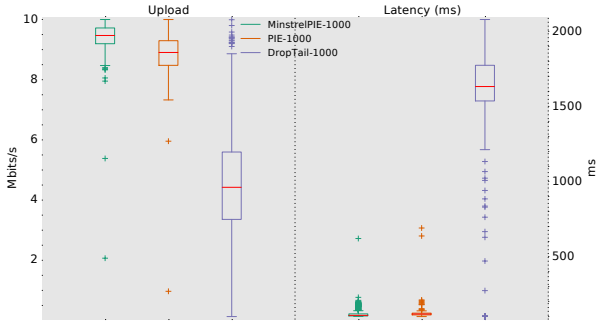
When the buffer size is 15 packets, it is observed that the RTT is marginally lesser with PIE and Minstrel PIE than tail drop queues. As the buffer size increases to 100 and 1000 packets, PIE and Minstrel PIE offer significant advantage over tail drop queues in terms of RTT.

The link utilization is similar for all three algorithms when the buffer size is 15 packets



(a) Buffer 15

(b) Buffer 100



(c) Buffer 1000

Figure 4.20: Heavy TCP Traffic with TCP CUBIC

and the traffic is light. However, the link utilization is marginally affected with tail drop queues when heavy traffic is used with a buffer size of 15 packets. This happens because the tail drop queues cause the large number of TCP flows passing through this small buffer to synchronize. The impact of global synchronization on the overall link utilization is also visible in the cases where buffer size is 100, more so with 1000 packets because the congestion window of TCP flows grow to a large value. The link utilization with PIE and Minstrel PIE is better in all the cases. In general, Minstrel PIE provides a better performance trade-off than PIE and tail drop queues.

4.4 Inferences

This chapter proposed an extension to the PIE AQM mechanism, called Minstrel PIE. It adapts the *qdelay_ref* to achieve a better trade-off between queue delay and link utilization. The results obtained from the simulation studies and real time experiments validate the effectiveness and robustness of Minstrel PIE against unresponsive traffic. Minstrel PIE can be incrementally deployed in a real network setup as it is a minor modification of PIE, and does not require the user to configure any parameter explicitly.

Furthermore, to maintain the fairness between responsive and unresponsive flows there is a need of FQ-Minstrel PIE. The detailed implementation and evaluation of FQ-Minstrel PIE is described in Chapter 5.

Chapter 5

Flow Queue Minstrel PIE

Typically, AQM mechanisms are expected to avoid congestion and minimize the impact of bufferbloat, whereas, protecting the fair share of responsive flows when they coexist with unresponsive flows is the responsibility of packet scheduling mechanisms (e.g., Stochastic Fair Queuing (SFQ) (McKenney, 1990), Deficit Round Robin (DRR) (Shreedhar and Varghese, 1996)). Minstrel PIE, being an AQM mechanism, is mainly designed to avoid network congestion and minimize the impact of bufferbloat. Recently, there has been an interest in designing hybrid packet scheduler and AQM mechanisms to collectively address the problems of congestion, bufferbloat and flow protection by giving each flow its own queue. Flow Queue CoDel (FQ-CoDel) (RFC 8290) and Flow Queue PIE (FQ-PIE) (Al-Saadi and Armitage, 2016) are two attempts in this direction. Along similar lines, we combine Minstrel PIE with flow queuing to provide flow protection and name the resulting mechanism as Flow Queue Minstrel PIE (FQ-Minstrel PIE). This chapter first provides the implementation details of FQ-PIE for the Linux kernel. Subsequently, the design and implementation of FQ-Minstrel PIE is discussed followed by results and analysis.

5.1 Flow Queuing

Flow Queuing (FQ) is intended to provide each flow with its own queue. The advantage of such a mechanism is that it protects responsive TCP flows from the impact of non-responsive flows such as Constant Bit Rate (CBR) multimedia traffic and also protects the thin streams e.g., interactive multimedia in presence of bulk TCP traffic (Khademi et al., 2013).

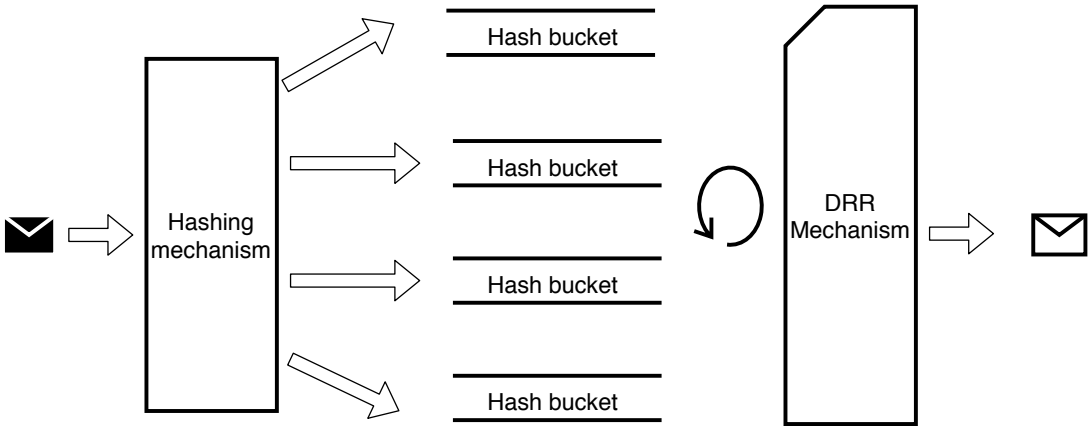


Figure 5.1: Flow Queue mechanism

Ideally, each flow is intended to have a separate queue. However, in practice, a hashing based scheme is used where flows are hashed into buckets. The Jenkin's hash function (Yamaguchi and Nishi, 2013) is used to calculate the hash value of a flow and hashing is dependent on 5-tuples: source IP address, destination IP addresses, source port number, destination port number and protocol number (RFC 8290).

Deficit Round Robin (DRR) (Shreedhar and Varghese, 1996) is a popular packet scheduling mechanism used for flow queuing. Each bucket is assigned a certain number of 'byte credits'. When a packet is dequeued from a bucket, the byte credits reduces by the packet size. This process is repeated until the byte credits reach zero, after which byte credits for the bucket is reset and the dequeue process is stopped for that bucket.

5.2 FQ-PIE

The pre-requisite to design and implement FQ-Minstrel PIE is to evaluate the performance of FQ-PIE. However, a model to evaluate FQ-PIE is missing in network simulators and the Linux kernel. Hence, this work initially focused on implementing FQ-PIE in the Linux kernel.

5.2.1 Design

FQ-PIE is implemented as a queue discipline (qdisc) in the Linux kernel. A qdisc defines how a packet has to be enqueued and dequeued from the buffers. The qdisc structure contains a queue along with references to the queue discipline methods. Our implementation is motivated by the design of the FQ-CoDel mechanism in the Linux kernel. Linux v5.1-rc1 has been used for implementing FQ-PIE.

5.2.2 Implementation

A flow is implemented by the `fq_pie_flow` struct as shown in Figure 5.2. The implementation uses the Jenkins hash function as the hashing mechanism, which is available in the Linux kernel. By default, 1024 hash buckets are created. This can be modified by changing the `flows` variable through the `tc` command of the iproute2 package.

Unlike plain DRR, FQ-PIE implements two sets of flows known as `new_flows` and `old_flows`. This is done so that new flows get priority over existing flows. When an empty bucket is enqueued with a packet, it is added to the end of `new_flows` list and given a `QUANTUM` set of credits. When a bucket is chosen for dequeue, packets are dequeued from the head of the bucket. The value of credits for the particular bucket is decreased by the size of the packet which is being dequeued. This process is continued until the credits for the bucket becomes zero or negative. After the credits are exhausted by the DRR mechanism, the dequeue mechanism ends for that bucket and it is added to the end of `old_flows` list. When a flow is exhausted of all it's packets, it is removed from the `old_flows` list. To protect against overload of packets, there is a configurable limit (`LIMIT`) to the total number of packets that can be enqueued across all flows.

The `list_head` flowchain enables the creation of a list of flows. This is needed for the DRR scheduling of the flows. When a bucket is exhausted of all its credits, that bucket is removed from the head of the flowchain and appended to the end of the flowchain. The qdisc stores two `list_head` data members which hold the list of flows for `old_flows` and `new_flows`. The `list_head` flowchain can be attached to either of these structures.

This scheme of using `new_flows` list and `old_flows` list along with DRR is based on the implementation of FQ-CoDel in the Linux kernel[1].

A Enqueue

The PIE and hence, FQ-PIE mechanisms are proactive during the enqueue phase. This function decides whether a packet must be dropped or enqueued. In FQ-PIE, the mechanism has an additional responsibility of classifying incoming packets into appropriate buckets. The classification is done by `fq_pie_classify()` as shown in line 2 of Algorithm 7 which uses the Jenkins Hash function.

In each bucket, the PIE mechanism is run independently. If the selected bucket is not in the list of old flows or new flows, that bucket is added to the end of list of new flows

[1]https://github.com/torvalds/linux/blob/master/net/sched/sch_fq_codel.c

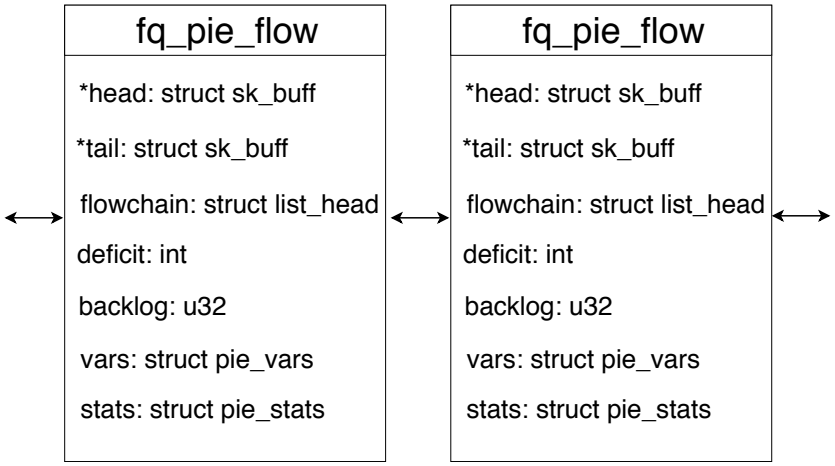


Figure 5.2: List of FQ-PIE flows

Algorithm 7: Enqueue packet algorithm

```
1: procedure ENQUEUE()
2:     idx ←fq_pie_classify()
       if total_packets > LIMIT then
           Drop packet
3:     drop ←drop_early(flows[idx])
       if drop = 0 then
           if empty(flows[idx]) then
               new_flows_add(flows[idx])
4:         flow_queue_add(flows[idx], skb)
5: end procedure
```

and is assigned a certain number of byte credits.

B Dequeue

During the dequeue process, as shown in Algorithm 8, the two round robin lists, *new_flows* and *old_flows* are accessed in a multilevel queue manner. The DRR scheme as mentioned in Section 5.1 is used. The default initial byte credit value is 1 MTU (Maximum Transmission Unit). If the *new_flows* are not empty, a packet from the head of *new_flows* is dequeued. A number of credits equal to the size of the dequeued packet is subtracted from the total credits of that flow. If the selected flow is empty or has exhausted its byte credits, it is appended to the end of *old_flows*. If the *new_flows* are empty, the first flow in *old_flows* is accessed. In the case where the flow exhausts its byte credits, that flow's credits are reinitialized with the default value.

Algorithm 8: Dequeue packet algorithm

```
1:  procedure DEQUEUE()
2:      sel_flow ←head(new_flows)
        if empty(sel_flow) then
3:      sel_flow ← head(old_flows)
        if empty(sel_flow) then
4:          return NULL
        else
5:          deficit ←credits(sel_flow)
        if deficit < 0 then
6:          add_credits(sel_flow)
7:          add_to_tail(old_flows, sel_flow)
8:      skb ←head(sel_flow) if skb = NULL then
            if sel_flow in new_flows then
9:              add_to_tail(old_flows, sel_flow)
            else
10:             remove_flow(old_flows, sel_flow)
11: end procedure
```

C Timer

In PIE, the timer function is used to call the `calculate_probability()` function at regular intervals. In FQ-PIE, it has an additional responsibility of performing the same action for all the queues. The Linux kernel defines a structure called `timer_list`, which can be made to call a function after a selected number of 'jiffies' (a unit of time). It is necessary to lock the qdisc structure when the timer's function is called. This is required because the callback function invoked by the timer object is issued as an asynchronous software interrupt and hence, there is a necessity for atomicity and the whole qdisc is locked from any modification[2]. The locking is done by a spinlock. Every qdisc has its own `spinlock_t` variable, which is used to lock the qdisc structure.

D Implementation challenges

One of the major implementation challenges faced was to implement packet timestamping approach in PIE for calculating the queuing delay. Although this is one of the optional approaches suggested in RFC 8033, details of incorporating this approach in PIE are missing. The problem is that timestamping approach calculates queuing delay at the dequeue time whereas PIE mechanism runs at the enqueue time. Moreover, timestamping approach provides *per packet queuing delay* whereas the drop probability (which needs

[2]https://www.kernel.org/doc/htmldocs/kernel-locking/locks.html

queuing delay information) is calculated once in 16 ms (default value of *tupdate* in Linux). Hence, the per packet queuing delay might not necessarily represent the exact state of the network during the calculation of drop probability. We noticed that the FreeBSD implementation of PIE did implement the timestamp based queuing delay calculation approach. We have used the FreeBSD code as a reference to implement timestamp based approach in FQ-PIE for Linux.

5.2.3 Evaluation

A Experimental Setup

This setup consider three real-time scenarios using Flent (Høiland-Jørgensen, 2015) to compare FQ-PIE with PIE and FQ-CoDel. The default parameter values of the mechanisms are considered. Flent's TCP upload test (`tcp_nup` - where n represents the number of TCP flows) is used to evaluate the performance. The tests are carried out with multiple TCP CUBIC (Ha et al., 2008) flows and a single on-off UDP flow to check the existence of a fair share between responsive and unresponsive flows. UDP flow is enabled and disabled every 50 seconds starting at the 25^{th} second (i.e UDP traffic starts at the 25^{th} second, stops at the 75^{th} second, is reinitialized at the 125^{th} second and so on). The complete simulation lasts for 300 seconds. Subsequently, in another test (`cubic_bbr`), we consider two flows each of CUBIC and BBR (Cardwell et al., 2016). This test enables us to determine the fairness of an AQM in the presence of different TCP congestion control variants. Finally, to test the performance of thin, latency sensitive applications, we run a VoIP G.711 standard stream in the presence of 4 bidirectional TCP CUBIC flows. VoIP G.711 uses 64 Kbit/s UDP streams. Flent allows us to emulate the VoIP traffic and collect the necessary statistics such as queuing delay, jitter and packet loss. These tests are run on the testbed shown in Figure 5.3 where all machines run Ubuntu 16.04 LTS with Linux kernel v5.1-rc1.

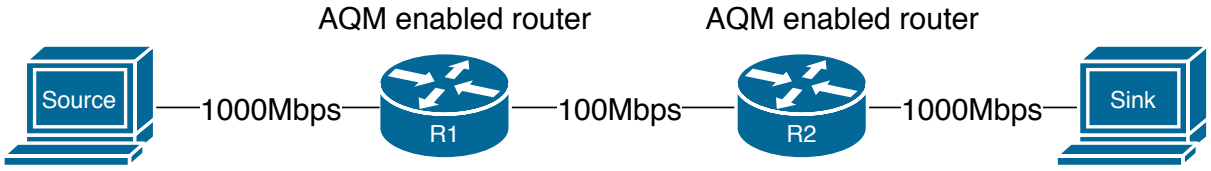


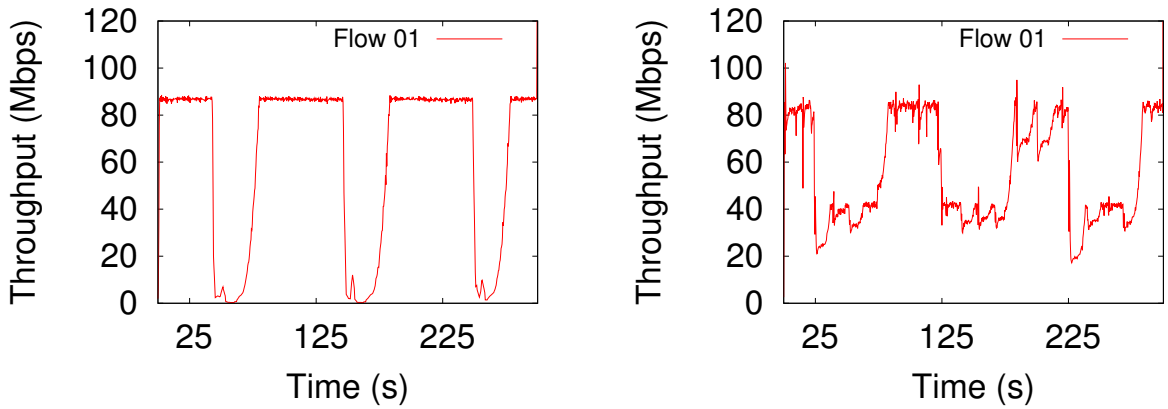
Figure 5.3: Testbed topology

The source node uses `netperf` to generate TCP traffic along with an `iperf` (Tirumala

et al., 2006) client to generate UDP traffic in all the UDP mix traffic tests. The link between the source and its adjacent router (R1) has a bandwidth of 1000 Mbps and a latency of 5 ms. The two routers share the bottleneck link of 100 Mbps, delay of 32.5 ms and have AQM mechanisms running on them. The link between the second router (R2) and the sink has a 5 ms latency and a 1000 Mbps bandwidth.

B Results and Discussions

The three metrics used for evaluation are TCP throughput, RTT and fairness. TCP throughput helps us to analyse the extent of fair sharing amongst multiple flows, whereas RTT gives us an insight into the impact of queuing delay in router buffers. Our evaluation consists of two scenarios which evaluate fairness between responsive and unresponsive flows, and among responsive flows.



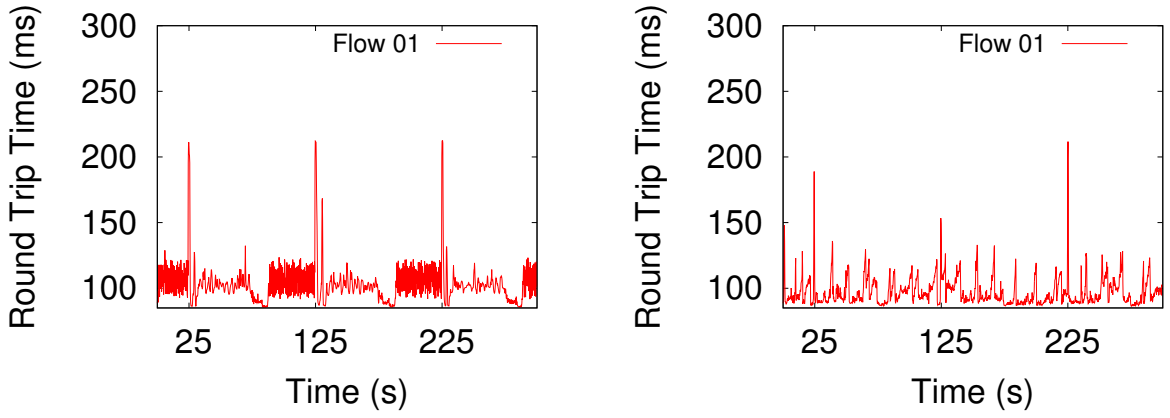
(a) PIE (b) FQ-PIE

Figure 5.4: TCP Throughput for `tcp_1up` test

C Responsive vs Unresponsive Flows

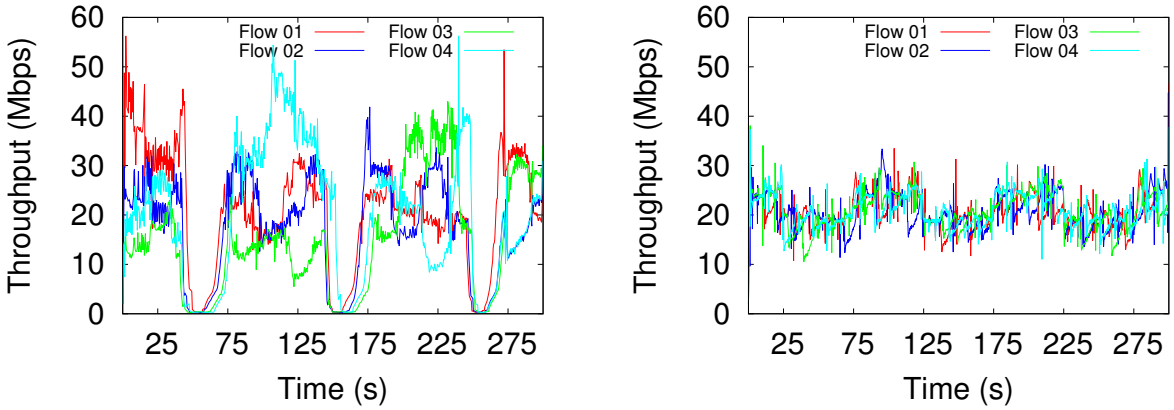
Initially, for the `tcp_1up` test, PIE and FQ-PIE results are obtained for the verification of flow queuing. Figure 5.4 represents the TCP throughput and and Figure 5.5) represents the RTT. Figure 5.4 (b) ensures the fairness for the TCP flow compared to Figure 5.4 (a) when UDP traffic is enabled. FQ-PIE achieves the similar TCP throughput whereas, Figure 5.5 shows the lowering density pattern for the RTT values from the PIE mechanism to the FQ-PIE mechanism.

We confirm this by the results obtained from the `tcp_4up` test. PIE and FQ-PIE exhibit similar behaviour in this scenario (Figure 5.6 and Figure 5.7).



(a) PIE (b) FQ-PIE

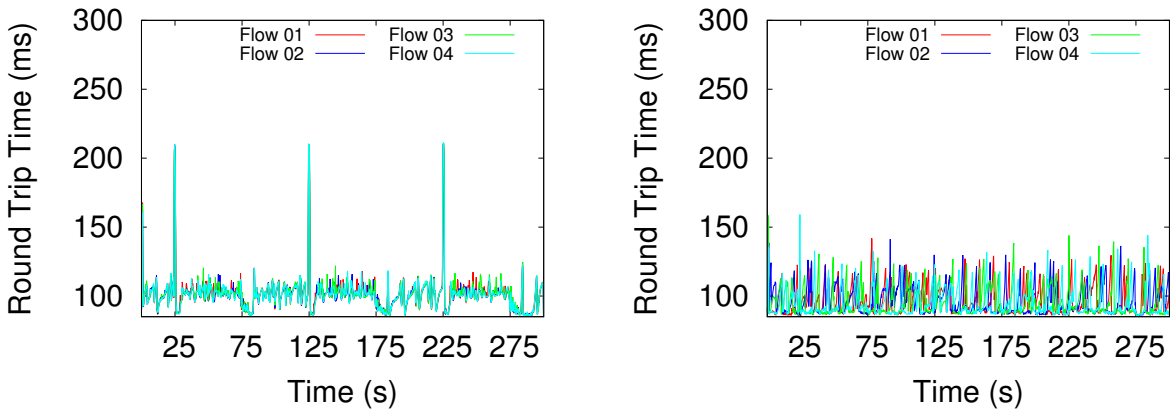
Figure 5.5: TCP Round Trip Time `tcp_1up` test



(a) PIE (b) FQ-PIE

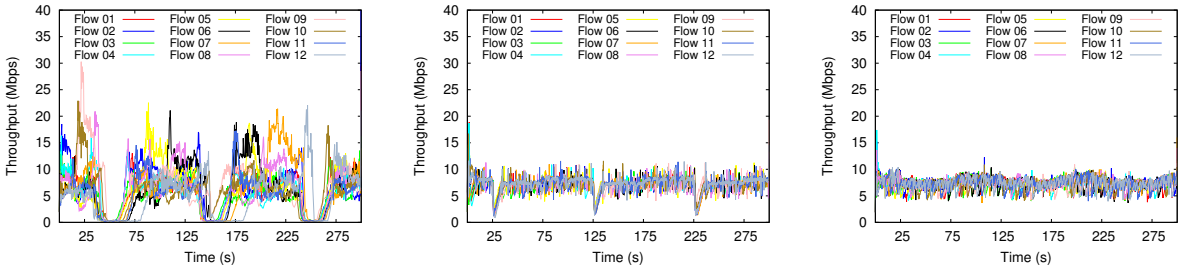
Figure 5.6: TCP Throughput for `tcp_4up` test

For the additional evaluation of TCP flow protection in the presence of unresponsive traffic, we observe the effect of an unresponsive, heavy flow on 12 TCP flows. This scenario gives a good estimate of fairness among the multiple TCP flows and the fairness with respect to the unresponsive traffic. Figure 5.8 and Figure 5.9 compare PIE, FQ-PIE and FQ-CoDel in terms of TCP throughput and RTT. As shown in Figure 5.8 (a), there is a sharp decline in TCP throughput with PIE each time the UDP flow starts. When the UDP flow stops, TCP throughput is restored. There is also a momentary spike in RTT each time the UDP flow starts. This observation can be attributed to the fact that unlike TCP, the UDP sender is congestion-agnostic and keeps sending packets irrespective of the network congestion state. This decreases the bandwidth available to TCP flows. Additionally, we observe that there is an uneven share of bandwidth among existing TCP flows due to lack of flow isolation in PIE.



(a) PIE (b) FQ-PIE

Figure 5.7: TCP Round Trip Time `tcp_4up` test

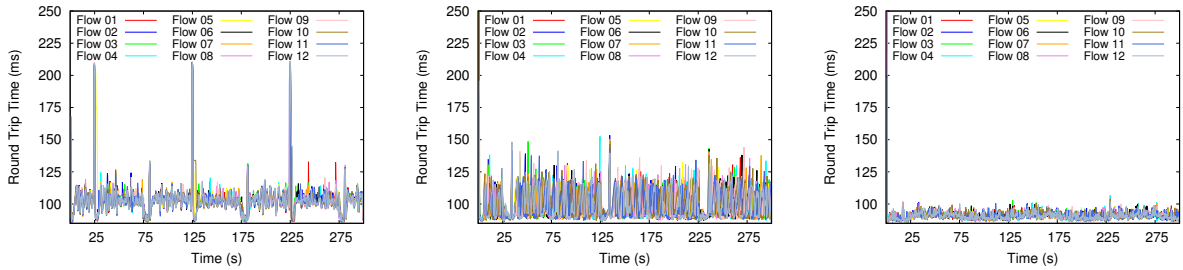


(a) PIE (b) FQ-PIE (c) FQ-CoDel

Figure 5.8: TCP Throughput for `tcp_12up` test

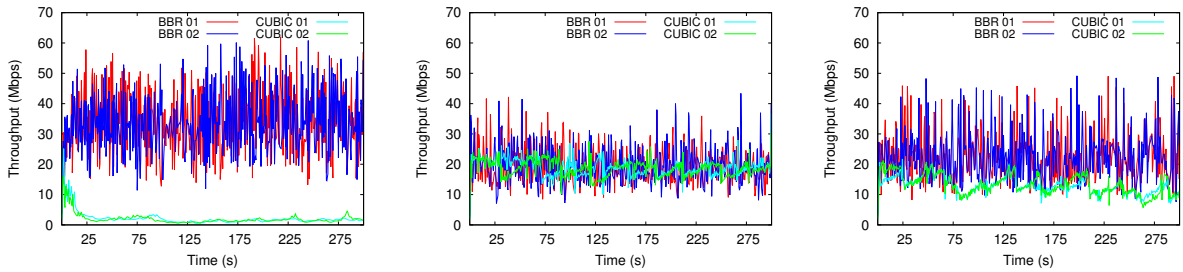
As observed in Figure 5.8 (b), there is a momentary decline in TCP throughput with FQ-PIE each time the UDP flow starts. However, unlike PIE, it recovers quickly and maintains a stable TCP throughput. The momentary drop in TCP bandwidth is due to the saturation of the qdisc by the UDP flow. However, due to the isolation of flows in FQ-PIE with the response of PIE to the UDP traffic, the qdisc de-saturates. Although there is a spike in RTT each time the UDP flow starts, it is more controlled than that of PIE. Figure 5.8 (b) and Table 5.1 shows that FQ-PIE ensures TCP flows get their fair share of bandwidth. This is due to the isolation of flows, solving the unfairness problem.

FQ-CoDel is as fair as FQ-PIE in terms of throughput as observed in Table 5.1. However, the momentary drop in throughput when a UDP flow starts, as observed in FQ-PIE (Figure 5.8 (b)) does not occur in FQ-CoDel. This is because FQ-CoDel follows an aggressive dropping strategy when the qdisc is completely saturated where up to 64 packets may be dropped from the largest flow as mentioned in Section 4.1 of RFC 8290. This allows for TCP packets to be enqueued, preventing the drop in TCP throughput.



(a) PIE (b) FQ-PIE (c) FQ-CoDel

Figure 5.9: TCP Round Trip Time for `tcp_12up` test

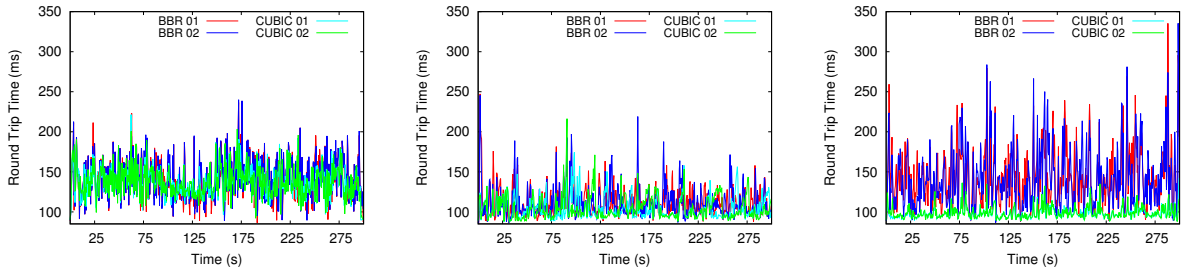


(a) PIE (b) FQ-PIE (c) FQ-CoDel

Figure 5.10: TCP Throughput for `cubic_bbr` test

D Fairness among Responsive Flows

CUBIC and BBR are two TCP congestion control variants being studied actively. CUBIC TCP is the default congestion control mechanism used in the Linux kernel and is an aggressive, loss based congestion control variant. BBR is a new congestion control variant proposed by Google. BBR provides congestion control by building a model of the network path and attempts to attain an ideal operating point with maximum bandwidth utilization. We evaluate the fairness among two CUBIC TCP and two TCP BBR flows by using PIE, FQ-PIE and FQ-CoDel. When PIE is used, we observe that BBR flows grab a greater share of bandwidth than CUBIC flows. This behaviour of TCP BBR can



(a) PIE (b) FQ-PIE (c) FQ-CoDel

Figure 5.11: TCP Round Trip Time for `cubic_bbr` test

Table 5.1: Calculation of Jain's Fairness Index

Test conducted	**AQM used**		
	PIE	FQ-PIE	FQ-CoDel
`tcp_1up`	0.94	0.98	0.96
`tcp_4up`	0.96	0.99	0.99
`tcp_8up`	0.74	0.99	0.99
`tcp_12up`	0.62	0.96	0.97
`cubic_bbr`	0.56	0.99	0.93

be attributed to its aggressive probing strategy which leads to higher throughput (Figure 5.10 (a)). As a consequence, CUBIC flows decrease their sending rates, thus failing to achieve their fair share of utilization. These observations are inline with those mentioned in (Scholz et al., 2018). When FQ-PIE is used, the problem of unfairness between BBR and CUBIC flows is resolved (Figure 5.10 (b)); both TCP flavors achieve a fair share of bandwidth and similar RTT.

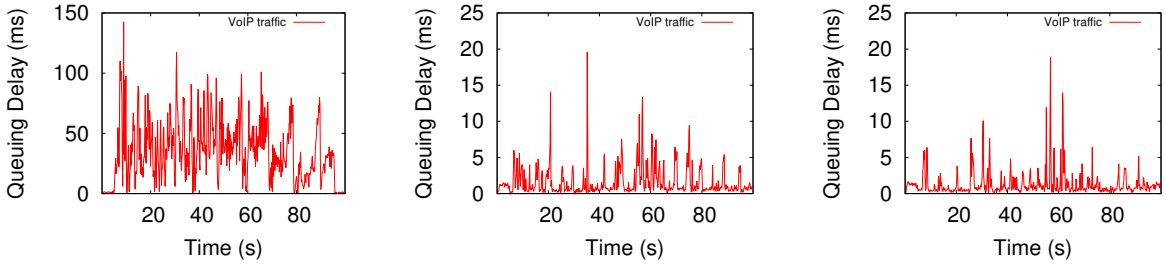
We note that BBR flows grab more bandwidth than CUBIC flows even when FQ-CoDel is used (ref. Figure 5.10 (c) and Table 5.1), whereas this is not the case with FQ-PIE. The main reason is that CoDel uses a linear-over-time packet drop schedule, whereas PIE uses a probability-based packet drop strategy. CoDel's packet drop policy is gentle, due to which the aggressive behaviour of BBR has more backlog. Consequently, BBR flows buffer more packets (and hence, occupy more bandwidth) than CUBIC flows which is confirmed by Figure 5.11 wherein it is observed that BBR flows have more RTT than CUBIC flows. Nevertheless, we believe that deeper investigations are required to understand the interactions of TCP BBR with modern queue management mechanisms such as FQ-CoDel and FQ-PIE.

E Protection for latency sensitive traffic

On Wide Area Networks (WANs), many real-time applications, such as DNS traffic, network management applications based on Simple Network Management Protocol (SNMP), audio and video transmissions share the bandwidth with non real-time applications. Real-time applications have unacceptable performances if queuing delays are incurred, and are referred to as latency sensitive traffic. The performance of these applications is measured

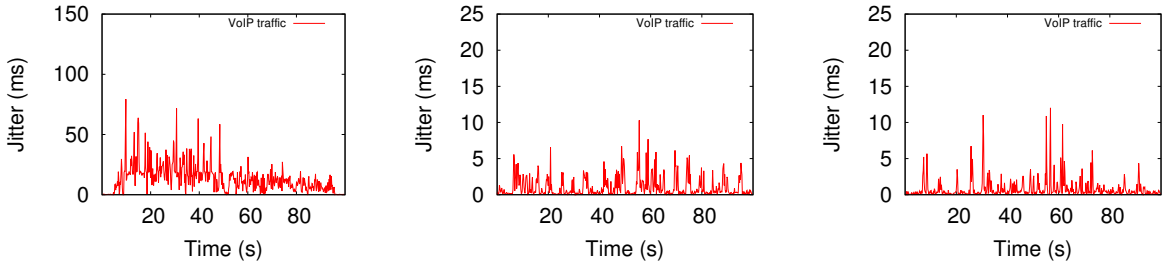
by Quality of Experience (QoE) which indicates how well the service is performing for the end user. When latency sensitive traffic and latency tolerant traffic share a bottleneck link, congestion in the link induces delay. This affects the Quality of Service (QoS), and subsequently the QoE of latency sensitive traffic. Hence, there is a need for providing protection for latency sensitive traffic when it co-exists with thick latency tolerant traffic.

VoIP is the transmission of Voice over Internet Protocol. It uses UDP as its underlying transport layer protocol. We evaluate the performance of FQ-PIE on protection of VoIP traffic from bulk TCP traffic. The test consists of 4 bi-directional TCP streams which run for 100 seconds. A single VoIP stream is tested for its one way delay, jitter and packet losses when it shares the same bottleneck link with the TCP streams.



(a) PIE (b) FQ-PIE (c) FQ-CoDel

Figure 5.12: Queuing Delay for `VoIP` test



(a) PIE (b) FQ-PIE (c) FQ-CoDel

Figure 5.13: Jitter for `VoIP` test

Note that the y-axis scale used for the queuing delay and jitter plots are different for PIE compared to the FQ* plots due to a varied difference in performance.

We observe that the VoIP stream performs well with low queuing delay, jitter and packet drops when FQ-PIE or FQ-CoDel is deployed (See Figure 5.12 (b) and (c), and (Figure 5.13 (b) and (c)). The flow protection mechanism in these AQM's ensures that the VoIP stream receives its fair share of bandwidth even in presence of thick TCP streams.

As the VoIP stream is thin, it does not experience a large queuing delay because the bandwidth of the stream is lesser than its obtained share of bandwidth from the AQM mechanism. Low queuing delay reduces the number of packet drops as observed in Table 5.2. This ensures that QoE is not degraded due to coexisting thick TCP traffic.

Table 5.2: Packet loss for VoIP flows (%)

AQM mechanism	**Packet loss(%)**
PIE	0.62%
FQ-PIE	0.00%
FQ-CoDel	0.00%

When PIE is used, the queuing delay and jitter is significantly higher compared to FQ-PIE and FQ-CoDel (Figure 5.12 (a)) and (Figure 5.13 (a)). Due to lack of flow protection in PIE, the queuing delay of the VoIP stream is dependent on the coexisting TCP flows. Additionally, as PIE does not differentiate between various flows during dropping stage, it is possible that VoIP packets are dropped due to the action of PIE, leading to packet losses as observed in Table 5.2. This severely impacts QoE for the end user.

Deriving motivation from the results obtained from FQ-PIE this work extends Minstrel PIE to support flow queuing.

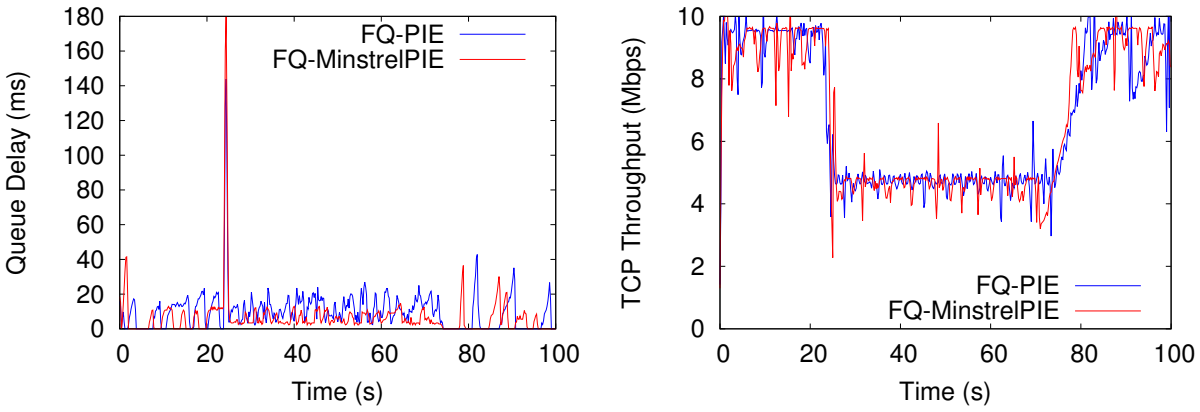
5.3 FQ-Minstrel PIE

There are no significant changes in the design and implementation of FQ-Minstrel PIE because the existing flow queuing features available in the kernel are leveraged to work with Minstrel PIE.

FQ-Minstrel PIE is evaluated in two different topologies i) the topology discussed in the Section 4.3.3, and ii) the topology discussed in the Section 5.2.3.

5.3.1 Evaluation with Section 4.3.3 topology

Both PIE and Minstrel PIE being AQM mechanisms do not provide flow protection. As discussed earlier, combining these mechanisms with flow queuing is a promising approach. We provide some preliminary insights into the benefits offered by combining Minstrel PIE with flow queuing and comparing its performance with FQ-PIE by using the same



(a) Queue Delay (b) TCP Throughput

Figure 5.14: Mix TCP and UDP Traffic with tcp_1up

Table 5.3: Fairness in tcp_1up test

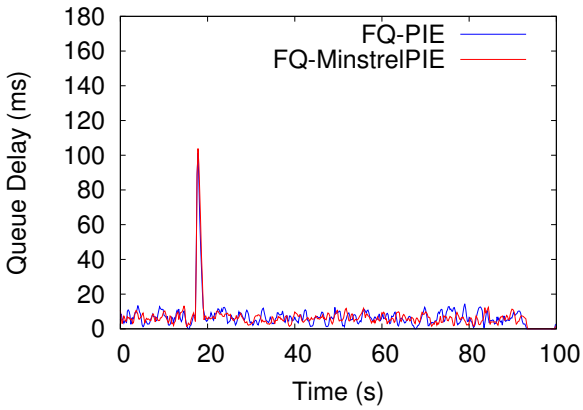
Flow Number	**Throughput (in Mbps)**		**Fairness**
	TCP 1	**UDP 1**	
FQ-PIE	4.70	4.80	0.99
FQ-Minstrel PIE	4.61	4.90	0.99

two configurations described in Section 4.3.3. The bottleneck bandwidth used for these experiments is 10Mbps as discussed in Section 4.3.3. Figure 5.14 (a) and (b) presents the queue delay averaged for active queues and TCP throughput for FQ-PIE and FQ-Minstrel PIE, respectively for 1-TCP-1-UDP configuration. Figure 5.14 (a) shows that FQ-Minstrel PIE has lesser queue delay averaged for active queues when UDP flow is on between 25 seconds to 75 seconds and Figure 5.14 (b) shows that using flow queuing offers significant advantage in providing fair share to the TCP flow. Table 5.3 confirms that TCP flow achieves its fair share of throughput.

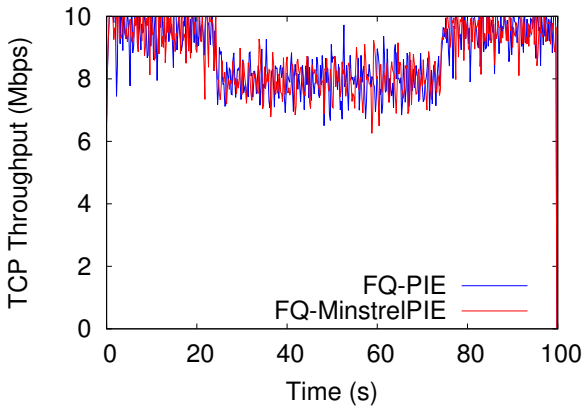
Table 5.4: Fairness in tcp_5up test

Flow Number	**Throughput (in Mbps)**						**Fairness**
	TCP 1	**TCP 2**	**TCP 3**	**TCP 4**	**TCP 5**	**UDP 1**	
FQ-PIE	1.58	1.60	1.58	1.60	1.60	1.62	**0.99**
FQ-Minstrel PIE	1.60	1.59	1.60	1.58	1.57	1.64	**0.99**

Figure 5.15(a) and 5.15(b) presents the queue delay averaged for active queues and



(a) Queue Delay



(b) TCP Throughput

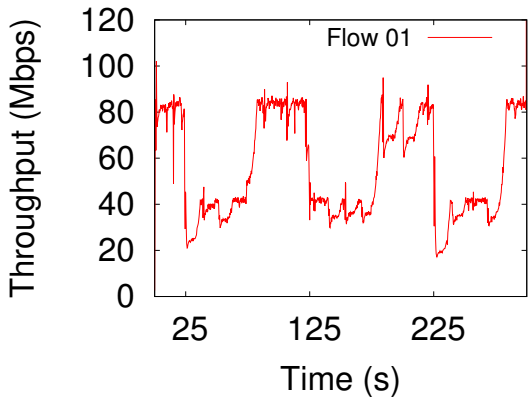
Figure 5.15: Mix TCP and UDP Traffic with tcp_5up

TCP throughput for FQ-PIE and FQ-Minstrel PIE, respectively for 5-TCP-1-UDP configuration. Since the number of TCP flows are more in this configuration, the aggregate TCP throughput achieved by PIE and Minstrel PIE is around 8 Mbps when UDP flow is on. Table 5.4 confirms that TCP flows get a fair share with flow queuing.

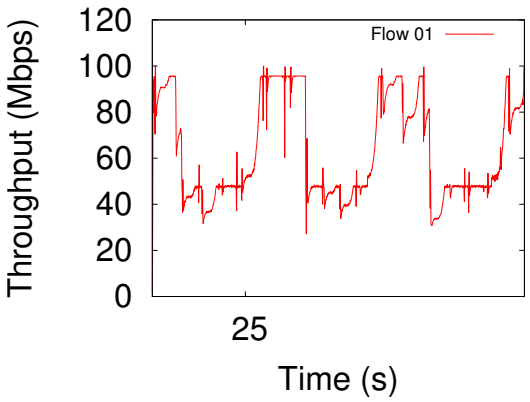
5.3.2 Evaluation with Section 5.2.3 topology

This subsection highlights the performance of FQ-Minstrel PIE in the same topology that are described in 5.2.3 with compare to FQ-PIE. The results for FQ-PIE are repeated from the subsection 5.2.3 for the performance comparison.

A Responsive vs Unresponsive Flows:

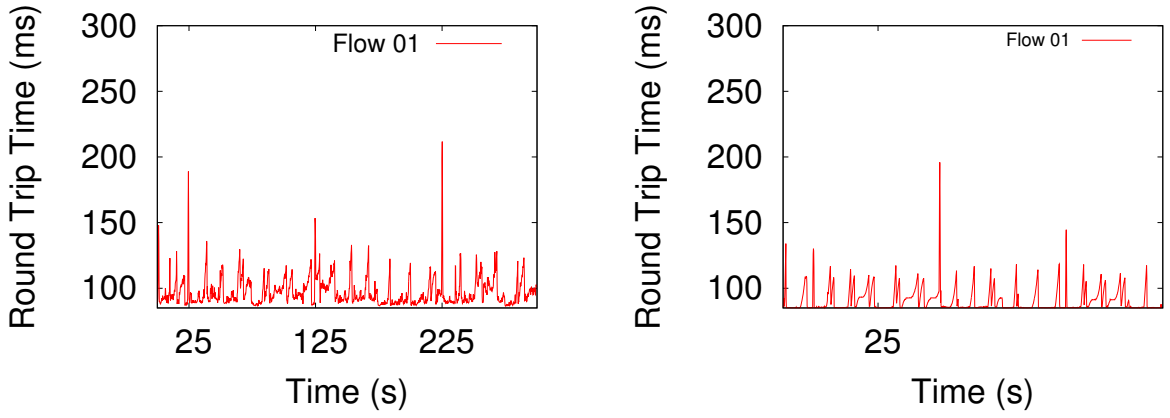


(a) FQ-PIE



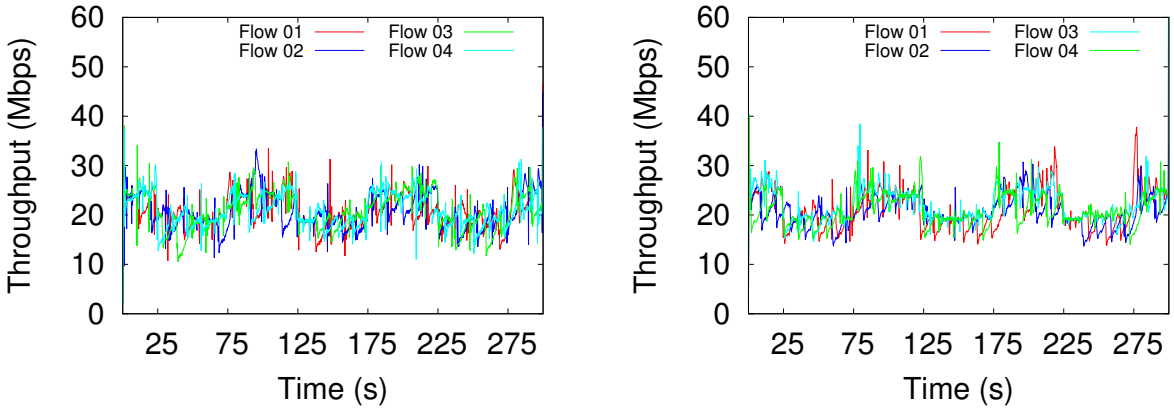
(b) FQ-Minstrel PIE

Figure 5.16: TCP Throughput for `tcp_1up` test with FQ-Minstrel PIE



(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.17: TCP Round Trip Time `tcp_1up` test with FQ-Minstrel PIE



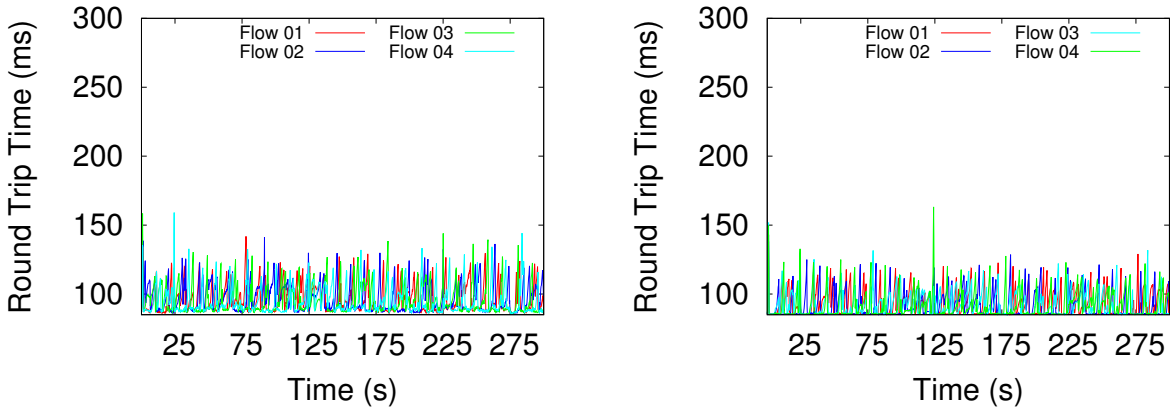
(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.18: TCP Throughput for `tcp_4up` test with FQ-Minstrel PIE

Initially, for the `tcp_1up` FQ-PIE and FQ-Minstrel PIE results are obtained for the verification of flow queuing in FQ-Minstrel PIE. Figure 5.16 represents the TCP throughput and Figure 5.17 represents RTT. Figure 5.16 (a) and Figure 5.16 (b) ensures the fairness for the TCP flow compared to PIE mechanism in Figure 5.4 (a) when UDP traffic is enabled. FQ-PIE and FQ-Minstrel PIE achieves the similar TCP throughput whereas, FQ-Minstrel PIE in Figure 5.17 (b) shows the lowering density pattern for the RTT values from the FQ-PIE mechanism in Figure 5.17 (a).

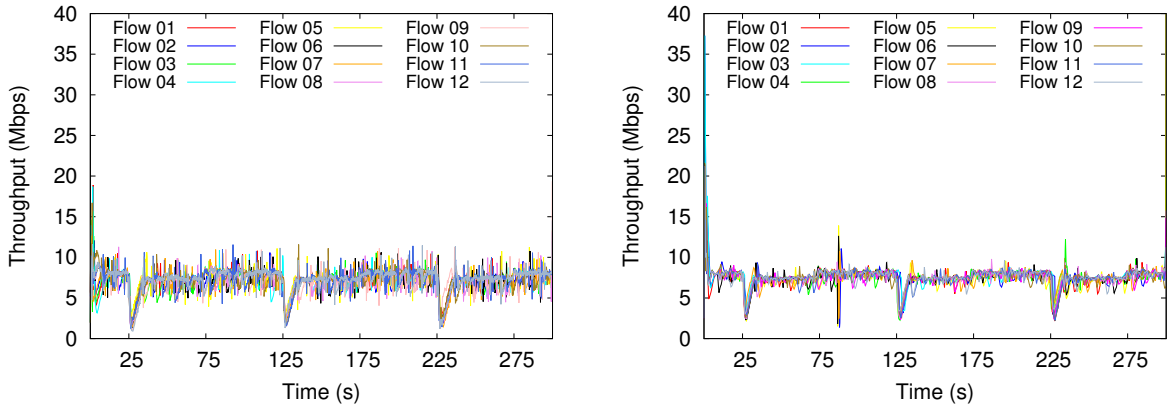
Although FQ-Minstrel PIE ((Figure 5.18 (b)) maintains the similar TCP throughput compared to FQ-PIE ((Figure 5.18 (a)) it controls the spikes in RTT slightly better as compared to FQ-PIE (Figure 5.19 (a)).

Figure 5.20 represents the TCP throughput and Figure 5.21 represents RTT for `tcp_12up` test. Figure 5.20 depicts that TCP throughput is more consistent with FQ-



(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.19: TCP Round Trip Time `tcp_4up` test with FQ-Minstrel PIE



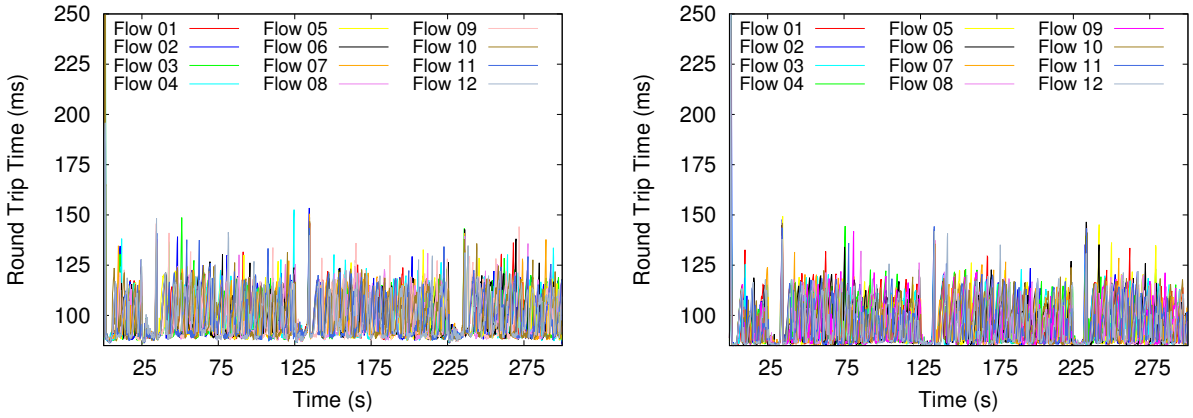
(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.20: TCP Throughput for `tcp_12up` test with FQ-Minstrel PIE

Minstrel PIE than FQ-PIE, and FQ-Minstrel PIE has better control on RTT which is depicted in Figure 5.21. This inline with our goals to achieve proper trade-off between link utilization and RTT. Table 5.5 confirms that FQ-Minstrel PIE does not affect the fairness of FQ-PIE for the responsive flows.

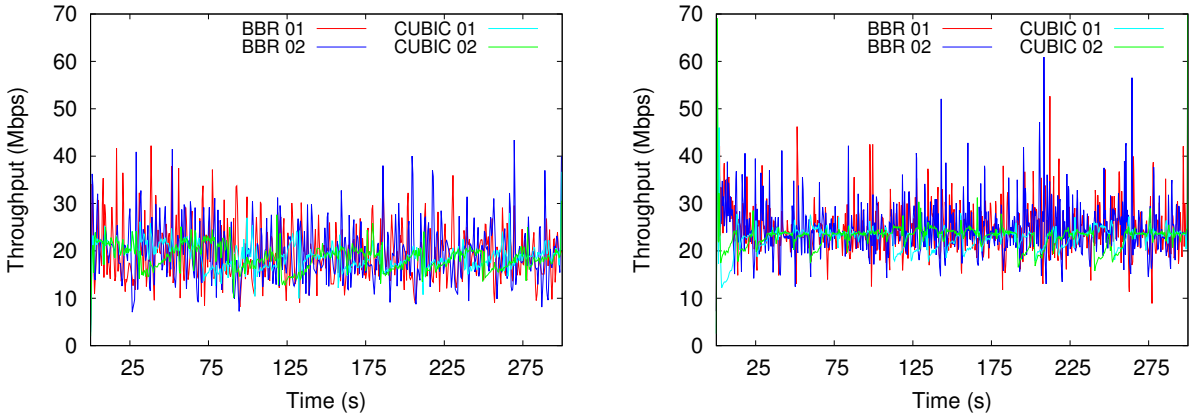
B Fairness among Responsive Flows:

Figure 5.22 represents the TCP throughput and Figure 5.23 represents RTT for `cubic_bbr` test. Figure 5.22 and Figure 5.23. The oscillations in throughput are minimal with CUBIC since, it adopts a multiplicative decrease of 30% whereas BBR does not follow this approach. BBR adjusts its pacing rate depending on an estimate of BDP. Hence, the decrease factor is not fixed in BBR, which eventually causes more oscillations in the throughput.



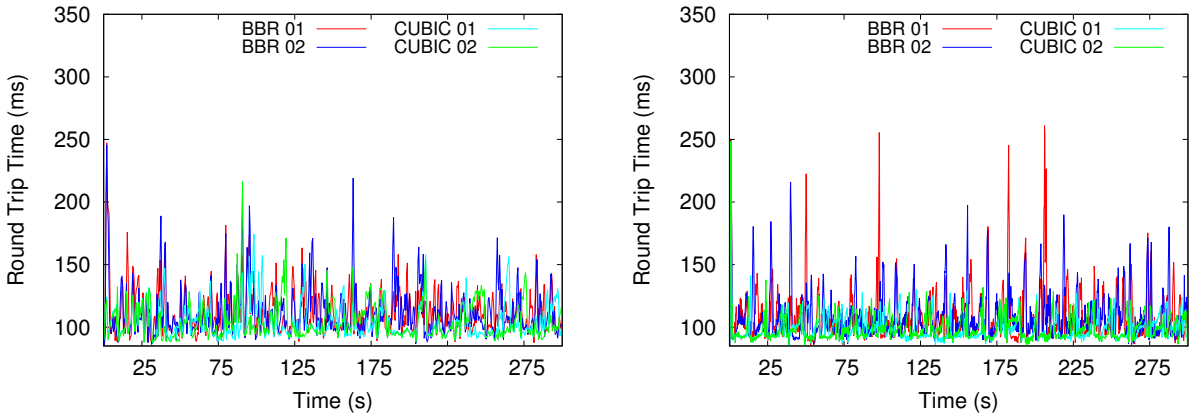
(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.21: TCP Round Trip Time for `tcp_12up` test with FQ-Minstrel PIE



(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.22: TCP Throughput for `cubic_bbr` test with FQ-Minstrel PIE

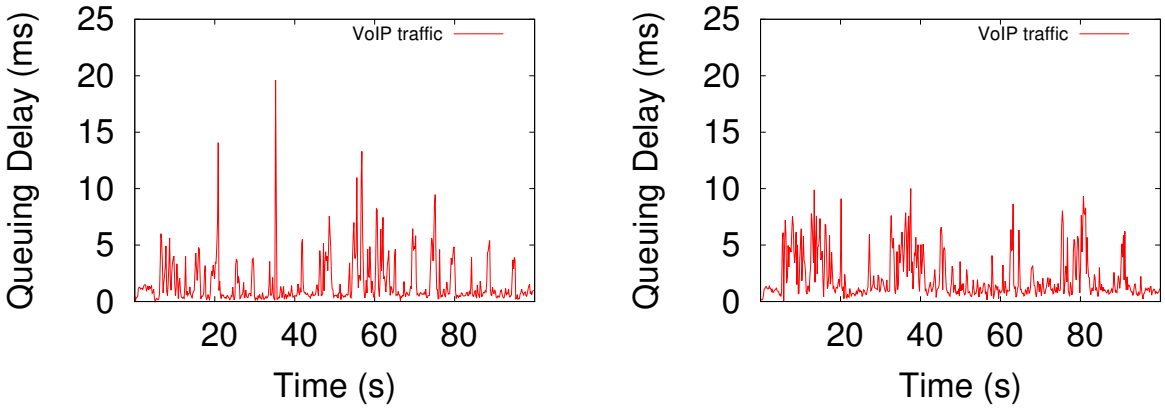


(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.23: TCP Round Trip Time for `cubic_bbr` test with FQ-Minstrel PIE

Table 5.5: Calculation of Jain's Fairness Index with FQ-Minstrel PIE

Test conducted	**AQM used**	
	FQ-PIE	FQ-Minstrel PIE
`tcp_1up`	0.98	0.97
`tcp_4up`	0.99	0.99
`tcp_8up`	0.99	0.99
`tcp_12up`	0.96	0.96
`cubic_bbr`	0.99	0.99



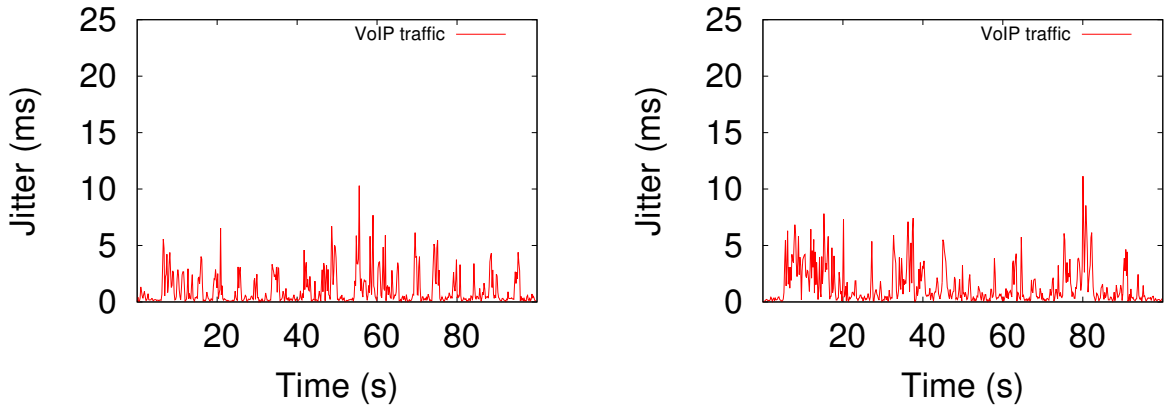
(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.24: Queuing Delay for `VoIP` test with FQ-Minstrel PIE

C Protection for latency sensitive traffic

We already noted the performance of FQ-PIE against latency sensitive traffic and it is shown that FQ-PIE performs better against the same. Therefore, this subsection analyzes whether the aggressiveness of Minstrel PIE affects the latency sensitive traffic or not with compare to FQ-PIE.

Figure 5.24 (a) and (b) shows the queuing delay for FQ-PIE and FQ-Minstrel PIE, respectively. FQ-PIE in Figure 5.24 (a) shows more spikes queuing delay than FQ-Minstrel PIE in Figure 5.24 (b). This is because FQ-Minstrel PIE always guarantees lesser queue delay for all kind of traffic than FQ-PIE due to its adaptive nature. Hence, we can see that optimized queue delay for FQ-Minstrel PIE than FQ-PIE. On the other hand, FQ-Minstrel PIE does not guaranty about the variations in the RTT which leads to follow



(a) FQ-PIE (b) FQ-Minstrel PIE

Figure 5.25: Jitter for `VoIP` test with FQ-Minstrel PIE

the same jitter as of FQ-PIE which can be observed in Figure 5.25 (a) and (b). On the same note Table 5.6 shows the packet loss 0% for FQ-PIE and Fq-Minstrel.

Table 5.6: Packet loss for VoIP flows (%) with FQ-Minstrel PIE

AQM mechanism	**Packet loss(%)**
FQ-PIE	0.00%
FQ-Minstrel PIE	0.00%

Figure 5.25 (a) and (b) presents the jitter results for FQ-PIE and FQ-Minstrel PIE, respectively. Both AQM mechanisms performing similarly while controlling the jitter. Subsequently, Table 5.6 shows that FQ-PIE and FQ-Minstrel PIE behaving similarly and do not disturb the latency sensitive traffic. Hence, FQ-Minstrel PIE also helps to optimize the performance of FQ-PIE as similar to Minstrel PIE do it for PIE.

5.4 Inferences

This chapter discusses the design and implementation of FQ-PIE and FQ-Minstrel PIE in the Linux kernel. First, the performance of FQ-PIE is evaluated against PIE and FQ-CoDel in a real testbed for the verification of flow queuing and later, the performance of FQ-Minstrel PIE is evaluated against FQ-PIE. The results indicate that FQ-PIE and FQ-Minstrel PIE both resolve the issue of fairness in PIE and Minstrel PIE, respectively when unresponsive flows share the bottleneck link with responsive flows. FQ-PIE performs better than FQ-CoDel when BBR flows coexist with CUBIC flows. We also demonstrate

that FQ-PIE performs better than PIE in protecting thin, latency-sensitive traffic from coexisting thick TCP flows. Whenever FQ-Minstrel PIE gets a chance to improve the trade-off between TCP throughput and queue delay than FQ-PIE, it does so in all the tests (`tcp_1up`, `tcp_5up`, `tcp_12up`, `cubic_bbr` and `VoIP` tests). FQ-PIE implementation has been made publicly available[3] and it has been submitted for review to Linux developers. FQ-PIE model is already merged into the mainline of the Linux kernel, and the addition of FQ-PIE module has been listed among the best features of Linux 5.6 by Phoronix.

[3]https://github.com/gautamramk/FQ-PIE-for-Linux-Kernel

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This work proposes three algorithms called Modified CoDel, Minstrel PIE and FQ-Minstrel PIE. These algorithms are designed to optimize the trade-off between bottleneck link utilization and queue delay when unresponsive flows coexist with responsive flows. It was observed that the CoDel algorithm has limited queue control when the traffic is unresponsive. To overcome this limitation, we proposed a new variant called Modified CoDel with minor modifications to the control law of CoDel. Initially, from the simulation studies, we observed that Modified CoDel has better queue control than CoDel in terms of handling the unresponsive traffic. However, on deeper investigations in real test-beds, we observed that CoDel has limitations while operating in low bandwidth scenarios and we found that our observations were inline with the bufferbloat community. Depending on this analysis, it was understood that Modified CoDel will have limited scalability due to CoDel's inherent limitation, and hence the thesis focus shifted to the PIE.

PIE is known to have better queue control compared to CoDel algorithm, particularly when the unresponsive traffic coexists. We noted that the PIE algorithm has a parameter called *average queue rate* which could be leveraged to make the algorithm to quickly adjust its packet drop probability depending on the presence and absence of unresponsive traffic. Minstrel PIE described in this thesis is an outcome of efforts in this direction. The results obtained from simulation studies and real-time experiments validate the effectiveness and robustness of Minstrel PIE against unresponsive traffic. Minstrel PIE can be incrementally deployed in a real network setup as it is a minor modification of PIE, and does not require the user to configure any parameter explicitly.

Furthermore, it is known that a single queue AQM algorithm like PIE or Minstrel PIE cannot provide flow isolation between responsive and unresponsive flows, which in turn results in unfair bandwidth allocation. To overcome this problem, FQ-Minstrel PIE is designed to provide fairness against unresponsive flows. FQ-Minstrel PIE ensures a fair allocation of resources to responsive flows when they coexist with unresponsive flows. We recommend that FQ-Minstrel PIE is a suitable and effective algorithm to be investigated further for real-time deployment.

Besides, this thesis makes three additional contributions; a fluid model is proposed for CoDel, PIE implementation in the Linux kernel is aligned to RFC 8033 and a new module for FQ-PIE has been implemented in the Linux kernel. The proposed fluid model gives better insights into CoDel's control law and the contributions related to PIE and FQ-PIE in the Linux kernel would assist the research community to deploy these algorithms in the real environment and perform deeper investigations into its advantages. Both these contributions are merged into the mainline of the Linux kernel, and the addition of FQ-PIE module has been listed among the best features of Linux 5.6 by Phoronix.

6.2 Limitations and Future work

Although there are two approaches to calculate queue delay in PIE, the limitation of Minstrel PIE is that it works only with Little's law. This is because Minstrel PIE depends on *average dequeue rate* (avg_dq_rate). One possible way to resolve this problem is to extend the design of Minstrel PIE to extract the current dequeue rate from the timestamping approach. This would require deeper investigations and we intend to continue our work in this direction.

The additional avenues which can be explored to expand the work in this thesis are listed below:

- An important aspect to consider during the deployment of the AQM algorithms into operational networks is the impact of multiple bottleneck links on the performance of the network and the QoS perceived by the user. This aspect is important from a modelling perspective and has not received due attention in the literature. Although some work exists on analysing Compound TCP (deployed in Microsoft Windows OS) with drop tail queues in an environment with multiple bottlenecks, much more remains unexplored. In fact, there is an opportunity to analyse the proposed queue

management policies in this thesis in such a setting in the future.

- Moreover, it would be interesting to study the impact of different TCP variants on the queue management algorithms proposed in this thesis, e.g, Compound TCP (which is a delay and a loss based protocol) and Reno (a classical, but a quite effective protocol that uses loss as the primary indicator of congestion). In addition to the above, this thesis has covered PIE, which is an extension of the PI algorithm, so this work can be extended to perform a comparative study in terms of stability and performance analysis of PIE and Minstrel PIE, with other TCP variants like Compound TCP or Reno TCP, or other popular variants of TCP.

- Another approach would be to approximate the buffer size with an infinite buffer system for which queueing models such as M/M/1, M/G/1 etc., are available and some closed form expressions for queuing delay etc. are available. The results obtained from these delay models can be then compared to the results presented in this thesis. Additionally, there is the scope to perform statistical analysis of the queue management policies proposed in this work with different buffer sizing regimes. Chapter 4, for example, makes a case for the small buffer regime with drop tail queues. It will be interesting to observe the results obtained from these analytical studies.

- Nowadays, the Internet is witnessing the problem of starvation for time-sensitive applications. One of the goals of this thesis is to improve the overall QoS that the application perceives. Thus, one of the potential approaches to extend the work done in this thesis is to provide differentiated services depending on the application requirements. This needs an intelligent traffic segregation interface at the application layer with the machine learning (artificial intelligence) approach or a database which comprises all the types of applications with their protocol identifiers. Internet of Things (IoT) applications have tight requirements in terms of QoS. By considering the importance of IoT traffic, it would be interesting to evaluate the performance of the algorithms proposed in this thesis in IoT environments.

Bibliography

Abramowitz, M. and Stegun, I. A. (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition.

Al-Saadi, R. and Armitage, G. (2016). Dummynet AQM v0. 2–CoDel, FQ-CoDel, PIE and FQ-PIE for FreeBSD's ipfw/dummynet framework. *Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. A*, 160418:18.

Appenzeller, G., Keslassy, I., and McKeown, N. (2004). *Sizing router buffers*, volume 34. ACM.

Athuraliya, S., Li, V., Low, S., and Yin, Q. (2001). REM: Active Queue Management. In *Teletraffic Engineering in the Internet Era*, volume 4 of *Teletraffic Science and Engineering*, pages 817–828. Elsevier.

Bergkvist, A., Burnett, D. C., Jennings, C., Narayanan, A., and Aboba, B. (2012). WebRTC 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

Boerlage, J. and Collom, R. (2016). Implementing Active Queue Management at the home to reduce NBN speed demands. Technical Report 161107A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia.

Cai, J., Zhang, Z., and Song, X. (2010). An analysis of UDP traffic classification. In *12th IEEE International Conference on Communication Technology (ICCT)*, pages 116–119. IEEE.

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. (2016). BBR: Congestion-Based Congestion Control. *ACM Queue*, 14, September-October:20–53.

Casoni, M., Grazia, C. A., Klapez, M., and Patriciello, N. (2017). How to avoid tcp congestion without dropping packets: An effective aqm called pink. *Computer Communications*, 103:49–60.

Chen, J., Hu, C., and Ji, Z. (2011). Self-tuning random early detection algorithm to improve performance of network transmission. *Mathematical Problems in Engineering*, 2011:1–17.

Chen, W., Min, G., and Zhang, H. (2012). Statistical adapting RED in dynamic networks. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 2560–2565. IEEE.

De Schepper, K., Bondarenko, O., Tsang, I., and Briscoe, B. (2016). PI 2: A Linearized AQM for both Classic and Scalable TCP. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 105–119. ACM.

Deepak, A., Shravya, K. S., and Tahiliani, M. P. (2017). Design and Implementation of AQM Evaluation Suite for ns-3. In *Proceedings of the Workshop on ns-3*, WNS3 '17, pages 87–94, New York, NY, USA. ACM.

Enachescu, M., Ganjali, Y., Goel, A., McKeown, N., and Roughgarden, T. (2005). Part III: Routers with Very Small Buffers. *Computer Communication Review*, 35:83–90.

Feng, W. ., Kandlur, D. D., Saha, D., and Shin, K. G. (1999). A self-configuring RED gateway. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 3, pages 1320–1328 vol.3.

Feng, W., Shin, K. G., Kandlur, D. D., and Saha, D. (2002). The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 10(4):513–528.

Feng, W.-c., Kandlur, D. D., Saha, D., and Shin, K. G. (2001). Stochastic fair blue: A queue management algorithm for enforcing fairness. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1520–1529. IEEE.

Floyd, S., Gummadi, R., and Shenker, S. (2001). Adaptive RED: An algorithm for increasing the robustness of RED's active queue management. Technical report, ICSI.

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

Forbes, C., Evans, M., Hastings, N., and Peacock, B. (2011). *Statistical Distributions, 4th Edition*. John Wiley & Sons, NJ, USA.

Francini, A. (2012). Periodic early detection for improved TCP performance and energy efficiency. *Computer Networks*, 56(13):3076–3086.

Gettys, J. and Nichols, K. (2011). Bufferbloat: Dark buffers in the Internet. *Communications of the ACM*, 55(1):57–65.

Ghoreishi, S. E., Aghvami, A. H., and Saki, H. (2015). Active queue management for congestion avoidance in multimedia streaming. In *2015 European Conference on Networks and Communications (EuCNC)*, pages 487–491. IEEE.

Grigorik, I. (2013). *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, Inc.

Groenewegen, D. and Kleppe, H. (2011). Detecting and quantifying bufferbloat in network paths. Technical report, www.Bufferbloat.net.

Ha, S., Rhee, I., and Xu, L. (2008). CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74.

Hassan, M. and Jain, R. (2003). *High Performance TCP/IP Networking*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Hayes, D., Ros, D., Andrew, L., and Floyd, S. (2007). TCP Evaluation Suite.

Henderson, T. R., Lacage, M., Riley, G. F., Dowell, C., and Kopena, J. (2008). Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527–527.

Høiland-Jørgensen, T. (2015). Flent: The FLExible Network Tester. In *The 11th Swedish National Computer Networking Workshop (SNCNW), Karlstad, Sweden, May 28–29*.

Høiland-Jørgensen, T., Täht, D., and Morton, J. (2018). Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways. In *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 37–42.

Hollot, C., Liu, Y., Misra, V., and Towsley, D. (2003). Unresponsive flows and AQM performance. In *INFOCOM*, volume 1, pages 85–95. IEEE.

Hollot, C. V., Misra, V., Towsley, D., and Gong, W. (2001). On designing improved controllers for AQM routers supporting TCP flows. In *INFOCOM, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies Proceedings*, volume 3, pages 1726–1734. IEEE.

Jain, R., Chiu, D. M., and WR, H. (1998). A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099.

Jain, T., Annappa, B., and Tahiliani, M. P. (2014). Performance Evaluation of CoDel for Active Queue Management in Wired-Cum-Wireless Networks. In *Advanced Computing & Communication Technologies (ACCT), 2014 Fourth International Conference on*, pages 381–385. IEEE.

Järvinen, I. and Kojo, M. (2014). Evaluating CoDel, PIE, and HRED AQM techniques with load transients. In *IEEE 39th Conference on Local Computer Networks (LCN)*, pages 159–167. IEEE.

Javam, H. and Analoui, M. (2006). Sared: Stabilized ared. In *2006 International Conference on Communication Technology*, pages 1–4.

Jiang, H., Wang, Y., Lee, K., and Rhee, I. (2012). Tackling bufferbloat in 3G/4G networks. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 329–342. ACM.

Jones, R. et al. (1996). NetPerf: a Network Performance Benchmark. *Information Networks Division, Hewlett-Packard Company, (1996).*

Kennedy, J., Armitage, G., and Thomas, J. (2017). Household bandwidth and the 'need for speed': Evaluating the impact of active queue management for home internet traffic. *Australian Journal of Telecommunications and the Digital Economy*, 5(2):113–130.

Khademi, N., Ros, D., and Welzl, M. (2013). The New AQM Kids on the Block: Much Ado About Nothing? *Research report http://urn.nb.no/URN:NBN:no-35645*.

Kim, T.-H. and Lee, K.-H. (2006). Refined adaptive RED in TCP/IP networks. In *2006 SICE-ICASE International Joint Conference*, pages 3722–3725. IEEE.

Kobayashi, K. (2015). Lawin: A latency-aware internet architecture for latency support on best-effort networks. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE.

Kuhn, N., Lochin, E., and Mehani, O. (2014). Revisiting old friends: is CoDel really achieving what RED cannot? In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 3–8. ACM.

Kuhn, N. and Ros, D. (2016). Improving PIE's performance over high-delay paths. *CoRR*, abs/1602.00569, 2016.

Kuhn, N., Ros, D., Bagayoko, A. B., Kulatunga, C., Fairhurst, G., and Khademi, N. (2017). Operating ranges, tunability and performance of CoDel and PIE. *Computer Communications*, 103:74–82.

Kulatunga, C., Kuhn, N., Fairhurst, G., and Ros, D. (2015). Tackling Bufferbloat in capacity-limited networks. In *European Conference on Networks and Communications (EuCNC)*, pages 381–385. IEEE.

Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W., and Shi, Z. (2017). The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 183–196, New York, NY, USA. ACM.

Li, F., Jiang, X., Chung, J. W., and Claypool, M. (2018). Who is the King of the Hill? Traffic Analysis over a 4G Network. In *IEEE International Conference on Communications (ICC)*, pages 1–6.

Li, W., Zeng-zhi, L., Yan-ping, C., and Ke, X. (2005). Fluid-based stability analysis of mixed TCP and UDP traffic under RED. In *10th IEEE International Conference*

on Engineering of Complex Computer Systems, ICECCS Proceedings, pages 341–348. IEEE.

Little, J. D. C. and Graves, S. C. (2008). *Little's Law*, pages 81–100. Springer US, Boston, MA.

McCanne, S. and Floyd, S. (1997). The LBNL network simulator (ns-2).

McKenney, P. E. (1990). Stochastic fairness queueing. In *Proceedings. IEEE INFOCOM '90: Ninth Annual Joint Conference of the IEEE Computer and Communications Societies The Multiple Facets of Integration*, volume 2, pages 733–740.

Misra, V., Gong, W.-B., and Towsley, D. (2000). Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 151–160. ACM.

Morton, J. (2016). [Cake] Proposing COBALT, Online available at: https://lists.bufferbloat.net/pipermail/cake/2016-May/001925.html.

Nichols, K. and Jacobson, V. (2012). Controlling Queue Delay. *Communications of the ACM*, 55(7):42–50.

Palaniappan, B. et al. (2013). Bufferfloat Mitigation for Real-time Video Streaming using Adaptive Controlled Delay Mechanism. *International Journal of Computer Applications*, 63(20):1–6.

Pan, R., Natarajan, P., Piglione, C., Prabhu, M. S., Subramanian, V., Baker, F., and VerSteeg, B. (2013). PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pages 148–155. IEEE.

Raghuvanshi, D. M., Annappa, B., and Tahiliani, M. P. (2013). On the effectiveness of CoDel for active queue management. In *Advanced Computing and Communication Technologies (ACCT), 2013 Third International Conference on*, pages 107–114. IEEE.

Raina, G., Manjunath, S., Prasad, S., and Giridhar, K. (2016). Stability and Performance Analysis of Compound TCP with REM and Drop-Tail Queue Management. *IEEE/ACM Transactions on Networking*, 24(4):1961–1974.

Raina, G., Towsley, D., and Wischik, D. (2005). Part II: Control Theory for Buffer Sizing. *SIGCOMM Comput. Commun. Rev.*, 35(3):79–82.

Raina, G. and Wischik, D. (2005). Buffer sizes for large multiplexers: TCP queueing theory and instability analysis. In *Next Generation Internet Networks*, pages 173–180.

Scholz, D., Jaeger, B., Schwaighofer, L., Raumer, D., Geyer, F., and Carle, G. (2018). Towards a Deeper Understanding of TCP BBR Congestion Control. In *IFIP Networking 2018*, Zurich, Switzerland.

Showail, A., Jamshaid, K., and Shihada, B. (2014a). An empirical evaluation of bufferbloat in IEEE 802.11 n wireless networks. In *Wireless Communications and Networking Conference (WCNC), 2014 IEEE*, pages 3088–3093. IEEE.

Showail, A., Jamshaid, K., and Shihada, B. (2014b). WQM: An aggregation-aware queue management scheme for IEEE 802.11n based networks. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing*, pages 15–20. ACM.

Shreedhar, M. and Varghese, G. (1996). Efficient fair queuing using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385.

Tahiliani, M. P. and Shet, K. (2013). Analysis of cautious adaptive RED (CARED). In *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*, pages 1029–1034. IEEE.

Tan, K. and Song, J. (2006). Compound TCP: A Scalable and TCP-friendly Congestion Control for High-speed Networks. In *4th International workshop on Protocols for Fast Long-Distance Networks (PFLDNet)*.

Tan, K., Song, J., Zhang, Q., and Sridharan, M. (2006). A Compound TCP Approach for High-Speed and Long Distance Networks. In *25th IEEE International Conference on Computer Communications, Proceedings IEEE INFOCOM*, pages 1–12.

Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. (2005). Iperf: The tcp/udp bandwidth measurement tool. htt p. *dast. nlanr. net/Projects*, page 38.

Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. (2006). Iperf, (2006),.

Wang, P., Zhu, D., and Lu, X. (2017). Active queue management algorithm based on data-driven predictive control. *Telecommunication Systems*, 64(1):103–111.

Winstein, K. and Balakrishnan, H. (2013). TCP Ex Machina: Computer-generated Congestion Control. In *Proceedings of the ACM*, SIGCOMM '13, pages 123–134, New York, NY, USA. ACM.

Wischik, D. and McKeown, N. (2005). Part I: Buffer sizes for core routers. *Computer Communication Review*, 35:75–78.

Xue, L., Kumar, S., Cui, C., Kondikoppa, P., Chiu, C.-H., and Park, S.-J. (2013). Afcd: An approximated-fair and controlled-delay queuing for high speed networks. In *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pages 1–7. IEEE.

Yamaguchi, F. and Nishi, H. (2013). Hardware-based hash functions for network applications. In *2013 19th IEEE International Conference on Networks (ICON)*, pages 1–6. IEEE.

Journals

- Sachin D. Patil and Mohit P. Tahiliani. "Minstrel PIE: Curtailing Queue Delay in Unresponsive Traffic Environments" Computer Communications, Volume 139, 2019, Pages 16–31. [SCI Indexed journal, Impact Factor: 2.613]

- Sachin D. Patil and Mohit P. Tahiliani. "Towards a better understanding and analysis of controlled delay (CoDel) algorithm by using fluid modelling" IET Networks, Volume 8, Issue 1, 2018, Pages 59–66. [Scopus Indexed journal, CiteScore: 2.01]

Conferences

- Sachin D. Patil and Mohit P. Tahiliani. "On the robustness of AQM mechanisms against non-responsive traffic" In 10^{th} IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), 2016, pp. 1-6.

- Ramakrishnan G., V.Saicharan, Bhasi M, Monis L., Sachin D. Patil and Mohit P. Tahiliani, "FQ-PIE Queue Discipline in the Linux Kernel: Design, Implementation and Challenges", In 44^{th} IEEE international Conference on Local Computer Networks (LCN 2019).

Under progress

- Sumukha PK, Prajval M, Ishaan R Dharamdas, Sachin D. Patil and Mohit P. Tahiliani, "Implementation, validation and evaluation of FQ-PIE in ns-3", to be submitted.

- Manish Kumar B, Hrishikesh Hiraskar, Dhaval Khandla, Leslie Monis, Sachin D. Patil and Mohit P. Tahiliani, "Alignment of PIE algorithm implementation with RFC 8033 and evaluation in the Linux kernel", to be submitted.

Open Source Contributions

- **Alignment of PIE algorithm with RFC 8033 in the Linux kernel**

  Source code is merged in the Linux kernel since version 5.1

- **Implementation of FQ-PIE algorithm in the Linux Kernel**

  Source code is merged in the Linux kernel sicnce version 5.5

- **Implementation of FQ-PIE algorithm in the ns-3**

  The source code is being submitted in ns-3 for review.

Brief Bio-Data

Sachin Dattatraya Patil

Research Scholar

Department of Computer Science and Engineering

National Institute of Technology Karnataka, Surathkal

P.O. Srinivasnagar

Mangalore - 575025

Phone: +91 7276076720

Email: sdp.sachin@gmail.com

Permanent Address

Sachin Dattatraya Patil

Gajanan Colony, Mali Plot

Near Sangli Railway Station

Sangli - 416416

Maharashtra, INDIA

Qualification

M. Tech. in Computer Science and Engineering, VJTI, Mumbai, Maharashtra, 2012.

B. E. in Information Technology, Government College of Engineering, Karad, Maharashtra, 2008.