Active Queue Management

Implementation of PI^2 Queuing Discipline for Classic TCP Traffic in ns-3

by

Rohit Prakash Tahiliani, B.E.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2017

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Rohit Prakash Tahiliani

August 29, 2017

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Rohit Prakash Tahiliani

August 29, 2017

Acknowledgments

In my journey towards the completion of my postgraduate thesis, I was assisted and guided by numerous people.

Words are insufficient to express my sincere gratitude towards my dissertation supervisor, Prof. Hitesh Tewari, for his constant guidance, support and encouragement. He has given me a lot of freedom in my research and allowed me to work on the problems of my interest. His passion for research showed me how academia can be exceedingly rewarding and great fun. I have learnt a lot and still have plenty to learn from him.

Whatever I am today, is because of the never ending support of my father Dr. Prakash U. Tahiliani and my mother Mrs. Meena P. Tahiliani. My brother, Dr. Mohit P. Tahiliani has been a rock solid support right since my childhood. I dedicate the successful completion of my post graduate thesis to my family. My brother's great confidence in my abilities encouraged me to always go an extra mile and pursue masters.

Rohit Prakash Tahiliani

University of Dublin, Trinity College

September 2017

Active Queue Management

Implementation of PI^2 Queuing Discipline for Classic TCP Traffic in ns-3

Rohit Prakash Tahiliani, M.Sc.
University of Dublin, Trinity College, 2017

Supervisor: Hitesh Tewari

This dissertation presents the implementation and validation of PI^2 Active Queue Management (AQM) algorithm in ns-3 simulator. AQM mechanisms have been extensively studied and deployed in the Internet to monitor and limit the growth of the queue at routers. These mechanisms *avoid* congestion by proactively informing the sender about congestion, either by dropping a packet or by marking a packet. Many algorithms such as Random Early Detection (RED) and Controlled Delay (CoDel) have been designed to control the queuing delay and retain high link utilization. The state of the art queue management algorithms include Proportional Integral controller Enhanced (PIE) and PI^2. PI^2 provides an alternate design and implementation to PIE algorithm without affecting the performance benefits it provides in tackling the problem of *bufferbloat*.

Bufferbloat is a situation arising due to the presence of large unmanaged buffers in the network. It results in increased latency and therefore, degrades the performance of delay-sensitive traffic. PIE algorithm tries to minimize the queuing delay by auto-tuning

its control parameters. However, with PI^2, this auto-tuning is replaced by just squaring the packet drop probability. Squaring the drop probability helps PI^2 offer a simplified design and improved performance without risking responsiveness and stability. In this dissertation, we have implemented a model for PI^2 in ns-3 and verified its correctness by comparing the results obtained from it to those obtained from the PIE model in ns-3. The results indicate that PI^2 offers a simple design and achieves similar or at times better responsiveness and stability than PIE.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet over the past few years has experienced significant growth in terms of the usage and the diversity of the applications. It has transformed from an experimental system into a gigantic and decentralized source of information. This success can be partly attributed to the congestion control mechanisms implemented in Transmission Control Protocol (TCP). These congestion control mechanisms are widely deployed in host operating systems and are extensively used by a variety of Internet applications. However, tremendous growth in the range of bandwidth, increase in Bit-Error Rates (BER) and increased diversity in applications have challenged the robustness of TCP congestion control mechanisms. These mechanisms must be able to accommodate and leverage the diversity in the characteristics of different networks (See Table 1.1), support a variety of application requirements and different traffic workloads (See Table 1.2). Thus, the need for optimizing these mechanisms has become extremely important.

Transport protocols with congestion control mechanisms are mainly classified into three categories: (i) *end-to-end protocols* (e.g., TCP Newreno [3]) that rely on implicit congestion signals such as packet loss, (ii) *network based protocols* (e.g., eXplicit Control Protocol (XCP) that rely on explicit feedback from the network and (iii) *end-to-end protocols with explicit feedback* (e.g., TCP with Active Queue Management/Explicit Congestion Notification) that rely on a few bits of explicit feedback from the network to

Table 1.1: Characteristics of different networks [1]

Network	**Capacity**	**Latency**	**BER**
Wired LANs (e.g., Ethernet)	10Mbps - 10Gbps	$<$ 1ms	$\leq 10^{-12}$
Data Centers	1Gbps - 1Tbps	100μs - 1ms	10^{-12}
Wired WANs	$\approx$10Mbps - 14Tbps	10ms - 300ms	10^{-12}
802.11 WLAN/Mesh Networks	$<$1Mbps - 600Mbps	1ms - 200ms	$>10^{-5}$
Cellular Data Networks (e.g., 3G)	384Kbps - 3Mbps	$\approx$ 100ms - 1s	10^{-5}
Satellite Networks	100Kbps - 155Mbps	250ms - 1s	10^{-10}

Table 1.2: Performance Requirements of Internet Applications [1]

Application	**Examples**	**Requirements**
Interactive	Voice over IP, Video Conferencing	Minimal latency, small jitter and less throughput variations
Short flows ($<$ 100KB)	Google Search, Facebook	Short response times
Medium sized transfers (100KB - 5MB)	Picasa, YouTube, Facebook photos	Low latency
Large transfers ($>$ 5MB)	Software updates, Video On-demand	Consistent high throughput

aid end-hosts in making congestion control decisions. While *end-to-end protocols* have performance limitations, *network based protocols* have been considered as hard to deploy because they need to maintain per-flow state at the routers. Since routers are complex and expensive devices, modifying them is a difficult task. Moreover, *network based protocols* require more bits for explicit feedback than are available in the IP header [1]. On the other hand, *end-to-end protocols with explicit feedback* have lower deployment complexity than *network based protocols* since they require modifications mainly at the end-hosts, with incremental support from the routers (e.g., deployment of AQM/ECN) [1].

The performance of TCP-based applications, apart from the congestion control mechanisms, critically depends on the choice of queue management scheme implemented in the routers. Queue management mechanisms control the length of the queues by dropping packets when necessary. Passive Queue Management (PQM) (e.g., drop-tail) is the most widely deployed queue management mechanism in Internet routers [4]. PQM does not

employ any preventive packet drop before the router buffer gets full and hence, is easy to deploy. However, due to the inherent problems of PQM such as global synchronization [5], lock-out [4], etc, Internet Engineering Task Force (IETF) recommends Active Queue Management (AQM) for the next generation of Internet routers [4]. Moreover, another limitation of PQM called persistently full buffer problem (referred to as bufferbloat [6]) has proved the acute need of widespread deployment of AQM. As a consequence, AQM algorithms are being re-investigated with a focus on controlling the queuing latency.

Unlike Random Early Detection (RED) [5] and its variants that attempt to control the queue length, new AQM algorithms such as Controlled Delay (CoDel) [7] and Proportional Integral controller Enhanced (PIE) [8] have been designed to minimize queue delay and retain high link utilization. Along the efforts in same direction, a new AQM algorithm called PI^2 [9] has been recently proposed, which offers similar or at times better responsiveness and stability than PIE, but has a simpler design and implementation.

1.1 Motivation

- Simulators are widely used for network performance evaluation. To the best of our knowledge, there does not exist a PI^2 implementation in popular network simulators like ns-2 [10] and ns-3. We believe that our implementation of PI^2 in ns-3 would provide an additional platform to the research community to verify its effectiveness and usefulness for future AQM architectures.

1.2 Goals of this dissertation

Our goals through this dissertation are twofold.

- First, to propose a new model for PI^2 algorithm in ns-3 [11] along with its design and implementation. The traffic control layer in ns-3 provides a strong support for queue

disciplines such as RED[5], ARED[12], CoDel[7], FqCoDel[13] and PIE[14]. Adding a model for PI^2, would add value to the simulator and the research community.

- Second, we plan to validate the implementation of our PI^2 model in ns-3 by comparing its results to those obtained from PIE model in ns-3 since both are expected to deliver near similar performance. Moreover, we believe that PI^2 would perform much better than PIE in some scenarios.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 provides insights into the Bufferbloat problem and associated challenges with Passive Queue Management (PQM). It provides a background into how Active Queue Management (AQM) algorithms address the problem of Bufferbloat via congestion avoidance and queue delay control. The chapter also discusses the state of the art AQM - PIE and PI^2.

Chapter 3 is a deep dive into the implementation of our PI^2 model in ns-3 simulator. The chapter provides a brief overview about ns-3 and implementation details of Traffic Control layer in ns-3. Furthermore, it also talks about the design and usage of queue disciplines present in ns-3. Finally, we propose the design and implementation details of our PI^2 model.

Chapter 4 details on the evaluation of our PI^2 model. The chapter describes the simulation setup and simulation scenarios under which PI^2 has been evaluated.The effectiveness of our PI^2 model is demonstrated by comparing the results obtained from our model to those obtained from the PIE model in ns-3.

Chapter 5 provides an overview on the ongoing research in the area of AQM and Transport Layer. We look into mechanisms such as DualQ[15] and TCP Prague[16]. The chapter talks about how the ongoing work in these projects align with the work done in this dissertation.

Chapter 6 presents the Conclusions and provides the possible future directions. The chapter also highlights the major contributions through this dissertation work and the areas where this work can be extended.

Chapter 2

Background

2.1 Bufferbloat - The Problem

With the ever increasing demand for interactive applications on the web such as real time video streaming, it is imperative for the network to perform well as to meet the requirements of latency sensitive applications. The networks today suffer from the problem of latency as a result of which the application performance is degraded. Bufferbloat is a situation arising due to the presence of large unmanaged buffers in the Internet. As a result of storage becoming inexpensive, the vendors were motivated to insert large buffers into the Internet without any appropriate testing. With large buffers in place, the TCP throughput is improved as the large buffers can accommodate more packets and thus allow the TCP sender to increase it's congestion window quickly. However, this gives rise to a new problem i.e, latency. With large buffers, the packets at the end of the queue have to undergo a large queuing delay. This degrades the performance of latency sensitive applications. Having large buffers is not an issue. However, having large unmanaged buffers is the real problem. Bufferbloat destroys the congestion avoidance mechanisms of TCP. This is attributed to large delays of Bufferbloat which incorrectly signals network congestion.

The problems arising due to Bufferbloat can be addressed by making use of Active

Queue Management (AQM) algorithms. These AQM actively manage the queue and keep the latency under control and thus helps to reduce the end-to-end delay. The next section describes the AQM algorithms in detail.

2.2 Active Queue Management (AQM)

The AQM schemes mitigate the problems arising due to large unmanaged buffers by proactively dropping the packets and explicitly signalling the source about congestion.These AQM schemes interact with the TCP to manage the amount of data to be kept in the network. The following sub section details the AQM algorithms and it's advantages.

2.2.1 AQM for Congestion Avoidance and Queue Delay Control

TCP provides congestion control by four main algorithms namely Slow Start, Additive Increase/Multiplicative Decrease (AIMD), Fast Retransmit and Fast Recovery. Slow start and AIMD are used for dynamically changing the size of a congestion window (*cwnd*). Slow Start increases the *cwnd* exponentially to quickly bring a newly started flow to the desired speed. In steady state, TCP uses AIMD to vary the *cwnd* in conjunction with fast retransmit and fast recovery. Fast Retransmit and fast Recovery are triggered in the event of a packet loss and are used to quickly recover from the state of congestion. These four algorithms, though modified several times in the recent past, have been the cornerstones of TCP congestion control research.

Congestion avoidance mechanisms differ from congestion control mechanisms, since former are proactive while latter are reactive. Though AIMD is also known as *Congestion Avoidance* algorithm, it is a misnomer since AIMD does not try to *avoid* congestion proactively [17]. Henceforth, we consider AIMD algorithms as *Congestion Control mechanisms* and AQM mechanisms as *Congestion Avoidance mechanisms* since AQM mechanisms proactively inform the sender about network state and *avoid* congestion.

Drop Tail queues have served the Internet for a very long time. With drop tail queues,

the packet gets dropped when the queue buffer is completely used. This approach has potential problems as described below.

- *Lock Out Problem* - the tail drop mechanism allows a single flow or a set of flows to occupy the entire queue. This situation can result in starvation for some connections.

- *Full Queues Problem* - the tail drop mechanism allows the queue to be full or almost full and signals the congestion only via packet drop. This leaves no room for the bursty traffic. Therefore, when the bursty traffic arrives, all the packets get dropped resulting in a problem of global synchronization.

- *Global Synchronization* - Traditional tail-drop gateways do not provide an early congestion notification. This leads to global synchronization, a phenomenon in which all senders sharing the bottleneck gateway reduce their sending rate at the same time, thereby under-utilizing the network resources.

- *Bufferbloat* - Since memory costs have reduced in the recent past, modern Internet routers are designed with extremely large buffers. As a result, today's Internet suffers from poor network performance because TCP variants implemented in modern operating systems are *end-to-end protocols* 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 drastically, thereby degrading the Quality of Service for delay sensitive applications such as DNS queries, Voice over IP (VoIP)VoIPVoice over Internet Protocol and other multimedia applications. This problem has been termed as *Bufferbloat*.

2.2.2 Motivation for AQM

AQM provides preventive measures to manage a queue and eliminates the limitations associated with PQM. AQM mechanisms have been extensively studied to monitor and

limit the growth of the queue at routers. These mechanisms avoid congestion by proactively informing the sender about congestion, either by dropping a packet or by marking a packet. The goals of AQM are specified as follows:

- *Reduce the number of packets dropped in the network* - by proactively dropping the packets the idea is to avoid full queues so as to accommodate the bursty traffic.
- *Reduce the end-to-end delay for the interactive applications* - by keeping the average queue size small, the end-to-end delay is reduced and this improves the performance of delay sensitive applications such as interactive video-audio sessions and other short lived web transfers.
- *Avoiding the problem of lock out* - AQM can avoid the problem of lock out by ensuring that there is buffer always available for the incoming packets.

Deployment of AQMs in the Internet can significantly improve the performance of the network. Today, AQM algorithms are being widely deployed in the Internet with a focus on controlling the queuing delay. In the context of AQM, it is important to understand that there are two classes of algorithms i.e, *queue management algorithms* and *scheduling algorithms*. The queue management algorithms manage the queue length by marking or dropping the packets. On the other hand, the scheduling algorithms are used to decide on which packet should be sent next so as to maintain fairness of bandwidth among the flows. Both the classes of algorithms can be used together. In this dissertation, we keep our discussion limited to the queue management algorithms. The next section describes the popular queue management algorithms.

2.3 AQM Algorithms

The AQM Working Group in the Internet Engineering Task Force (IETF) has set some general guidelines which must be followed by any AQM scheme. These guidelines are:

- Queue management algorithm must directly control the queuing latency rather than controlling the queue length. This is attributed to the queue drain rate which may vary with the flows.
- Queue management algorithm must maintain a right balance between latency and throughput. In an effort to achieve low latency, the algorithm must not degrade the link utilization. Similarly, in order to ensure high link utilization, the algorithm must not introduce high latency. A good AQM scheme is the one which provides low latency without affecting the link utilization.
- Queue management algorithm must be easy to implement, scale and deploy in both hardware and software.

2.3.1 Proportional Integral Enhanced (PIE)

PIE is now an Experimental RFC (RFC 8033) and also a recommended AQM algorithm for Data Over Cable Service Interface Specification (DOCSIS) cable modems (RFC 8034). It uses the Proportional Integral (PI) [18] controller to keep the queuing delay to a specified target value by updating the drop probability at regular intervals. Like RED, the deployment of PIE is simple. Moreover, PIE like CoDel uses queuing delay as the measure of congestion. Therefore, PIE combines the benifits of both RED and CoDel. PIE observes the trends in latency samples and determines the level of congestion. Following are the four major components of PIE:

Random Dropping: On packet arrival, PIE enqueues or drops the packet based on the drop probability, p which is obtained from drop probability calculation component. p is compared with a uniform random variable u. The packet is enqueued if $p < u$, otherwise dropped.

Random dropping is bypassed in PIE in the below conditions:

- When the old queuing delay is less than half of the target/reference delay.

- When the drop probability is not too high i.e, is less than 0.2.
- When the queue has less than a couple of packets.

Drop Probability Calculation: PIE updates the drop probability based on the trends in the latency samples. It observes if the latency is increasing or decreasing and accordingly updates the drop probability. This happens at every *tupdate* interval. It is calculated as [2]:

$p = \alpha * (qdelay - target) + \beta * (qdelay - qdelay\_old)$

where:

- *qdelay:* queuing delay during the current sample.
- *qdelay_old:* queuing delay during the previous sample.
- *target:* desired queuing delay.
- α *and* β*:* auto-tuning factors in PIE

PIE makes use of the Proportional Integral controller in order to control the queuing latency. The difference between the queuing delay during the current sample and the queuing delay during the previous sample helps understand whether the queuing delay is increasing or decreasing. α is the scaling factor which decides on how much it impacts the drop probability.

Moreover, the difference between the queuing delay during the current sample and the desired queuing delay helps determine whether the drop probability has stabilized. β is the scaling factor which decides on the adjustments should be made in order to stabilize the drop probability.

Queuing delay estimate: PIE uses Little's law [19] to estimate the current queuing delay.One of the other approach to measure the queuing delay is to make use of the time

stamp at packet enqueue and use the time stamp at packet dequeue.

Burst Tolerance: PIE allows the short term packet bursts to pass through for a specified interval. By default, the burst allowance is set to 150ms. Burst allowance is user configurable and can be set to a desired value.

Apart from the components listed above, PIE has several other enhancements such as ECN which helps to improve performance of PIE. Further details on this can be found in [14]

2.3.2 PI^2

Like PIE, PI^2 uses PI controller to keep the queuing delay within a specified target value. However, unlike PIE, it removes the auto-tuning feature from PIE and makes the drop decision by applying the squared drop probability. Furthermore, it extends PIE to support both Classic (e.g., Reno) and Scalable (e.g., Data Center TCP [20]) congestion controls. PI^2 performs packet classification and operates accordingly so as to support the coexistence of both Classic and Scalable congestion controls in the Internet.

Scalable TCP such as Data Centre TCP (DCTCP) can help to keep the queuing delay low without compromising other factors. DCTCP is being referred as *Scalable* because as the flow rate scales, the saw tooth variations in the rate grow with TCP friendly variants such as Reno and Cubic but not with DCTCP. Therefore, controls such as DCTCP are referred to as *Scalable* and controls such as Reno and Cubic are referred to as *Classic* [2].

DCTCP is too aggressive to exist alongside Classic TCP. The Classic traffic starves whem co-existing along with DCTCP. Therefore, till date DCTCP has only been deployed in controlled environments such as private data centers. However, DCTCP can be modified to make it deployable on the Internet. PI^2 authors while performing their research on making the family of Scalable controls deployable on the Internet, discovered that PI

controller is inherently linear when controlling Scalable congestion controls. However, this is not the case with Classic controls [2].

In the figure below, we can observe that for Classic controls such as TCP Reno p is encoded as a square of p'. We make the Classic drop probability proportional to the square of the Scalable marking probability.This is because we need to make the Reno flow rate equal the DCTCP [15].

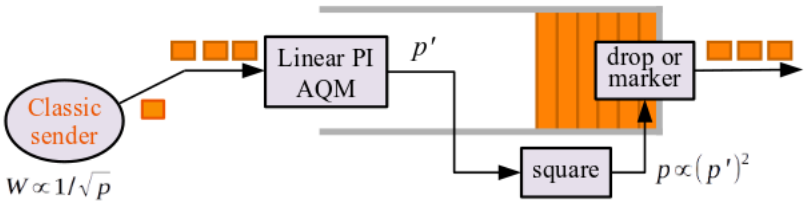


Figure 2.1: Replacing PIE with PI^2 [2]

On the other hand, for Scalable congestion controls such as DCTCP, the output p' can be used directly; no encoding is needed here.

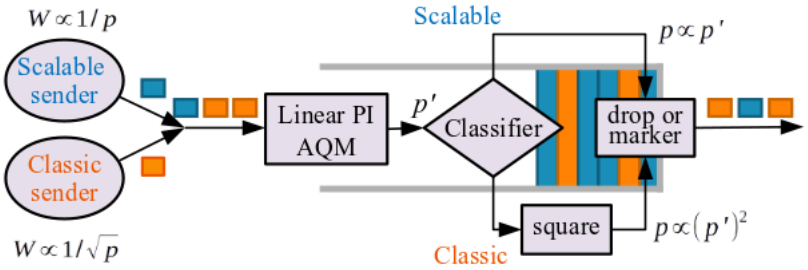


Figure 2.2: Co-existence of Scalable and Classic Traffic [2]

In this dissertation, we limit our discussion to implementing PI^2 for Classic TCP traffic in ns-3 because the differentiation between Classic TCP traffic and Scalable TCP traffic is achieved by using Explicit Congestion Notification (ECN) [21] which is not yet completely supported in the main line of ns-3.

2.3.3 Differences between PIE and PI^2

Drop decision: PIE drops the packets by comparing the drop probability, p with the uniform random variable, u. On the other hand, PI^2 drops the packets by comparing p^2 with u. Squaring the drop probability helps PI^2 offer a simple design and eliminate the corrective heuristics of PIE without the risking responsiveness and stability [9].

Burst allowance: PI^2 disables the burst allowance as to avoid an impact on the Data Center TCP fairness [9].

Other heuristics: PI^2 chooses to remove a few more heuristics which are a part of Linux Implementation of PIE. Details and justifications on removing these heuristics have been provided in [9].

Chapter 3

Implementation of PI^2 in ns-3

3.1 Network Simulator-3

ns-3 is a popular open source discrete-event network simulator used primarily for research and educational purposes. It maintains an open environment for the researchers to contribute their code to the community and provides a simulation engine to the users to conduct simulation experiments. The key advantage of ns-3 is that it can be used to perform large scale studies which would be impossible or very tedious to perform with real time systems. Users can perform large scale simulations in a highly controlled and reproducible environment and learn about how the networks operate.

While there are a number of network simulation tools available on the Internet, the good feature of ns-3 lies in it's modular design. ns-3 has been designed as a set of libraries which can work in conjuction with external software libraries. Though most of the work with ns-3 is on the command line using C++ or Python, ns-3 also works well with several animators and visualization tools.

It is important to note that ns-3 is not backward compatible with ns-2. ns-3 is a completely new simulator and most of the models from ns-2 have been ported on to ns-3. Considering the open nature, popularity, support on technical forums and the ongoing development in ns-3, it makes a great choice for this dissertation work.

3.2 Traffic Control Layer in ns-3

Similar to the Linux Traffic Control infrastructure, ns-3 introduces a Traffic Control layer which sits in between the network layer and the net device. It intercepts the outgoing packets flowing downwards from the network layer to the network device and the incoming packets flowing upwards from the network device to the network layer. Currently, the ns-3 Traffic Control module processes only the outgoing packets. The outgoing packets are enqueued into a queuing discipline where appropriate actions can be performed such as marking, scheduling, dropping or policing the packets.

3.2.1 Transmitting Packets

Both IPv4 and IPv6 interfaces make use of the `TrafficControlLayer` object to send the packets down to the Traffic Control layer for appropriate action. Once the packet processing is done and a suitable action is taken, the Traffic Control layer invokes the `NetDevice::Send ()` to send it on to the right net device.

3.2.2 Receiving Packets

Whenever an IPv4 or IPv6 interface is added to the L3 protocol, the callback chain is configured for the packet exchange as shown below.

```
NetDevice -> Node -> TrafficControlLayer -> IPv4,6 L3 Protocol
```

In ns-3, a node can send or receive the packets with the help of classes from L2 and L3 helpers. L2 helpers consist of the classes from the NetDevice side and L3 helpers consist of the classes from the Internet module. A detailed description of these classes can be

found in the official ns-3 documentation [1].

3.3 Queue disciplines in ns-3

3.3.1 Model Description

The Traffic Control layer receives the packets to be sent to the net device. Traffic Control layer on receiving the packets sends it to the queuing discipline a.k.a queue disc for scheduling and policing. Usually every net device has a root queue disc installed on it. However, this is not mandatory and in such cases the Traffic Control layer sends the packets directly to the net device. The queue disc can be a simple queue or even some complex hierarchical structure. Some of the elements that the queue disc may contan are:

- *Queue* - where the packets are actually stored for transmission.
- *Class* - where suitable treatment can be given to the packets based on the type of traffic.
- *Filter* - which determine the queue or class that applies to the packets.

The interaction between the queue disc and the traffic control layer is simple. After the packet is enqueued, the traffic control layer requests the queue disc to dequeue the packets till the threshold is reached. A netdevice may also request the queue disc to dequeue the packet when the transmission queue is empty and can even tell the queue disc to stop dequeuing when the transmission queue is almost full.

3.3.2 Model Design

Every queue disc in ns-3 extends from the `QueueDisc` class and must implement the following methods.

[1] https://www.nsnam.org/documentation/

- `bool DoEnqueue (Ptr<QueueDiscItem> item)`: To enqueue a packet in queue.
- `Ptr<QueueDiscItem> DoDequeue (void)`: To Dequeue a packet from the queue.
- `Ptr<const QueueDiscItem> DoPeek (void) const`: Peek a packet in the queue.
- `bool CheckConfig (void) const`: Check if the queue configuration is right.
- `void InitializeParams (void)`: Initialize parameters of queue disc.

`QueueDisc` base class implements the following functionalities.

- Provides methods to get or add queues, classes or filters.
- Provides a `Classify` method which helps to classify the packet by processing the list of filters.
- Provides methods to extract multiple packets from queue disc.

3.3.3 Usage and Helpers

The `InternetStackHelper` creates a `TrafficControlLayer` object attached to every node by default. When the IP address allocation is done for the node, the IPv4,6AddressHelper assigns the IP address to the interface and even installs a default queue disc, *PfifoFastQueueDisc* on the device. Now, let us look at how we can configure the type and attributes of the queue disc.

```
   TrafficControlHelper tch;
uint16_t handle = tch.SetRootQueueDisc ("ns3::PfifoFastQueueDisc");
tch.AddInternalQueues (handle, 3, "ns3::DropTailQueue",
"MaxPackets", UintegerValue (1000));
QueueDiscContainer qdiscs = tch.Install (devices);
```

The code above will add three internal queues to root queue disc which is of type *PfifoFastQueueDisc*.

3.4 Implementation of PI^2 Queue disc

This section provides insights into the implementation of PI^2 algorithm in ns-3. PI^2 algorithm has been implemented in a new class named `PiSquareQueueDisc` which is inherited from `QueueDisc`. `QueueDisc` is an abstract base class provided by the traffic control layer and has been subclassed to implement queuing disciplines such as Random Early Detection (RED) [5], PIE and CoDel. The following `virtual` methods provided in `QueueDisc` should be implemented in the respective classes of every queuing discipline:

- `bool DoEnqueue (Ptr<QueueDiscItem> item)`: enqueues or drops the incoming packet.
- `Ptr<QueueDiscItem> DoDequeue (void)`: dequeues the packet.
- `Ptr<const QueueDiscItem> DoPeek (void) const`: peeks into the first item of the queue.
- `bool CheckConfig (void) const`: checks the configuration of the queue disc.
- `void InitializeParams (void)`: initializes the parameters of the queue disc.

Figure 3.1 shows the relation between the parent class `QueueDisc` and the derived class `PiSquareQueueDisc`. In addition to the methods mentioned above, `PiSquareQueueDisc` implements the following two methods: `CalculateP` and `DropEarly`. These are specific to the PI^2 algorithm. Figure 3.2 depicts the interactions among the core components of PI^2.

On packet arrival, `DoEnqueue` is invoked which thereafter invokes `DropEarly` to check if the incoming packet should be dropped or enqueued. `CalculateP` calculates the drop

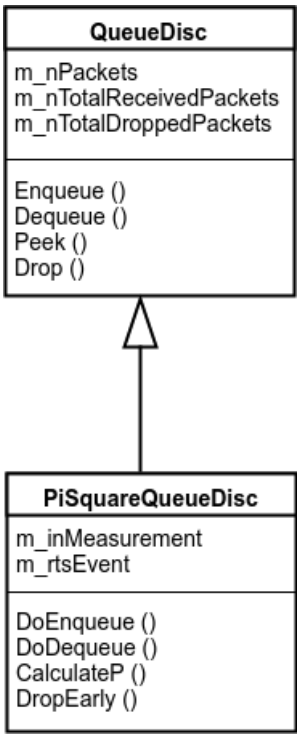


Figure 3.1: Class Diagram for PI^2 model in ns-3.

probability at regular intervals *(tupdate)*. `DoDequeue` is invoked on packet departure and estimates the average drain rate.

3.4.1 Dropping Packets Randomly

This functionality is implemented in `DoEnqueue` method in `PiSquareQueueDisc`. Like PIE, PI^2 drops the packets randomly based on the drop probability, p obtained from `CalculateP`. PI^2 applies the squared drop probability. The squaring is implemented by multiplying p by itself. `DropEarly` therefore, makes the drop decision based on the comparison between the squared drop probability and a random value u obtained from `UniformRandomVariable` class in ns-3. On packet arrival, `DoEnqueue` invokes `DropEarly`. The packet is enqueued if `DropEarly` returns false, otherwise dropped.

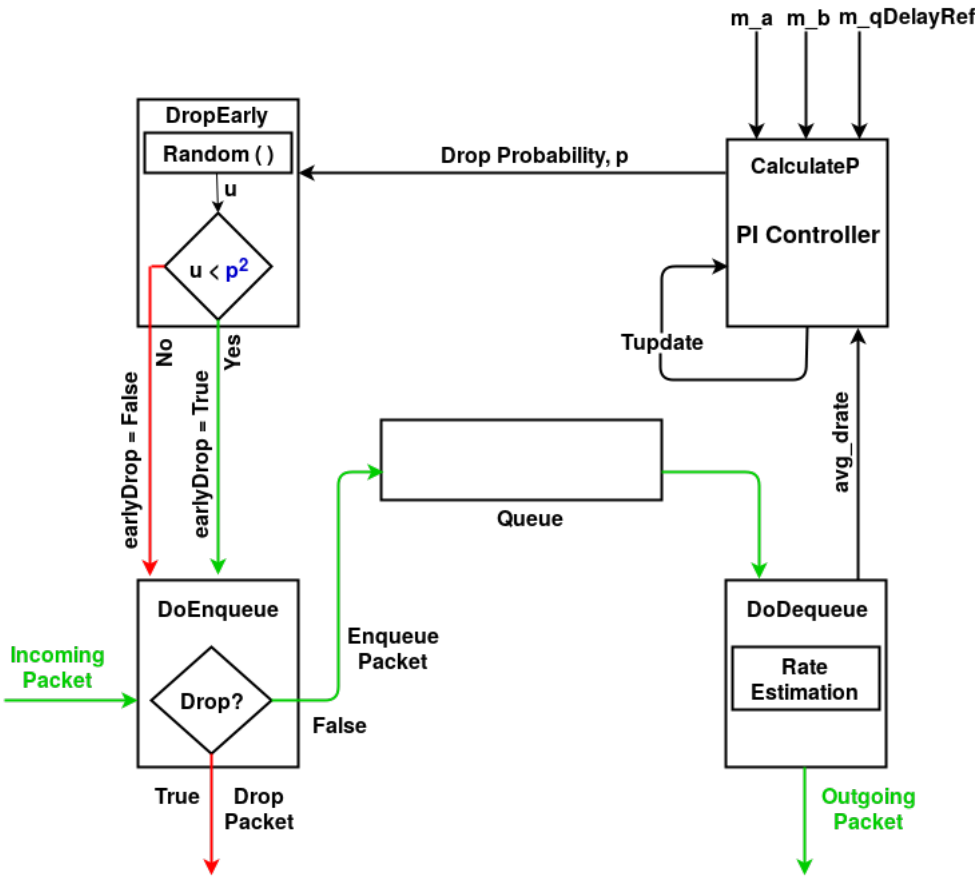


Figure 3.2: Interactions among components of PI^2 in ns-3.

3.4.2 Drop Probability Calculation

This functionality is implemented in `CalculateP` method in `PiSquareQueueDisc` class. PI^2 periodically calculates the drop probability based on the average dequeue rate *(m_avqDqRate)* and updates the old queuing delay *(m_qDelayOld)*. Table 3.2 provides a list of parameters used in the calculation of drop probability. Variables used in PI^2 Linux implementation are mapped onto corresponding variables used in ns-3 model.

Table 3.1: PI^2 variables to calculate p.

PI^2 variable	**ns-3 variable**
tupdate	m_tUpdate
qdelay	m_qDelay
qdelay_old	m_qDelayOld
target	m_qDelayRef
alpha	m_a
beta	m_b
avg_dq_rate	m_avqDqRate

3.4.3 Estimation of Average Departure Rate

This functionality is implemented in `DoDequeue` method in `PiSquareQueueDisc` class. On packet departure, `DoDequeue` calculates the average departure rate *(m_avqDqRate)* if the queue is in the measurement cycle. Table 3.2 provides a list of parameters required to calculate *m_avqDqRate*. Variables used in PI^2 Linux implementation are mapped onto corresponding variables used in ns-3 model.

Table 3.2: PI^2 variables to estimate *avg_drate.*

PI^2 variable	**ns-3 variable**
qlen	m_packets / m_bytesInQueue
QUEUE_THRESHOLD	m_dqThreshold
dq_count	m_dqCount
dq_tstamp	m_dqStart
dtime	tmp
ϵ	fixed to 0.5

All the variables are set internally and updated by PI^2. The only configurable parameter provided by the user is *m_qDelayRef*.

3.4.4 Other Heuristics

PI^2 implementation of Linux removes scaling heuristics and many other heuristics as compared to Linux implementation of PIE. This is because the basis on which these heuristics were chosen is not documented anywhere. In line with the PI^2 Linux implementation, our PI^2 model in ns-3 removes the following heuristics when compared to PIE ns-3 model.

- In PIE, when the probability is below 0.2 and the queue delay is below half of the target delay, no marking or dropping is applied. This has been removed from PI^2. If enabled, the threshold for the probability should be 0.45[2].
- In PIE, when the probability is higher than 0.1, Δ p is set to 0.02. This has been disabled. According to the PI^2 authors, further investigation is required before it is enabled.
- In PIE, when the queue delay is greater than 250ms the Δ p is set to 0.02. This too has been removed from the PI^2 model of ns-3.

PI^2 chooses to remove a few more heuristics which are a part of Linux Implementation of PIE. Details and justifications on removing these heuristics have been provided in [9].

3.4.5 Limitations

- The current PI^2 model in ns-3 is implemented to work only with Classic TCP traffic and without ECN. This is because ECN and Scalable TCP such as DCTCP are currently not available in the mainline of ns-3.

Chapter 4

Model Validation

We have designed a test suite with unit tests for verifying the implementation of PI^2 model in ns-3, which is a mandatory step in the process of merging new models into `ns-3-dev`. Our implementation of PI^2 model along with test suite is currently under review.[1]

To further verify the correctness of our implementation, we compare the results obtained from our model of PI^2 to those obtained from the PIE model in ns-3. The simulation scenarios considered for comparison are:

- Light TCP traffic, 5 TCP Sources.
- Heavy TCP traffic, 50 TCP Sources.
- Mix TCP and UDP traffic, 5 TCP + 2 UDP.
- CDF of Queuing Delay.

These scenarios are in line with the ones used by the authors of PI^2 [9]. However, due to the unavailability of CUBIC [22] and ECN models in ns-3, we have used TCP NewReno [3] without ECN for the evaluation. Our aim is to ensure that our implementation exhibits the key characteristics of the PI^2 algorithm. The performance parameters used for comparison are throughput and queue delay. Table 4.1 presents the details of simulation setup.

[1] https://codereview.appspot.com/314290043/

4.1 Simulation Setup

Table 4.1: Simulation setup.

Parameter	**Value**
Topology	Dumbbell
Bottleneck RTT	76ms
Bottleneck buffer size	200KB
Bottleneck bandwidth	10Mbps
Bottleneck queue	PI^2
Non-bottleneck RTT	2ms
Non-bottleneck bandwidth	10Mbps
Non-bottleneck queue	DropTail
Mean packet size	1000B
TCP	NewReno
target	20ms
tupdate	30ms
alpha	PIE - 0.125, PI^2 - 0.3125
beta	PIE - 1.25, PI^2 - 3.125
dq_threshold	10KB
Application start time	0s
Application stop time	99s
Simulation stop time	100s

It must be noted that PI^2 makes use of higher values for α and β as compared to PIE. The values are 2.5 times higher than PIE. This makes PI^2 much more responsive in comparison to PIE. The total base RTT is kept to 100ms. (i.e, 76 ms of Bottleneck RTT, 2ms + 2ms of Non-bottleneck RTT, 20ms of target delay). This is because the values of α and β

derived from the theoretical analysis are stable only up to the base RTT of 100ms.

4.2 Scenario 1: Light TCP Traffic

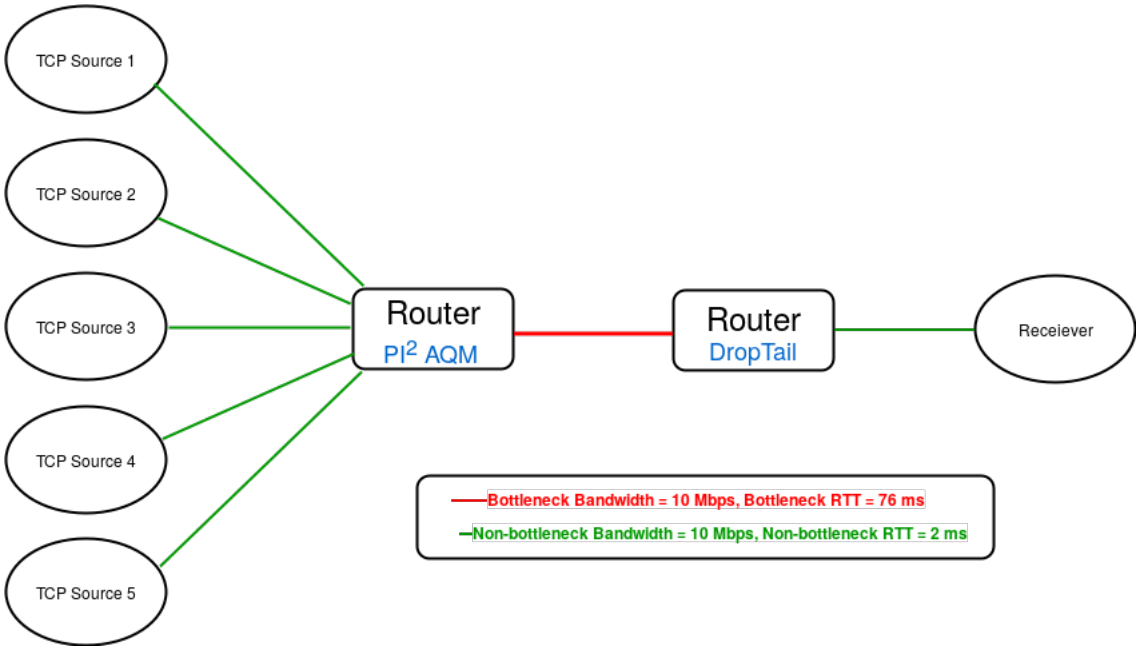
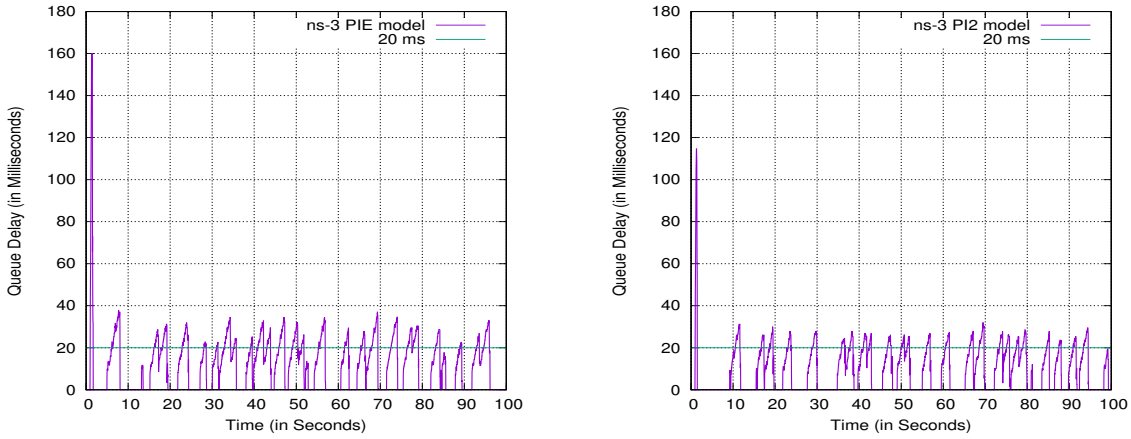


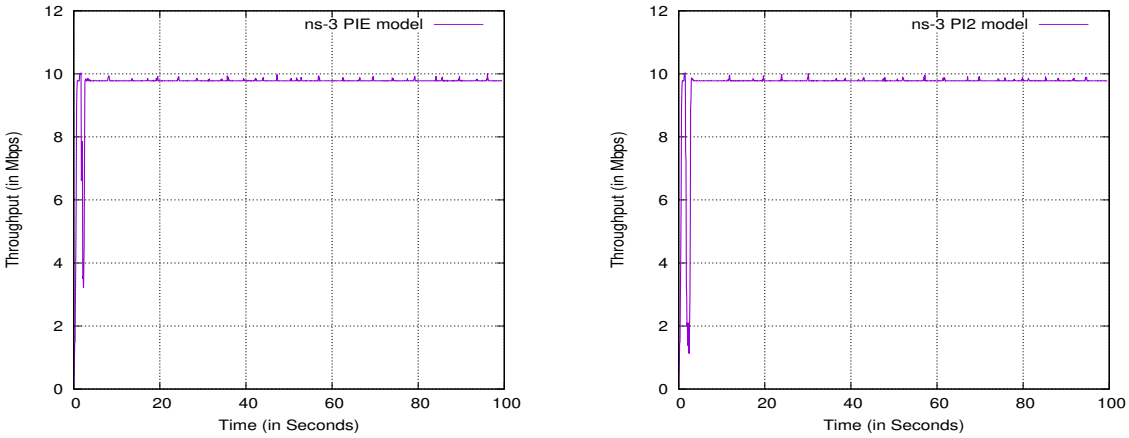
Figure 4.1: Light TCP Traffic Dumbbell Topology.



(a) ns-3 PIE (b) ns-3 PI^2

Figure 4.2: Queue delay with light TCP traffic.

In this scenario, a dumbbell topology is used to simulate 5 TCP flows that start at the same time and pass through the same bottleneck link. Other simulation parameters



(a) ns-3 PIE (b) ns-3 PI^2

Figure 4.3: Link throughput with light TCP traffic.

are set as shown in Table 4.1. Figure 4.2 shows the variations in queuing delay over time. We can observe the initial peak in the instantaneous queuing delay for both PI^2 and PIE results. This is attributed to the burst traffic generated due to all 5 TCP sources starting at the same time. Moreover, it can be observed that PI^2 to some extent provides better control on the queuing delay. The initial peak in PIE goes to 160ms. However, PI^2 keeps it under 120ms. Both PI^2 and PIE bring down the queuing delay quickly and maintain it around the reference delay for the rest of the simulation. We can infer that both PI^2 and PIE produce similar results and control the queuing delay to a desired target value. However, during the burst it can be observed that PI^2 offers better control in comparison to PIE.

Figure 4.3 shows the instantaneous throughput. Initially the throughput degrades due to packets being dropped by PI^2 and PIE in an effort to control the queuing delay and maintain it around the desired target delay. It can be noted that throughput degradation with PI^2 is slightly more because of its tighter control on the queue delay. Nevertheless, both algorithms yield similar performance for the rest of the simulation.

4.3 Scenario 2: Heavy TCP Traffic

This scenario is same as Scenario 1, but configures 50 TCP flows instead of 5 TCP flows. All TCP sources start transmission at the same time. Figure 4.5 shows the variations in queuing delay over time. Similar to previous scenario, the initial peak in PIE goes to 160ms. However, PI^2 keeps it under 120ms.Moreover, we can observe that PI^2, like PIE, quickly brings down the queuing delay and keeps it around the desired target value despite heavy TCP traffic. It can also be noted that the oscillations in the queue delay are much more in PIE as compared to PI^2.

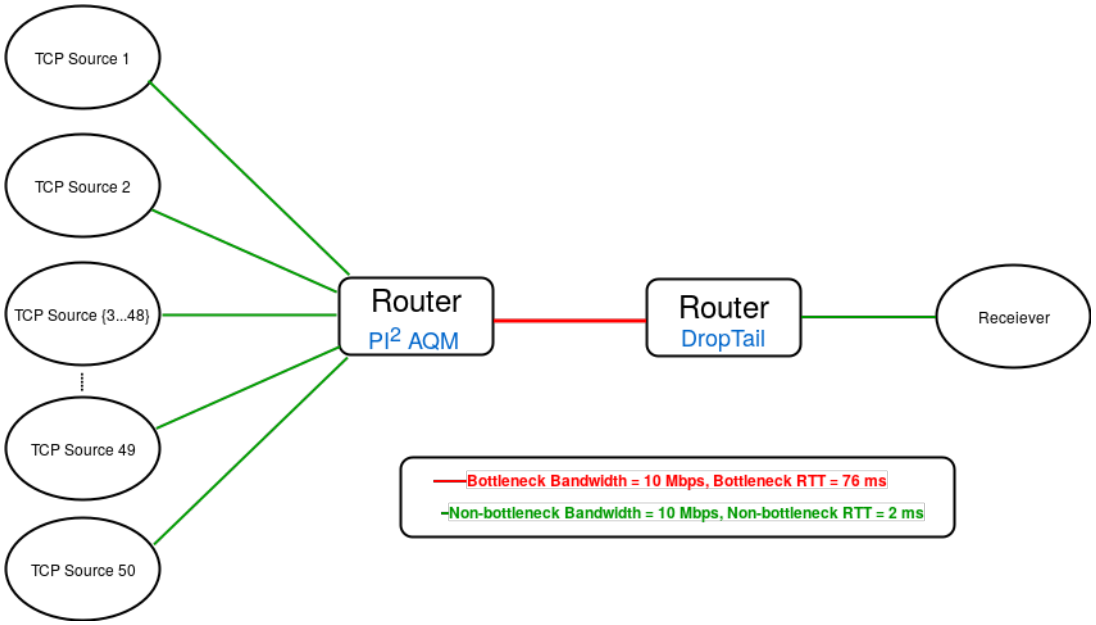
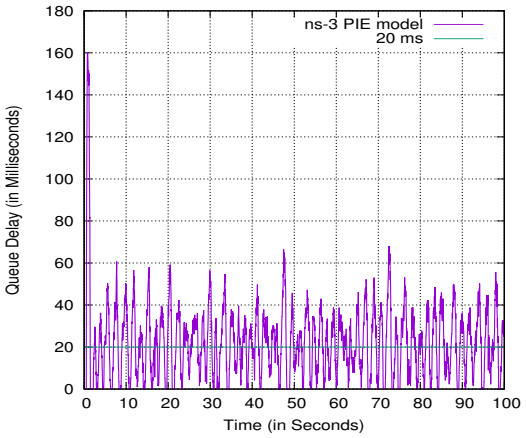


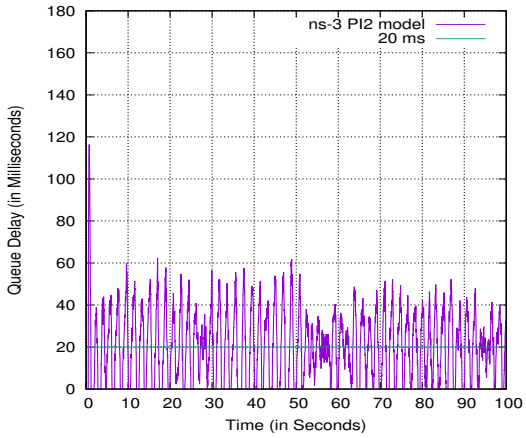
Figure 4.4: Heavy TCP Traffic Dumbbell Topology.

The results are similar to those obtained for Scenario 1. Although the amount of burst in this scenario is much larger than that in Scenario 1, PI^2 continues to perform better than PIE in controlling the queue delay.

Figure 6 shows the instantaneous throughput. Unlike previous scenario, we observe that the link throughput is not penalized in either PIE or PI^2 in this experiment, mainly due to a large number of TCP flows sharing the link capacity.

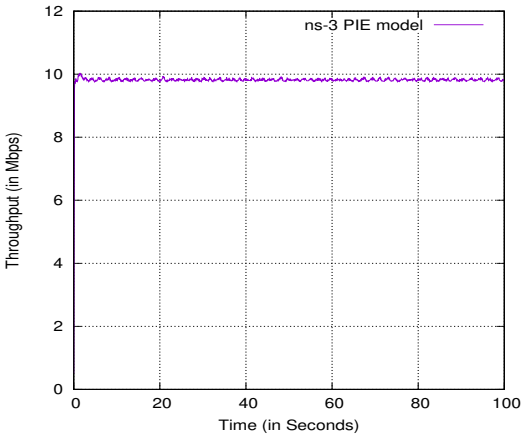


(a) ns-3 PIE

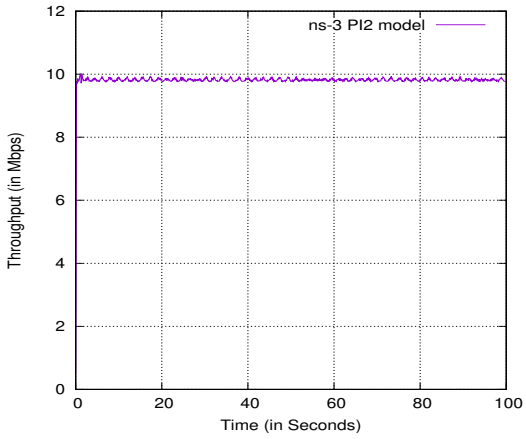


(b) ns-3 PI^2

Figure 4.5: Queue delay with heavy TCP traffic.



(a) ns-3 PIE



(b) ns-3 PI^2

Figure 4.6: Link throughput with heavy TCP traffic.

4.4 Scenario 3: Mix TCP and UDP Traffic

This simulation scenario is to determine whether PI^2 can function normally with unresponsive UDP traffic. We use dumbbell topology and simulate 5 TCP and 2 UDP flows passing through the same bottleneck link. All TCP and UDP flows begin transmission at the same time. UDP sources transmit at a rate of 10 Mbps. Other simulation parameters are same as mentioned in Table 4.1.

We observe that the results obtained for PI^2 and PIE are similar. Figure 4.8 shows that PI^2 and PIE control the queuing delay successfully even in the presence of unresponsive

UDP traffic.

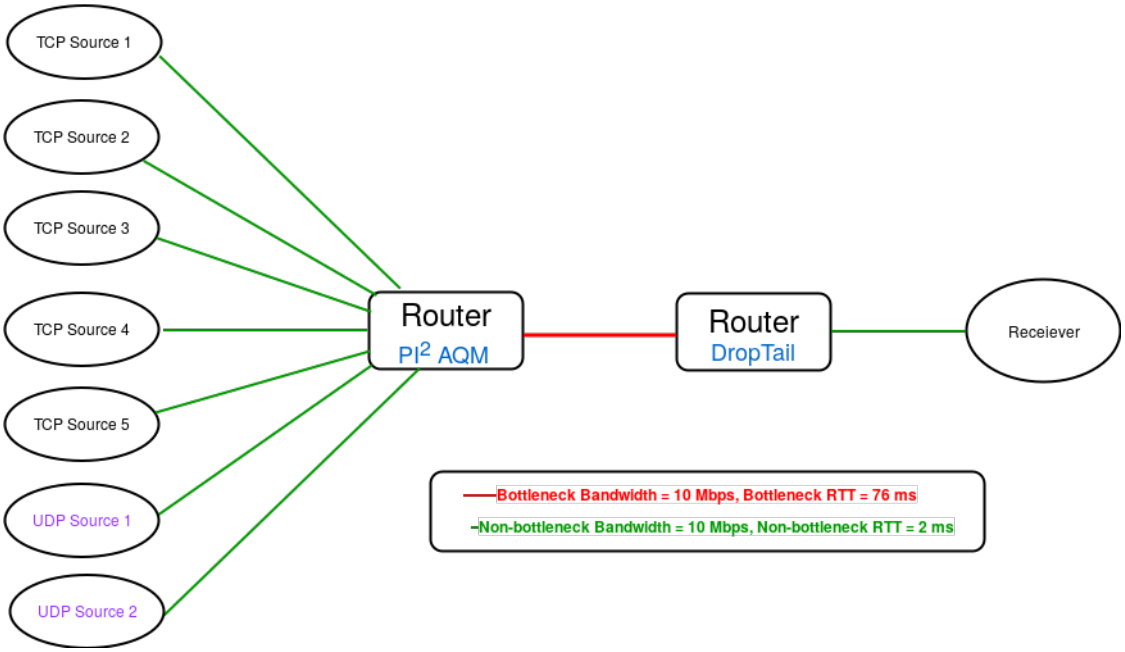
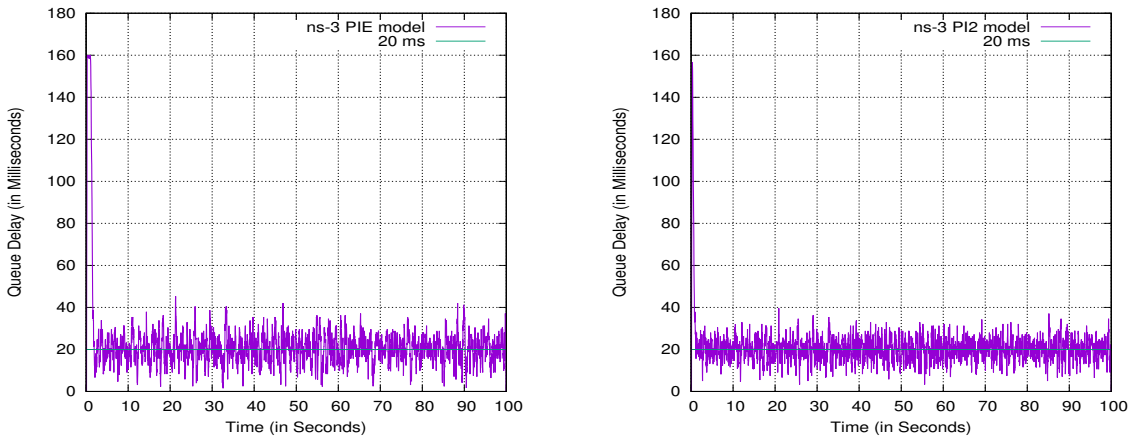


Figure 4.7: Mix TCP + UDP Traffic Dumbbell Topology.

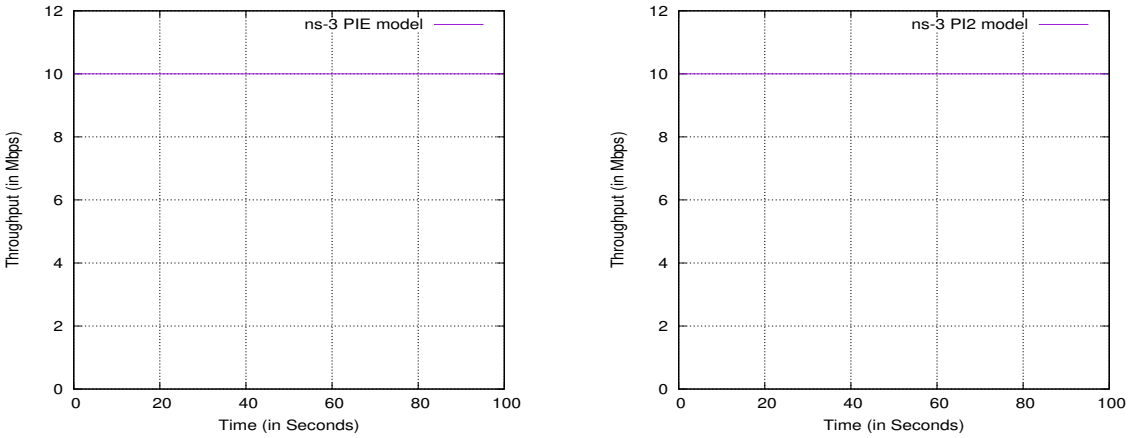


(a) ns-3 PIE

(b) ns-3 PI^2

Figure 4.8: Queue delay with mix TCP and UDP traffic.

Moreover, in Figure 4.9 we can observe that the bottleneck bandwidth is completely utilized with both the algorithms.

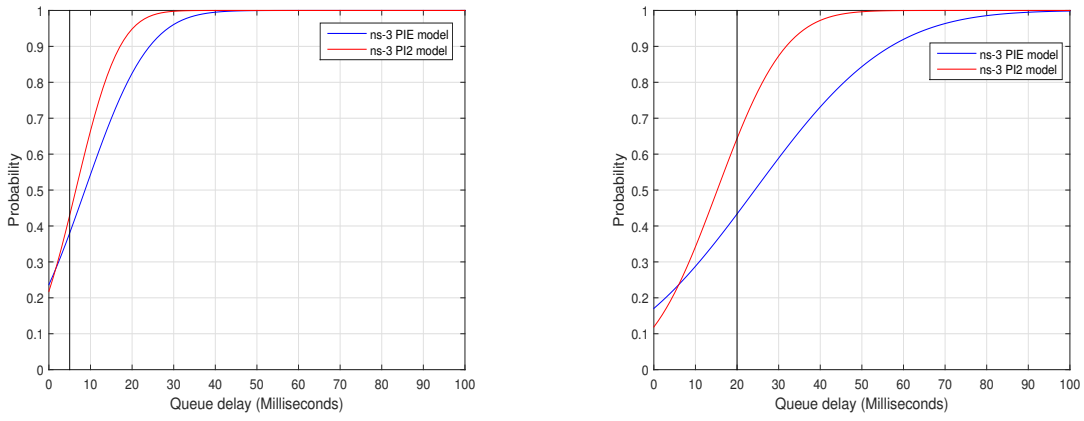


(a) ns-3 PIE (b) ns-3 PI^2

Figure 4.9: Link throughput with mix TCP and UDP traffic.

4.5 Scenario 4: CDF of Queuing Delay

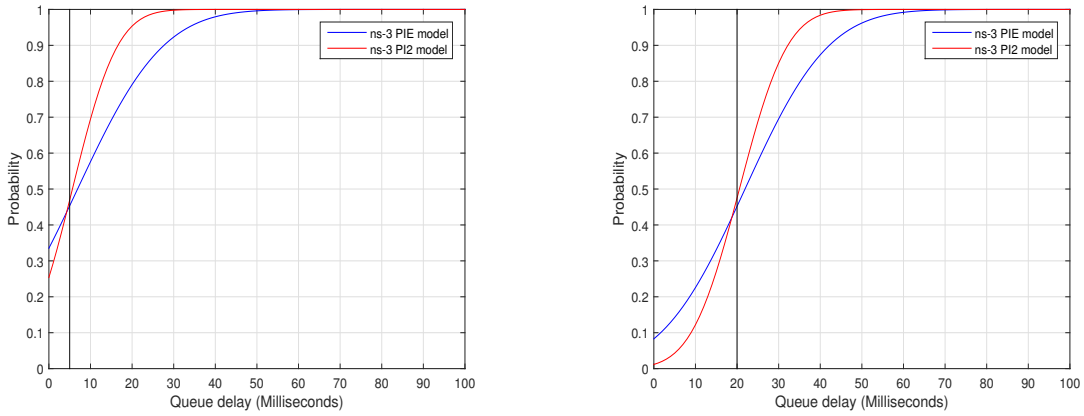
In this scenario, we compare the CDF of queuing delay obtained for PI^2 and PIE. We conduct two experiments using different traffic loads as done in [9]. First, we use 20 TCP flows with target delay of 5ms and 20ms. Next, we use a mix traffic consisting of 5 TCP and 2 UDP flows with target delay of 5ms and 20ms. Rest of the simulation parameters are same as listed in Table 4.1.



(a) 20 TCP Flows and target delay = 5ms (b) 20 TCP Flows and target delay = 20ms

Figure 4.10: CDF of queuing delay with 20 TCP flows.

Figure 4.10 and 4.11 show the CDF plots comparing the queuing delay of PI^2 and PIE.



(a) 5 TCP + 2 UDP Flows and target delay = 5ms

(b) 5 TCP + 2 UDP Flows and target delay = 20ms

Figure 4.11: CDF of queuing delay with 5 TCP and 2 UDP flows.

In line with the observations made by the authors of PI^2, we observe that PI^2 performs no worse and infact, offers notable improvement over PIE in some cases. We note that PI^2 clearly outperforms PIE when the traffic is TCP-only. The margin of improvement slightly reduces when TCP and UDP traffic coexist.

4.6 Summary

To summarize, it can be observed that PI^2 with its simple design can deliver similar or in some cases better performance in comparison to PIE. From the aforementioned experiments, we can observe that the performance of PI^2 is better than PIE when the traffic is TCP only. Moreover, in presence of unresponsive traffic such as UDP, PI^2 still manages to deliver similar performance as PIE.

Results from the simulation study confirm the correctness of our PI^2 model implementation in ns-3. Moreover, it can be noted that the results are inline with the authors of PI^2 [2].

Chapter 5

Future of Internet Transport

The services offered by the Transport protocols such as TCP and UDP are inadequate to meet the emerging demands of applications. Latency is a critical performance factor to most of the applications today. Much has been done in order to reduce the propagation time by using Content Delivery Networks (CDN) and placing servers closer to the users. However, queuing latency remains a major issue.

AQM helps to control the queue delay and avoid congestion. However, even with the perfectly tuned AQM, the sawtoothing rate of TCP will either cause queuing delay to vary or cause the link to be under-utilized. Protocols such as DCTCP have shown that small changes to the TCP can help eliminate the sawtooth varying rate of TCP. DCTCP is Scalable TCP as it keeps the queuing delay low without affecting or compromising the link utilization. Unfortunately, these Scalable TCP cannot co-exist with Classic TCP such as TCP Reno and Cubic. This is attributed to aggressive congestion control in Scalable TCP which starves the Classic TCP flows.

5.1 DualQ

The DualQ Coupled AQM architecture solves the problem of coexistence of the Scalable and Classic traffic. The Coupled AQM ensures that a flow runs at the same rate whether

it uses DCTCP or TCP friendly controls such as Reno or Cubic and it does so without inspecting transport layer flow identifiers. The AQM exploits the behaviour of scalable congestion controls like DCTCP so that every packet in every flow sharing the queue for Scalable TCP traffic can be served with very low latency [15].

DualQ algorithm has 2 main aspects:

- Coupled AQM that addresses throughput fairness between Classic and Scalable flows.
- DualQ structure provides latency separation for Scalable flows so that they do not undergo the queuing delay due to Classic flows.

In order to make Scalable flows run at around the same rate as Classic flows, the Coupled AQM applies the approach adopted in PI^2 AQM. Therefore, the DualQ Framework implementation in ns-3 in future can make use of our PI^2 queuing discipline.

5.2 TCP Prague

tcpPrague is an effort to coordinate the implementation and standardization of TCP Prague across all platforms. TCP Prague is considered to be an evolution of DCTCP and is expected to function along with other TCP variants. DCTCP so far has only been used in privately controlled environments such as data centers. This is attributed to aggressive congestion control of DCTCP as compared to TCP friendly variants such as Reno and Cubic. TCP Prague is expected to:

- Take the full advantage of ECN in the network.
- Marking Frequency never changing with flow rate.

TCP Prague is still a work in progess. However, modules such as DualQ have been finalised and briefly experimented. Both TCP Prague and DualQ AQM framework implementation in ns-3 would make use of our PI^2 queuing discipline.

Recent developments such as DualQ and TCP Prague reflect the active interest of the research community in Active Queue Management (AQM).Therefore, availability of PI^2 queuing discipline in network simulators like ns-3 is crucial to aid this future development.

Chapter 6

Conclusions

In this dissertation, we have discussed about the challenges imposed due to Bufferbloat and the problems associated with PQM. We have seen how AQM deployed at the routers can help mitigate the excess queuing latency introduced due to Bufferbloat. Moreover, we have looked into state of the art AQM - PI^2 and PIE. Furthermore, we have designed, implemented and tested a model in ns-3 for PI^2 AQM which provides a simple design and implementation in contrast to PIE. Finally, we have discussed on how the Transport layer in the Internet is evolving and how our work fits into the ongoing research.

6.1 Major Contributions

To summarize, our major contributions in this dissertation work are as outlined below.

- Designed and Implemented a model for PI^2 AQM in the Traffic Control layer in ns-3. In our opinion, implementation of PI^2 in ns-3 would provide an additional platform to the research community to verify its effectiveness and usefulness for future AQM architectures. Our proposed model is based on the Linux code of the authors of PI^2.[1]

[1] https://github.com/olgabo/dualpi2/blob/master/sch_pi2/sch_pi2.c

- Designed a test suite for evaluating the working of PI^2 algorithm. The test suite covers all the basic functionality such as verifying the attribute settings of PI^2 algorithm and performing basic enqueue / dequeue of packets.

- Evaluated the effectiveness of our implementation by comparing the results obtained from PI^2 model of ns-3 to those obtained from the PIE model of ns-3. The results indicate that PI^2 offers a simple design and achieves similar or at times better responsiveness and stability than PIE.

- Designed and Implemented model of PI^2 has been submitted to the ns-3 developers and is currently under review. The same can be accessed here. [2]

  Moreover, we have published a research paper at 1st International Workshop on Future of Internet Transport co-located with IFIP Networking 2017 at KTH Royal Institute of Technology, Stockholm, Sweden. The paper will appear in the symposium proceedings published by IFIP and will be submitted to IEEE Xplore Digital Library [23].

6.2 Future Work

Some of the major areas where this work can be extended is outlined below.

- At the time of writing this dissertation, ECN is currently not supported in the main line of ns-3. PI^2 uses ECN in order to distinguish between the Classic and Scalable TCP flows. Therefore, on the availability of ECN in ns-3, we can extend our PI^2 implementation to work with ECN.

- Similarly, Scalable Congestion Controls such as DCTCP are currently in development and haven't been merged to mainline of ns-3. Therefore, on availability of Scalable TCPs such as DCTCP, we can extend our implementation of PI^2 to work with Scalable Congestion Controls.

[2] https://codereview.appspot.com/314290043/

- Direct Code Execution (DCE) feature of ns-3 allows ns-3 to make use of the underlying Linux stack for our experimentation. Currently, DCE supports only older Linux kernels. On the availability of support for newer Linux kernels, we can compare PI^2 model in ns-3 with PI^2 model implemented in the Linux kernel.

Appendix A

Abbreviations

Short Term	Expanded Term
TCP	Transmission Control Protocol
BER	Bit Error Rates
XCP	eXplicit Control Protocol
AQM	Active Queue Management
ECN	Explicit Congestion Notification
IETF	Internet Engineering Task Force
PQM	Passive Queue Management
CoDel	Controlled Delay
PIE	Proportional Integral Enhanced
PI	Proportional Integral
RED	Random Early Detection
ARED	Adaptive Random Early Detection
FqCoDel	FlowQueue Controlled Delay
UDP	User Datagram Protocol
DCTCP	Data Center Transmission Control Protocol
DOCSIS	Data Over Cable Service Interface Specification
RTT	Round Trip Time
CDF	Cumulative Distribution Function
AIMD	Additive Increase Multiplicative Decrease
CDN	Content Delivery Network

Appendix B

List of Publications

The following paper was presented at 1st International Workshop on Future of Internet Transport co-located with IFIP Networking 2017 at KTH Royal Institute of Technology, Stockholm, Sweden. The paper will appear in the symposium proceedings published by IFIP and will be submitted to IEEE Xplore Digital Library.

- Tahiliani, R. P., & Tewari, H. Implementation of PI^2 Queuing Discipline for Classic TCP Traffic in ns-3.

Appendix C

Pseudo code for PI^2

This section details the major components of PI^2 along with their pseudo code. Following are the major components of PI^2 in ns-3:

- DoEnqueue - invoked at every packet arrival.
- DropEarly - apply the squared drop probability and compare with random variable to make a decision.
- CalculateP - calculates drop probability at every *tupdate* interval.
- DoDequeue - invoked at every packet departure.

Table C.1: Configurable parameters in PI^2 model.

PI^2 parameter	**Description**
m_qDelayRef	Reference Queuing Delay

Table C.2: Internal parameters in PI^2 model.

PI^2 parameter	**Description**
m_Alpha set to 0.3125	Weight in the drop probability calculation.
m_Beta set to 3.125	Weight in the drop probability calculation.
m_tUpdate	Interval to calculate the drop probability.
m_qDelayOld set to 0	Previous queuing delay.
m_dropProb	Packet drop probability.

DoEnqueue function: invoked on every packet enqueue.DoEnqueue checks if the current queue length is greater than the set threshold. If yes, it drops the packet. If not, DoEnqueue invokes DropEarly to check if the packet should be dropped or enqueued.

```
DoEnqueue (Ptr<QueueDiscItem> item)
{
  uint32_t nQueued = GetQueueSize ();

  if ((GetMode () == Queue::QUEUE_MODE_PACKETS &&
       nQueued >= m_queueLimit)
       || (GetMode () == Queue::QUEUE_MODE_BYTES
           && nQueued + item->GetPacketSize () > m_queueLimit))
    {
      // Drops due to queue limit: reactive
      Drop (item);
      return false;
    }
  else if (DropEarly (item, nQueued))
```

```
  {
    // Early probability drop: proactive
    Drop (item);
    return false;
  }
```

DropEarly function: makes the drop decision based on the comparison between the squared drop probability and a random value, u obtained from `UniformRandomVariable` class in ns-3. If u is greater than the squared drop probability, then the packet is enqueued. Otherwise, the packet is dropped.

The squaring of drop probability can be implemented in two ways. First, the drop probability p can be multiplied by itself. Second, is to compare p with a maximum of two random variables during the drop decision.We adopt the first approach as it's easier to perform in a software implementation. The latter can be preferred for a hardware implementation [2].

```
DropEarly (Ptr<QueueDiscItem> item, uint32_t qSize)
{
  double p = m_dropProb;
  uint32_t packetSize = item->GetPacketSize ();

  if (GetMode () == Queue::QUEUE_MODE_BYTES)
    {
      p = p * packetSize / m_meanPktSize;
    }
  bool earlyDrop = true;
  double u =  m_uv->GetValue ();
```

```
  if (GetMode () == Queue::QUEUE_MODE_BYTES
    && qSize <= 2 * m_meanPktSize)
    {
      return false;
    }
  else if (GetMode () == Queue::QUEUE_MODE_PACKETS
        && qSize <= 2)
    {
      return false;
    }
  // Apply the squared drop probability
  if (u > p * p)
    {
      earlyDrop = false;
    }
  if (!earlyDrop)
    {
      return false;
    }
  return true;
}
```

CalculateP: Calculates the drop probability at every *tupdate* interval. CalculateP is responsible for estimating the current queuing delay based on Little's law, auto-tuning the scaling factors, calculating the drop probability and updating the old queuing delay.

The drop probability is updated at *tupdate* interval based on how far the current

latency is away from the target and whether the queuing latency is currently increasing or decreasing.

```
CalculateP ()
{
  Time qDelay;
  double p = 0.0;
  bool missingInitFlag = false;
  if (m_avgDqRate > 0)
    {
      qDelay = Time (Seconds (GetInternalQueue (0)->
                  GetNBytes () / m_avgDqRate));
    }
  else
    {
      qDelay = Time (Seconds (0));
      missingInitFlag = true;
    }
  m_qDelay = qDelay;
  // Calculate the drop probability
  p = m_a * (qDelay.GetSeconds () - m_qDelayRef.GetSeconds ())
  + m_b * (qDelay.GetSeconds () - m_qDelayOld.GetSeconds ());
  p += m_dropProb;

  // For non-linear drop in prob
  if (qDelay.GetSeconds () == 0 && m_qDelayOld.GetSeconds () == 0)
    {
```

```
      p *= 0.98;
    }

  m_dropProb = (p > 0) ? p : 0;

  if ( (qDelay.GetSeconds () < 0.5 * m_qDelayRef.GetSeconds ()) &&
  (m_qDelayOld.GetSeconds () < (0.5 * m_qDelayRef.GetSeconds ())) &&
  (m_dropProb == 0) && !missingInitFlag )
    {
      m_dqCount = -1;
      m_avgDqRate = 0.0;
    }

  m_qDelayOld = qDelay;
  m_rtrsEvent = Simulator::Schedule (m_tUpdate,
                &PiSquareQueueDisc::CalculateP, this);
}
```

DoDequeue: is invoked on every packet departure and estimates the average departure rate. The average departure rate is estimated only when the queue is in the measurement cycle.

```
PiSquareQueueDisc::DoDequeue ()
{
  if (GetInternalQueue (0)->IsEmpty ())
    {
      NS_LOG_LOGIC ("Queue empty");
```

```
    return 0;
  }

Ptr<QueueDiscItem> item = StaticCast<QueueDiscItem>
      (GetInternalQueue (0)->Dequeue ());
double now = Simulator::Now ().GetSeconds ();
uint32_t pktSize = item->GetPacketSize ();

// if not in a measurement cycle and the queue has built
// up to dq_threshold, start the measurement cycle

if ( (GetInternalQueue (0)->GetNBytes () >=
      m_dqThreshold) && (!m_inMeasurement) )
  {
    m_dqStart = now;
    m_dqCount = 0;
    m_inMeasurement = true;
  }
if (m_inMeasurement)
  {
    m_dqCount += pktSize;

    // done with a measurement cycle
    if (m_dqCount >= m_dqThreshold)
      {

        double tmp = now - m_dqStart;
```

```
  if (tmp > 0)
    {
      if (m_avgDqRate == 0)
        {
          m_avgDqRate = m_dqCount / tmp;
        }
      else
        {
          m_avgDqRate = (0.5 * m_avgDqRate) +
                (0.5 * (m_dqCount / tmp));
        }
    }

  // restart a measurement cycle if there is enough data
  if (GetInternalQueue (0)->GetNBytes () > m_dqThreshold)
    {
      m_dqStart = now;
      m_dqCount = 0;
      m_inMeasurement = true;
    }
  else
    {
      m_dqCount = 0;
      m_inMeasurement = false;
    }
}
```

```
    }

  return item;
}
```

Bibliography

[1] Ihsan Ayyub Qazi. *An efficient framework of congestion control for next-generation networks*. University of Pittsburgh, 2010.

[2] Koen De Schepper, Olga Bondarenko, Jyh Tsang, and Bob Briscoe. PI2 AQM for Classic and Scalable Congestion Control. Sept. 2016.

[3] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. Technical report, 2012.

[4] M. Hassan and R. Jain. *High Performance TCP/IP Networking*, volume 29. Prentice Hall, 2003.

[5] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *Networking, IEEE/ACM Transactions on*, 1:397–413, 1993.

[6] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *Communications of the ACM*, 55(1):57–65, 2012.

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

[8] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*, pages 148–155. IEEE, 2013.

[9] Koen De Schepper, Olga Bondarenko, Jyh Tsang, and Bob Briscoe. PI2: 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, 2016.

[10] Network Simulator 2. `http://www.isi.edu/nsnam/ns`, 1995.

[11] Network Simulator 3. `https://www.nsnam.org`, 2011.

[12] S. Floyd, R. Gummadi, S. Shenker, et al. *Adaptive RED: An algorithm for increasing the robustness of RED's active queue management*, 2001.

[13] Toke Hoeiland-Joergensen, Paul McKenney, Dave Taht, Jim Gettys, and Eric Dumazet. Flowqueue-codel. *IETF ID draft-ietf-aqm-fq-codel-00*, 2014.

[14] F. Baker G. White B. Ver Steeg M. Prabhu C. Piglione R. Pan, P. Natarajan and V. Subramanian. PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem. internet draft draft-ietf-aqm-pie-10, internet engineering task force. Sept. 2016.

[15] K De Schepper, B Briscoe, O Bondarenko, and IJ Tsang. Dualq coupled aqm for low latency, low loss and scalable throughput. Technical report, Internet Draft draft-briscoe-aqm-dualq-coupled-00, IETF, 2015.

[16] Tcp prague. `https://www.ietf.org/mailman/listinfo/tcpprague`, 2017.

[17] Sumitha Bhandarkar, AL Reddy, Yueping Zhang, and Dimitri Loguinov. Emulating aqm from end hosts. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 349–360. ACM, 2007.

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

[19] J. D. C. Little and S. C. Graves. Little's law. In *Building intuition*, pages 81–100. Springer, 2008.

[20] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM computer communication review*, volume 40, pages 63–74. ACM, 2010.

[21] K. Ramakrishnan, S. Floyd, D. Black, et al. The addition of Explicit Congestion Notification (ECN) to IP, 2001.

[22] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[23] Rohit P Tahiliani and Hitesh Tewari. Implementation of pi 2 queuing discipline for classic tcp traffic in ns-3.