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

Rohit P. Tahiliani, Hitesh Tewari
School of Computer Science & Statistics,
Trinity College Dublin
Email: tahiliar@tcd.ie, htewari@cs.tcd.ie

***Abstract*—This paper presents the implementation and validation of PI^2 Active Queue Management (AQM) algorithm in ns-3. PI^2 provides an alternate design and implementation to Proportional Integral controller Enhanced (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. In this paper, we implement a model for PI^2 in ns-3 and verify 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.**

***Index Terms*—Queuing Disciplines, Bufferbloat, PI^2**

I. INTRODUCTION

The proliferation of delay-sensitive applications on the Internet has given rise to new challenges for queue management. On the other hand, reduced memory costs and the need to accommodate large bursts have encouraged the vendors to increase the router buffer sizes. Although this solves the issue of packet loss and improves TCP throughput, it leads to increased queuing latency. Management of large buffers is indispensable because the unmanaged buffers result in a number of problems such as bufferbloat [1], lock-out [2] and global synchronization [3].

AQM algorithms are being re-investigated with a focus on controlling the queuing latency. Algorithms such as Controlled Delay (CoDel) [4] and Proportional Integral controller Enhanced (PIE) [5] have been designed to minimize queue delay and retain high link utilization. Recently, a new AQM algorithm called PI^2 [6] has been proposed which offers same responsiveness and stability as PIE, but has a simpler design and implementation.

Our contributions in this paper are twofold. First, we propose a new model for PI^2 algorithm in ns-3 [7] along with its design and implementation. Our proposed model is based on the Linux code of the authors of PI^2.[1] To the best of our knowledge, there does not exist a PI^2 implementation in popular network simulators like ns-2 [8] and ns-3. We believe that our implementation of PI^2 in ns-3 would provide an additional platform to the community to verify its effectiveness and usefulness for future AQM architectures, such as DualQ [9]. Second, we 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.

The rest of the paper is organized as follows: Section II provides a brief background on PIE, PI^2 and the differences between both. Section III details the design and implementation of PI^2 model. Section IV presents the validation of PI^2 model in ns-3. Section V summarizes and concludes the paper.

II. BACKGROUND

A. PIE

PIE is now an Experimental RFC (RFC 8033) and also a recommended AQM algorithm for DOCSIS cable modems (RFC 8034). It uses the Proportional Integral (PI) [10] controller to keep the queuing delay to a specified target value by updating the drop probability at regular intervals. Following are the major components of PIE:

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

Drop Probability Calculation: This happens at every *tupdate* interval. It is calculated as [11]:
$p = \alpha * (qdelay - target) + \beta * (qdelay - qdelay\_old)$
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

Queuing delay estimate: PIE uses Little's law [12] to estimate the current queuing delay.

Burst Tolerance: PIE allows the short term packet bursts to pass through for a specified interval.

B. 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)

---

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

ISBN 978-3-901882-94-4

and Scalable (e.g., Data Center TCP [13]) congestion controls. In this paper, 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) [14] which is not yet completely supported in the main line of ns-3. The components discussed in Section II.A apply even to PI^2 with minor changes.

C. Differences between PI^2 and PIE

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 [6].

Burst allowance: PI^2 disables the burst allowance as to avoid an impact on the Data Center TCP fairness [6].

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 [6].

III. PI^2 MODEL IN NS-3

This section provides insights into the implementation of PI^2 algorithm in ns-3. PI^2 algorithm has been implemented in a new class called `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) [3], 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 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 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 probability at regular intervals *(tupdate)*. `DoDequeue` is invoked on packet departure and estimates the average drain rate.

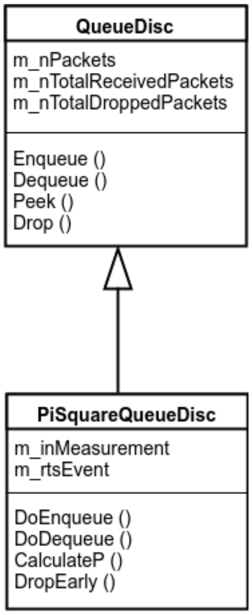


Fig. 1: Class Diagram for PI^2 model in ns-3.

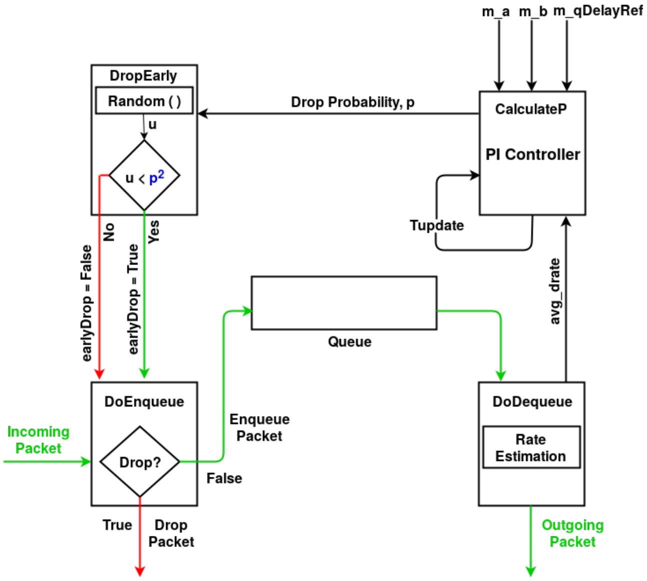


Fig. 2: Interactions among components of PI^2 in ns-3.

A. 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.

B. 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 I 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 I: 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

C. 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 II 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 II: 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*.

IV. 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.[2]

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: (i) varying the traffic and (ii) comparing the CDF of queue delay. These scenarios are in line with the ones used by the authors of PI^2 [6]. However, due to the unavailability of CUBIC [15] and ECN models in ns-3, we have used TCP NewReno [16] 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 III presents the details of simulation setup.

TABLE III: 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

Scenario 1: 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 are set as shown in Table III. Figure 3 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.

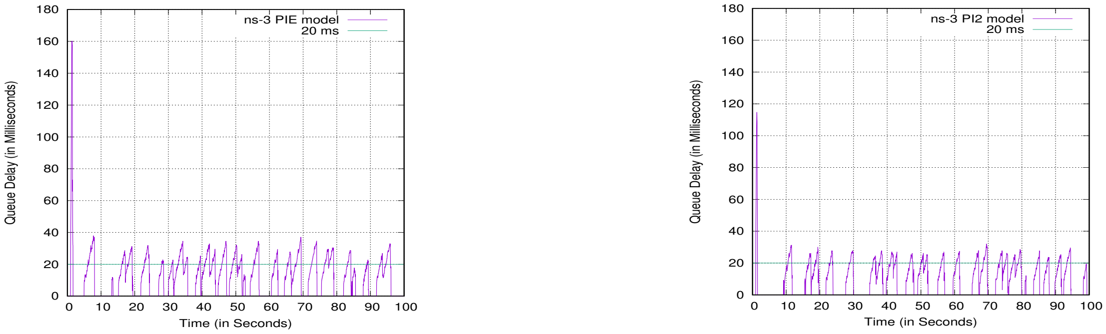
Figure 4 shows the instantaneous throughput. Initially the throughput degrades due to packets being dropped by PI^2 and PIE in an effort to control the queuing delay and maintain it around the desired target delay. It can be noted that throughput degradation with PI^2 is slightly more because of its tighter control on the queue delay. Nevertheless, both algorithms yield similar performance for the rest of the simulation.

Scenario 2: Heavy TCP Traffic

This scenario is same as Scenario 1, but configures 50 TCP flows instead of 5 TCP flows. Figure 5 shows the variations

---

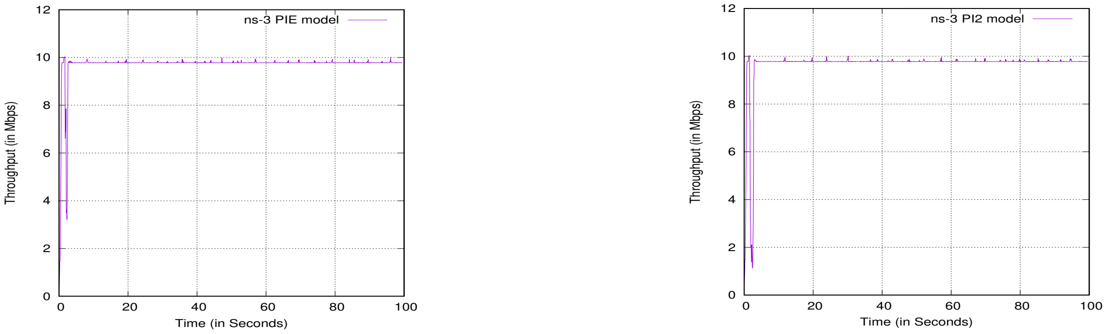
[2] https://codereview.appspot.com/314290043/



(a) ns-3 PIE

(b) ns-3 PI^2

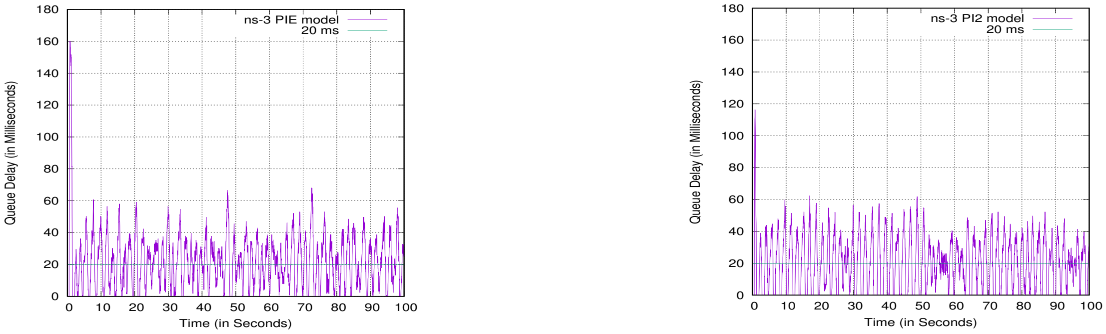
Fig. 3: Queue delay with light TCP traffic.



(a) ns-3 PIE

(b) ns-3 PI^2

Fig. 4: Link throughput with light TCP traffic.



(a) ns-3 PIE

(b) ns-3 PI^2

Fig. 5: Queue delay with heavy TCP traffic.

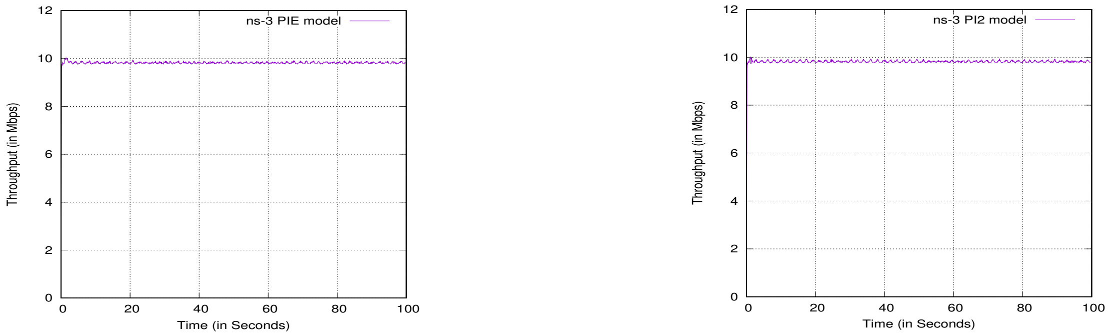
in queuing delay over time. Similar to the previous scenario, we can observe that PI^2, like PIE, quickly brings down the queuing delay and keeps it around the desired target value despite heavy TCP traffic. The results are similar to those obtained for Scenario 1. Although the amount of burst in this 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.

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 III.

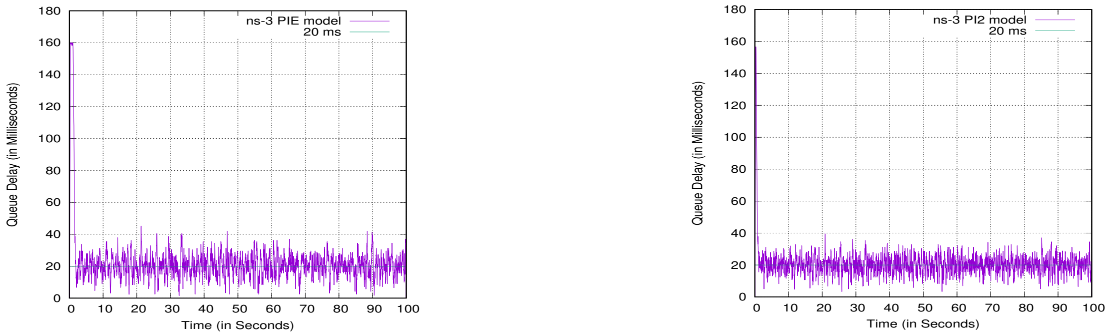
We observe that the results obtained for PI^2 and PIE are



(a) ns-3 PIE

(b) ns-3 PI^2

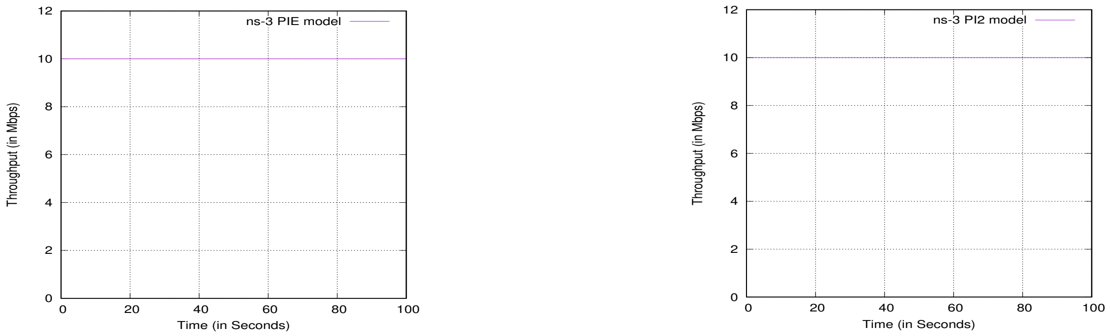
Fig. 6: Link throughput with heavy TCP traffic.



(a) ns-3 PIE

(b) ns-3 PI^2

Fig. 7: Queue delay with mix TCP and UDP traffic.



(a) ns-3 PIE

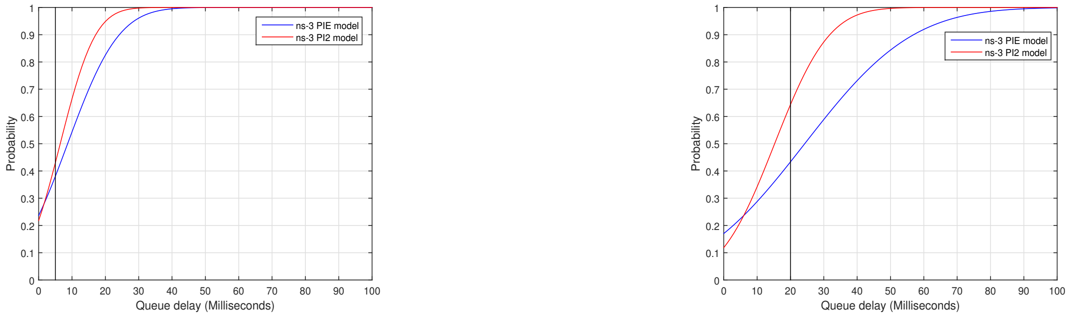
(b) ns-3 PI^2

Fig. 8: Link throughput with mix TCP and UDP traffic.

similar. Figure 7 shows that PI^2 and PIE control the queuing delay successfully. Moreover, in Figure 8 we can observe that the bottleneck bandwidth is completely utilized with both the algorithms.

Scenario 4: CDF of Queue 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 [6]. 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 III. Figure 9 and 10 show the CDF plots comparing the queuing delay of PI^2 and PIE. 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.



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

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

Fig. 9: CDF of queuing delay with 20 TCP flows.

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

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

Fig. 10: CDF of queuing delay with 5 TCP and 2 UDP flows.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we describe the implementation of PI^2 algorithm in ns-3 for Classic TCP flows. We present the design of our model and the interactions among different components of PI^2. Furthermore, we evaluate the effectiveness of our implementation by comparing the results obtained from it to those obtained from the PIE model of ns-3. We note that PI^2 with its simple design can deliver similar performance as PIE. Our implementation of PI^2 has been submitted for review. On the availability of ECN in main distribution of ns-3, we plan to extend PI^2 to work alongside ECN. Moreover, PI^2 model in ns-3 can be further extended to work for scalable congestion control algorithms like Data Center TCP after they are available in the main distribution.

REFERENCES

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

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

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] 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.

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