DUALPI2 - Low Latency, Low Loss and Scalable Throughput (L4S) AQM

Olga Albisser
Simula Research Laboratory

Koen De Schepper
Nokia Bell Labs

Bob Briscoe
Independent

Olivier Tilmans
Nokia Bell Labs

Henrik Steen
Simula Research Laboratory

Abstract

We implemented the DualPI2 AQM as a Linux qdisc and present the details of DUALPI2 implementation, explaining which problems it can solve.

Traditionally, classic loss-based congestion controls like Cubic or Reno need a relatively large queue to utilize the network efficiently and with reasonable loss levels. These large queues introduce unnecessary delay. Like in Data Centers, the problem can be partially solved by using scalable congestion controls that can use shallow immediate ECN marking thresholds instead of loss as congestion signal. But then, another problem is introduced - classic and scalable TCP congestion controls are not able to coexist without classic TCP starving itself. This starvation occurs due to the difference in how classic and scalable congestion controls respond to congestion signals.We explain how DUALPI2 solves this problem and can be used to mix Internet and Datacenter TCP traffic both on the Internet and in Data Centers, without compromising on neither the ultra-low latency performance of DCTCP, nor the Reno/Cubic throughput performance. DualPI2 uses a scheduler with only 2 queues, coupled for fairness, and a classifier that uses only IP header inspection, being transport protocol independent (supporting TCP, SCTP, QUIC,).

The largest benefit of our solution is the ability to deploy congestion controls like DCTCP on the public Internet and allow a mix of DCTCP and Internet congestion controls in the Data Center. The standardization of the DualPI2 AQM and the way the Low Latency, Low Loss and Scalable traffic (L4S) is identified, are being finalized in the IETF.

1 Introduction

Low or near-zero latency bcecomes more and more important for many, if not most applications. Almost any web or interactive application, such as voice and video clients, remote desktop, online gaming, finance apps, or similar would sometimes suffer from even a relatively small increase in latency, which would result in reduced performance and impaired user experience.

Latency is a complex problem that needs to be addressed with respect to various aspects [4] and at different levels of data delivery - both at end systems and in the network.

In this paper, we will focus on reducing the queuing delay in the network, with certain requirements imposed on the end systems. Even state-of-the art Active Queue Management (AQM) [13, 7] are only able to reduce the latency to nearly the same order as typical base round-trip time delay, being constrained by bottlenecks with low flow multiplexing and the necessity to buffer a round trip flight of data to prevent underutilization.

The DualPI2 AQM presented in this paper achieves close to zero qeueing delay for all applications, and not just a fraction of the link's traffic, as offered by a differentiated service (Diffserv) class such as EF [5]. The service we introduce accomodates not only applications that require low latency, but also capacity-seeking applications that require both low latency and high throughput. Also, it eliminates congestion loss introduced by using drop as a signal. We call this service L4S - Low Latency, Low Loss, Scalable throughput.

To benefit from L4S service, senders are required to use 'Scalable' congestion controls, as discussed in § 2. While DCTCP is an example of a scalable congestion control, L4S service is not intented for DCTCP in particular, but rather for a rangle of such controls. Besides, DCTCP needs certain safety and perforance adjustments before it can be used effectively in production, while a group of DCTCP developers has informally agreed on 'TCP Prague' requirements to both replace a confusing name and summarize the enhancements that need to be introduced.

However, in this paper we use DCTCP 'as is', solely as an example of a scalable control, and focus on network related changes only.

Using scalable congestion control together with 'Classic' congestion control (also discussed in § 2) introduces the coexistence problem. With state-of-the-art AQMs, the two congestion controls, for example, Cubic and DCTCP, are not

deployed together, because due to different response to congestion signals, 'Classic' congestion control senders would starve themselves. DualPI2 AQM solves this coexistence problem by using two queues, which both isolate the two types of traffic and use coupling to appear as a single resource pool for best utilization and emit congestion signals at the rate necessary for each traffic type, explained in more detail in § 3.

In § 4, we discuss the deployment cases and configuration details, while a brief evaluation of DualPI2 performance, compared to state-of-the-art AQMs is presented in § 5.

2 Motivation and Background

Scalable Congestion Control The AQM mechanism we present guarantees low latency for L4S sources, while their end systems are required to use a scalable congestion controller. A congestion controller is defined as 'Scalable' if the rate of the congestion signals per round trip scales together with bandwidth-delay product (BDP or window) changes.

Classic version of TCP congestion controller, for example, Cubic or Reno, does not satisfy that requirement. Although Cubic does scale better than Reno, it's still not fully scalable, meaning that it's recovery time after experiencing drops and reducing the rate is too large.

On the other hand, DCTCP, being defined as scalable [3], uses fixed recovery time (half a round trip), regardless of the change in BDP. Even though the dynamic behaviour of DCTCP is not scalable due to is unscalable window update algorithm [10], these problems can be solved without further changes in the network.

We require Scalable congestion control to be used by L4S sources, because it both maintains rate control and keeps the link utilized even if BDP changes.

ECN Explicit Congestion Notification We require L4S sources to use ECN[14] mostly because ECN is purely a signal and does not introduce an impairment, while dropping is both signal and an impairment in the form of packet loss. If dropping would be used by sources that use scalable control, it would introduce a very large impairment, especially at high load, due to more agressive control and the need for stronger signal. Besides, ECN can be emitted immediately, in contrast to drop, where drop-based AQMs hold back from introducing loss in case it's only a sub-RTT burst. Another advantage of using ECN is no need to add smoothing delay, which is needed with drop because without RTT knowledge for each flow, the network has to smooth over worst-case RTT. Scalable senders can smooth the network signals themselves, or skip the smoothing in slow start [2] .

Another huge benefit of ECN is the obvious latency benefit of near-zero congestion loss, removing retransmission and time-out delays, which is very important for short flows [15].

3 Solution design

The solution design will be decribed in three steps. First, we will explain the overtall structure (§ 3.1). In the second step, we will describe the details of each key aspect of the solution:

- coexistence between L4S and Classic flows § 3.2
- isolation of L4S service from Classic § 3.3
- overload handling § 3.4

Finally, in the third step, we will describe the Linux qdisc implementation in § 3.5.

3.1 Solution Structure

The main aspect of our solution structure is using two queues for two types of traffic with opposing requirements for delay. L4S traffic needs ultra-low latency, which cannot be achieved with large queue, while Classic traffic does require a queue of certain size to keep the link utilized. Obviously, we can't satisfy both requirements in a single queue.

To determine which queue each packet should be classified to, we check the 2-bit ECN field in packet IP header. Both types of traffic can use ECN. Classic sources use either 'ECT(0)' to indicate that they supported ECN or 'Not-ECT' otherwise, and in both cases packets are classified into Classic queue. L4S sources are required to use 'ECT(1)' with default configuration ('l4s_ecn' and 'l4s_dualq' parameters) to get their traffic classified into L4S queue. However, 'ECT(1)' is an experimental ECN codepoint and is being redefined for L4S. The current version of DCTCP still uses 'ECT(0)', therefore, we made it possible to configure our AQM with DCTCP compatibility ('dc_ecn' and 'dc_dualq' parameters), where 'ECT(0)' packets are classified into L4S queue, and only 'ECT(0)' packets go into the classic queue.

An L4S source using scalable congestion control reaches very low latency by enabling the network to signal congestion frequently, using ECN marking. However, such frequent signaling would introduce a problem for a Classic source and push it to starving itself. To solve this problem, we introduce a coupling mechanism between the two queues, to ensure the coexistence possibility for both traffic types. Such coexistence is achieved by maintaning TCP-fairness - rough steady state rate balance per RTT [9]. In our solution, we couple the congestion signals of the two queues, introducing a stronger signal in L4S queue, and weaker in the classic one, similarly to the single-queue coupled AQM in [6].

Congestion signals are also used to solve the problem of how often to schedule each queue. Our solution allows the end systems of each traffic type to 'schedule' themselves, based on the congestion signals frequency received from the network. However, since L4S traffic controls its own delay

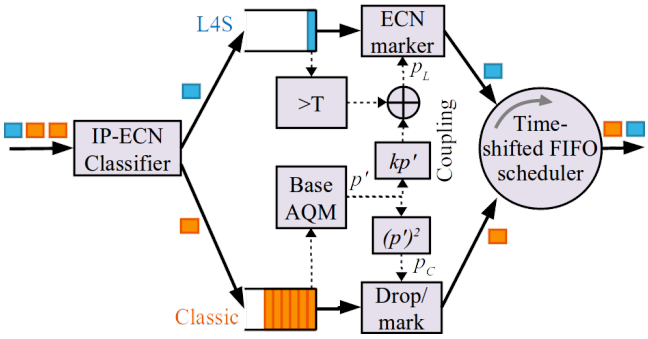

Figure 1: Dual Queue Coupled AQM

very well, it gets priority in cases when there is a 'disagreement'. Still, this does not cause the Classic traffic to starve itself, as the L4S priority only applies when the delay difference between the two queues does not exceed the threshhold.

The structure overview of the whole DualQ Coupled AQM is shown in Figure 1. with the classifier and scheduler as the first and last stages, and own native AQM for each queue in the middle.

3.2 Coupled AQM for Window Balance

The goal of coupling the two queues is achieving window fairness for Classic (C) and Scalable (L4S) congestion controls, to make their coexistence possible. To reach this goal, we need to determine the signal (marking or dropping) intensity for each control that would maintain window balance between them. First, we derive the formula to characterize the steady state window W for each control as a function of signal (loss or ECN-marking probability p). Second, like [6], we set the windows for L4S and C to be equal and formulate the relationship between the respective congestion signals.

For C, we use Reno, not because it's commonly used but as a worst case scanario, applying the simplified equation from [11]. For L4S, we use DCTCP, but we do not apply the step marking equation from DCTCP paper [1]; instead, we use probabilistic marking DCTCP equation, as derived in Appendix A of [6]. For balance between the windows ($W_{\text{reno}} = W_{\text{dc}}$, we substitute the equation for each window resulting in 1. Then we rearrange to derive the coupling relationship in 2:

$$\sqrt{\frac{3}{2p_{\text{reno}}}} = \frac{2}{p_{\text{dc}}} \quad (1) \qquad p_C = \left(\frac{p_L}{k}\right)^2, \quad (2)$$

where coupling factor $k = 2\sqrt{2/3} = 1.64$ for Reno. The formula in (2 applies both for Reno and TCP Cubic in Reno mode, except for TCP Cubic in Reno mode, the coupling factor becomes $k = 2/1.68 = 1.19$. We round it up to $k = 2$ to avoid floating point division in the kernel, which proves to be sufficient, as shown in the results of our experimets in § 5.

In the AQM, the coupling is implemented in two stages. First, base probability p' is calculated, which is then transformed for each queue, according to the coupling relation:

- L4S: $p_L = k * p'$
- Classic: $p_C = (p')^2$

3.3 Dual Queue for Low Latency

Each of the two queues needs its own native AQM, as in many cases, one of the queues will be empty. The low queuing delay in L4S queue is achieved with a small marking threshold (T), defined in time units with a floor of two packets.

When there is traffic in both queues, an L4S packet can be marked either by its native AQM or by the coupled AQM if T is exceeded. However, the coupling ensures that L4S traffic reaches T only when being bursty or if there is not enough traffic in the Classic queue.

We use a time-shifted FIFO scheduler [12] to decide between the head packets of the two queues, selecting the packet that has waited in the queue longest, after subtracting a constant timeshift to prioritize L4S packets. To protect Classic traffic from unresponsive L4S flow, we no longer give L4S packets such priority if the extra delay of the leading Classic packet exceeds the timeshift.

3.4 Overload Handling

Since we use a priority scheduler, we need to make sure that in overload conditions, we do not harm Classic traffic more we would when using a single queue.

The AQM subtracts the unresponsive traffic from the total capacity, allowing the responsive flows to share the remaining capacity. This applies to both types of traffic - L4S and Classic. Classic drop probability is used to handle the unresponsive traffic after marking probability reaches the maximum. At the same time, responsive flows continue to get ECN marking from the native AQM when the threshold is reached, preserving low delay.

3.5 Linux qdisc Implementation

Algorithm 1 Enqueue for Dual Queue Coupled AQM

```
1: STAMP(pkt)                                   ▷ Attach arrival time to packet
2: if LQ.LEN() + CQ.LEN() >L then
3:     DROP(pkt)                                ▷ Drop packet if Q is full
4: else
5:     if LSB(ECN(pkt))==0 then                 ▷ Not ECT or ECT(0)
6:         CQ.ENQUEUE(pkt)                      ▷ Classic
7:     else                                     ▷ ECT(1) or CE
8:         LQ.ENQUEUE(pkt)                      ▷ L4S
```

Algorithms 1 & 2 give a simplified summary of DualPI2 enqueue and dequeue implementations as pseudocode, omitting overload and saturation logic for clarity. The full code is available in a open-sourced Github repository.[1] Packets

Algorithm 2 Dequeue for Dual Queue Coupled AQM

```
1:  while LQ.LEN() + CQ.LEN() >0 do
2:      if LQ.TIME() + D ≥ CQ.TIME() then
3:          LQ.DEQUEUE(pkt)                                  ▷ L4S
4:          if (LQ.TIME() > T) ∨ (p_L > RAND()) then
5:              MARK(pkt)
6:      else
7:          CQ.DEQUEUE(pkt)                                  ▷ Classic
8:          if p_C > RAND() then
9:              if ECN(pkt)==0 then                          ▷ Not ECT
10:                 DROP(pkt)                                ▷ Squared drop
11:                 continue                                 ▷ Redo loop
12:             else                                         ▷ ECT(0)
13:                 MARK(pkt)                                ▷ Squared mark
14:     RETURN(pkt)                                          ▷ return the packet, stop here
```

are time-stamped and classified in enqueue. On dequeue, the time-shifted FIFO scheduler is implemented, which takes the packet that waited the longest. If an L4S packet is scheduled, line 4 marks the packet when the L4S threshold is exceeded or when the packet is randomly decided to be marked according to the probability p_L. When a Classic packet is scheduled, line 8 determines whether to signal congestion with probability p_C. Then line 9 checks if the Classic packet supports ECN and if so, uses ECN, otherwise, the packet is dropped.

The internal base signalling probability (p') is output by the core PI Algorithm (3) which only needs occasional execution [8].

Algorithm 3 PI core: Every T_{update} p is updated

```
1: curq = CQ.TIME()
2: p' = p' + α*(curq − TARGET) + β*(curq − prevq)
3: p_L = k * p'
4: p_C = (p')^2
5: prevq = curq
```

Probability calculation The first step is calculating the internal probability p'. This calculation is based on queue delay in each queue - L4S and classic.

Then, we calculate *delta*, which includes a change of queue delay comparing to the target delay, and a change comparing to previous queing time. The integral gain factor α is used to scale the change in queing time and restore any persistent standing queue to the user specified target delay (passed as *target* parameter), while proportional gain factor β is used to scale the change in queuing time comparing to previous measurement. Both changes are added to *delta* as follows:

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

We add delta to base probability, and check for overflow and underflow by comparing the probability to its previous value. If $delta > 0$ and current probability is smaller than its previous value, there was an overflow, so we set the current value to a maximum. If *delta* is zero or negative, there was an underflow, so we set the current value to zero.

As a next step, we check if switchover to drop is configured, defined by l_drop parameter. The l_drop parameter sets the maximum probability above which classic drop is applied to all traffic in both queues. Since we use a coupling factor between L4S and Classic queues, which we refer to as k, we need to align maximum drop probability 100% L4S marking in case l_drop is disabled (set to zero). To do this, we set the probability value to MAX_PROB/k, where MAX_PROB is maximum probability represented by the largest 32 bit integer.

All the steps above are executed once in an interval of time defined by $tupdate$ parameter, meaning the probability is updated every $tupdate$ ms. For different queues, the probability is tranformed according to the coupling relationship described in § 3.2.

This probability is used each time a packet is enqueued or dequeued, depending on what is configured. At dequeue, we peek at a packet from each queue and calculate the queue delay for each of them respectively. If any of the queues is empty, we dequeue a packet from the queue that is not empty and do not apply any dropping or marking. If both queues have packets, and biased L4S queue delay is greater or equal than classic queue delay, we dequeue an L4S packet, and if otherwise, we dequeue a classic queue packet. The biased queue delay is calculated by adding time-shift to the L4S queuing time: $qdelay_lb = tshift + qdelay_l << tspeed$
Time-shift is passed to the scheduler as $tshift$ parameter in time units, while $tspeed$ represents L4S FIFO time speed in bit shifts and is used to give a bias to L4S delay by scaling it.

4 Deployment

DualPI2 can be configured for different deployment scenarios. The README document in the Github repository lists the parameters that can be passed to the AQM to set the desired configuration and their default values[2]. All parameters are optional, and default values will be used is no parameters are specified.

Some of the parameters are worth paying additinal attention to, since they might need to be changed, depending on the deployment scenario. The first deployment scenario we target is adding DualPI2 at a path bottleneck on the Internet. In that case, using default parameters will classify ECT(1) and CE packets into L4S queue, while the rest of the traffic, ECT(0) and not ECT packets, will go through the Classic queue. In this scenario, scalable congestion control is expected to use ECT(1), while classic control can use either not ECT of ECT(0).

Another deployment scenario that can be relavant is Data centers where it is not possible to change everything at the same time, for example, in cases when there is no single system administrator. In that case, DualPI2 would make incre-

mental deployment possible, allowing scalable and classic congestion controls to co-exist until full deployment is completed.

For cases when DCTCP is used, DualPI2 has to be configured with ‘dc_dualq‘ and ‘dc_ecn‘ parameters, to enable L4S service for all ECT packets. For scalable congestion controls that only use ECT(1), default parameters (‘l4s_dualq‘ and ‘l4s_ecn‘) should be used. Scalable marking will then only be applied to ECT(1) packets, while ECT(0) traffic will get classic marking. Refer to section § 3.2 for explanation of the difference between scalable and classic marking.

DualPI2 can also be deployed with a single queue instead of two coupled queues, then it will behave like similar single queue AQMs (PIE or similar). Such scenario is not our main goal, but can be used at certain transitioning stages, when only a single congestion control is used, although, the same performance will be achieved when DualPI2 is configured with two queues (only one of them will be used).

The ‘alpha‘ and ‘beta‘ parameters are based on the stability analysis (Appendix A of [6]) and are used to convert changes in queueing delay into changes in mark or drop probability. The default values are proven to be sufficient for link speeds of up to 200 Mbit/sec and RTTs of up to 100ms, but for different scenarios, further stability analysis is required to choose the right values. The ‘tupdate‘ parameter represents probability calculation timer frequency, defining how often this probability is updated; its value is proportional to ‘alpha‘ and ‘beta‘, so ‘tupdate‘ is changed, ‘alpha‘ and ‘beta‘ should be adjusted accordingly, to ensure that the change in frequency results in the same response with finer or larger steps, instead of more (or less) aggressive congestion signalling. The ‘k‘ parameter is a coupling factor with default value that is suitable for congestion controls mentioned in § 3.2. Changing its value will result in smaller or larger difference in signal intensity issued for each type of traffic.

The ‘l_thresh‘ parameter defines queue size (in units of time) at which the L4S packets get marked. It can be set in either time units or packets by switching between ‘et_packets‘ and ‘et_time‘ parameters, but we recommend to use ‘et_time‘ as it gives a more accurate estimation of the queue delay. The default ‘l_thresh‘ value (1 ms) is small to ensure low delay, but it can be beneficial to increase it for bottlenecks with smaller link speed, due to higher packet serialization times. For instance, on a 4 Mbps link, the serialization time for a single packet is 3ms (1ms on 12 Mbps link), and the default ‘l_thresh‘ value of 1ms would result in all L4S packets being marked. It is therefore recommended to set ‘l_thresh‘ to an equivalent of at least 2 packets serialization time (6ms for 4 Mbps link, 3ms for 12Mbps link).

The target queuing delay for the Classic queue is set using ‘target‘ parameter. . The ‘t_shift‘ paramter represents scheduler bias in time units. As mentioned in § 3.2, we use time shifted bias to prioritize L4S packets. The ‘t_shift‘ time is subtracted from the delay of an L4S packet during scheduling, but if a Classic packet has larger delay than ‘t_shift‘, L4S packets are no longer given a priority. Therefore, changing ‘t_shift‘ to a larger value would mean that larger delay is allowed in Classic queue, so ‘t_shift‘ should be proportional to ‘target‘. We recommend to set ‘t_shift‘ to ‘*target*‘ ∗ 2. Another parameter that gives bias to L4S traffic is ‘t_speed‘; its value represents the number of bits we will use to left-shift the L4S delay, as mentioned in § 3.5. This type of scaling is disabled by default as ‘t_shift‘ is initialized to zero, but using a larger ‘t_speed‘ value would require to re-evaluate ‘t_shift‘ value, as ‘t_speed‘ will increase the L4S delay bias exponentially with every extra bit. For example, when default value of ‘t_shift‘ (40 ms) and ‘t_speed‘ (0) are used and the delay of a given L4S packet is 2ms, L4S delay will get additional bias of 40ms. If we increase ‘t_speed‘ to 3, left-shifting L4S delay of 2ms by 3 will convert it to 16 ms, giving additional 14ms bias.

The Classic taildrop is limited to 100% probability divided by ‘k‘ by enabling ‘c_limit‘ parameter by default. This can be changed by passing ‘l_drop‘ paremeter, which represents maximum L4S probability where classic drop is applied to all traffic. Setting ‘l_drop‘ disables ‘c_limit‘ and vice versa. By default, packets are dropped on dequeue, this can be changed by switching between ‘drop_enqueue‘ and ‘drop_dequeue‘, which are rather self-descriptive. We recommend using ‘drop_dequeue‘ as it provides faster response, and we have observed better performance with drop on dequeue.

The maximum number of packets that can be enqueued is set in the ‘limit‘ parameter, but if a parent qdisc uses a larger limit, it will override the value of this parameter.

5 Evaluation

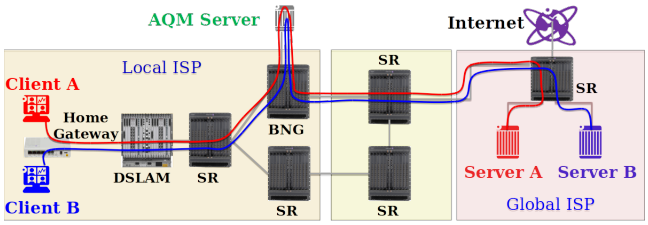

Figure 2: Testbed configuration

To evaluate the performance of DualPI2, we have conducted experiments in testbed consisting of a classical residential service delivery network composed of Residential Gateway, xDSL DSLAM (DSL Access Multiplexer), BNG (Broadband Network Gateway), Service Routers (SR) and application servers, as shown in Figure 2.

The main attribute of our testbed is a setup that includes two client and servers pairs, and AQM Server in the middle, allowing to have two different sources of traffic and a bottleneck where AQM is installed.

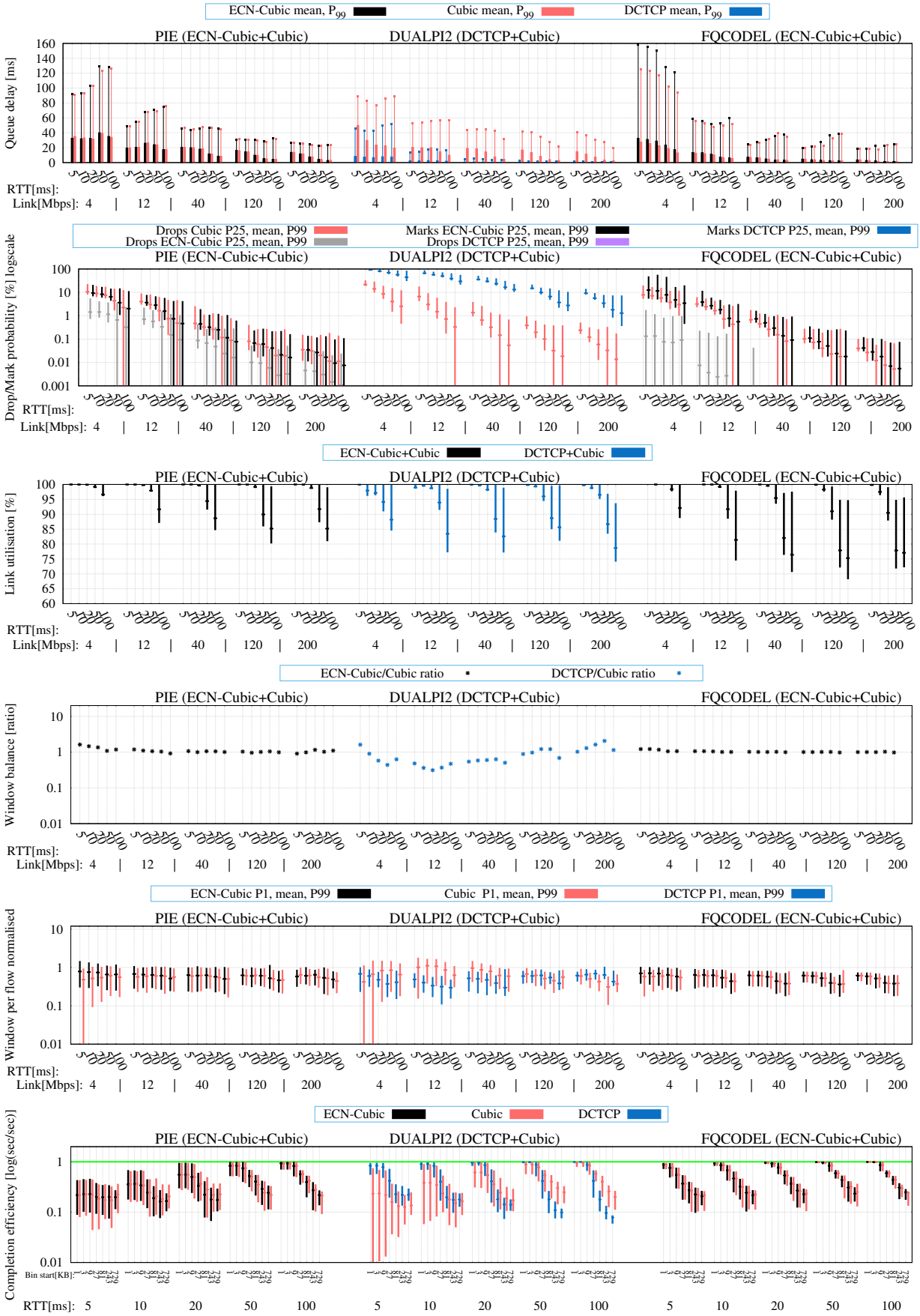

Figure 3: Heavy dynamic workload: 1 long flow and 300 short requests per second for each CC.

We have conducted a series of experiments with different scenarios, evaluating different aspects of AQM performance, but for brevity, we present the results of the main aspects of our evaluation, focusing on queue delay and window balance, as those are the main goals of DualPI2. Each experiment lasted 250 secs and was performed with different TCP congestion controls enabled on each client-server pair, repeated in different combinations of link speed (4,12,40,120,200 Mbps) and RTT (5,10,20,50,100). We used a single long-running flow and multiple short flows. To emulate short flows, we used an exponential arrival process with an average of 10 requested items per second for the 4 Mbps link capacity, scaled for the higher link speeds up to 500 requests for the 200 Mbps links. Every client request opened a new TCP connection, closed by the server after sending data with a size according to a Pareto distribution with $\alpha = 0.9$, with a size between 1 KB and 1 MB. The client logged the completion time and transfer size. Timing was started just before opening the TCP socket, and stopped after the client detected that the connection was closed by the server.

We performed the same set of experiments for 2 different AQMs - DualPI2, PIE and FQ Codel, with two different congestion control combinations - DCTCP + Cubic and ECN-Cubic + Cubic.

Queue delay measurements were done in each AQM by logging the sojourn time of the head of the queue at each dequeue. Window balance was calculated by dividing the window size of scalable (or ECN capable) congestion control by classic, where a value of 1 represents perfectly balanced window, while values greater than 1 indicate the ratio at which scalable congestion control (in DCTCP + Cubic comparison) exceeds the window of the classic control.

To better quantify the average and percentiles of the completion times, we used the Completion Efficiency representation, which was calculated by dividing actual completion time by theoretical completion time, where theoretical completion time represented the best achievable efficiency. We then binned the samples in log scale bins (base 3) and calculated the average, 1st and 99th percentiles. The green theoretical completion time is now at 1 (maximum efficiency).

As shown in Figure 3, DualPI2 achieves ultra-low queuing delay for L4S traffic, which is not possible to achieve with state-of-the-art AQMs, such as PIE and FQ CODEL. The delay is slightly higher for lower link speeds, as we allow a floor of 2 packets ECN marking threshold to avoid 100% marking. Window balance for DualPI2 is also maintained rather well, in contrast to PIE, where DCTCP has a clear window advantage.

6 Conclusion

In this paper, we have shown that DualPI2 achieves both ultra low latency and scalable for L4S traffic, without harming Classic traffic, making the coexistnce of scalable and classic congestion controls possible.

References

[1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). *Proc. ACM SIGCOMM'10, Computer Communication Review 40*, 4 (Oct. 2010), 63–74.

[2] ALIZADEH, M., JAVANMARD, A., AND PRABHAKAR, B. Analysis of DCTCP: Stability, Convergence, and Fairness. *Proc. ACM SIGMETRICS'11* (2011).

[3] BANSAL, D., AND BALAKRISHNAN, H. Binomial Congestion Control Algorithms. In *Proc. IEEE Conference on Computer Communications (Infocom'01)* (Apr. 2001), IEEE, pp. 631–640.

[4] BRISCOE, B., BRUNSTROM, A., PETLUND, A., HAYES, D., ROS, D., TSANG, I.-J., GJESSING, S., FAIRHURST, G., GRIWODZ, C., AND WELZL, M. Reducing Internet Latency: A Survey of Techniques and their Merits. *IEEE Communications Surveys & Tutorials 18*, 3 (Q3 2016), 2149–2196.

[5] DAVIE, B., ET AL. An Expedited Forwarding PHB (Per-Hop Behavior). Request for Comments 3246, Internet Engineering Task Force, Mar. 2002.

[6] DE SCHEPPER, K., BONDARENKO, O., TSANG, I.-J., AND BRISCOE, B. PI^2 : A Linearized AQM for both Classic and Scalable TCP. In *Proc. ACM CoNEXT 2016* (New York, NY, USA, Dec. 2016), ACM.

[7] HOEILAND-JOERGENSEN, T., MCKENNEY, P., TÄHT, D., GETTYS, J., AND DUMAZET, E. The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm. Internet Draft draft-ietf-aqm-fq-codel-06, Internet Engineering Task Force, Mar. 2016. (work in progress).

[8] HOLLOT, C. V., MISRA, V., TOWSLEY, D., AND GONG, W. Analysis and design of controllers for AQM routers supporting TCP flows. *IEEE Transactions on Automatic Control 47*, 6 (Jun 2002), 945–959.

[9] IRTEZA, S., AHMED, A., FARRUKH, S., MEMON, B., AND QAZI, I. On the Coexistence of Transport Protocols in Data Centers. In *Proc. IEEE Int'l Conf. on Communications (ICC 2014)* (June 2014), pp. 3203–3208.

[10] KELLY, T. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review 32*, 2 (Apr. 2003).

[11] MATHIS, M., SEMKE, J., MAHDAVI, J., AND OTT, T. The macroscopic behavior of the TCP Congestion Avoidance algorithm. *Computer Communication Review 27*, 3 (July 1997).

[12] MENTH, M., SCHMID, M., HEISS, H., AND REIM, T. MEDF - a simple scheduling algorithm for two real-time transport service classes with application in the UTRAN. In *Proc. IEEE Conference on Computer Communications (INFOCOM'03)* (Mar. 2003), vol. 2, pp. 1116–1122.

[13] PAN, R., PIGLIONE, P. N. C., PRABHU, M., SUBRAMANIAN, V., BAKER, F., AND VER STEEG, B. PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem. In *High Performance Switching and Routing (HPSR'13)* (2013), IEEE.

[14] RAMAKRISHNAN, K. K., FLOYD, S., AND BLACK, D. The Addition of Explicit Congestion Notification (ECN) to IP. Request for Comments 3168, RFC Editor, Sept. 2001.

[15] SALIM, J. H., AND AHMED, U. Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks. Request for Comments 2884, RFC Editor, July 2000.

Notes

[1]Open source at https://github.com/olgaalb/sch_dualpi2

[2]Documentation for DualPI2 at https://github.com/olgaalb/sch_dualpi2/blob/master/README.md