Congestion Control Aware Queuing

Maximilian Bachl, Joachim Fabini, Tanja Zseby
Technische Universität Wien
firstname.lastname@tuwien.ac.at

ABSTRACT

Recent model-based congestion control algorithms such as BBR use repeated measurements at the end-point to build a model of the network connection and use it to achieve optimal throughput with low queuing delay. Conversely, applying this model-based approach to Active Queue Management (AQM) has so far received less attention. We propose an AQM scheduler based on fair queuing, which adapts the buffer size depending on the needs of each flow without requiring active participation from the endpoint. We implement this scheduler for the Linux kernel and show that it interacts well with the most common congestion control algorithms and can significantly increase throughput compared to CoDel while avoiding overbuffering.

ACM Reference Format:
Maximilian Bachl, Joachim Fabini, Tanja Zseby. 2019. Congestion Control Aware Queuing. In *Proceedings of Workshop on Buffer Sizing*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the last decades various Active Queue Management (AQM) mechanisms have been proposed to minimize excessive *standing queues* in the Internet. One of the most influential recent efforts is CoDel [11] whose goal is that the queuing delay at the bottleneck link is at least once under 5 ms in a moving window of 100 ms.

While it is important to keep the queuing delay constrained, it is also necessary to ensure that overly aggressive flows cannot benefit by "stealing" less aggressive flows' bandwidth. Thus researchers and engineers have developed *Fair Queuing (FQ)* mechanisms [6, 12] to isolate different flows' queues so that for example a delay-sensitive live video call cannot be impaired by a concurrent bulk transfer which takes all the available bandwidth.

Recent approaches have tried to combine AQM with FQ. [13] demonstrate *fq_codel*, a queuing discipline (qdisc) that uses FQ and lets CoDel manage each queue. [8] expand upon this and create the *cake* qdisc that also adds features such as not only per-flow queuing but also per-host queuing for even increased fairness. Furthermore they also include bandwidth shaping into their solution and aim to create one qdisc that is easy to configure, can be easily deployed on home routers and offers all features in one solution.

While we do not want to make statements about the general performance of CoDel, we show that fq_codel and cake do not optimally use available bandwidth in common network configurations for common Congestion Control Algorithms (CCAs). This becomes especially prevalent for links with a high bandwidth or a large Round-Trip Time (RTT) but is already noticeable for common scenarios, such as a link with 100 Mbit/s and an RTT of 50 ms.

We show that the impaired performance is a result of keeping the queuing delay under 5 ms, which hinders CCAs such as Reno or Cubic from reaching maximum throughput. The main drawback is that these qdiscs aim to keep the delay under the threshold no matter the effect on throughput and do not take the Congestion Control (CC) of the flow into account.

As a remedy, we conceive an AQM mechanism that explicitly measures the behavior of the CC of a flow and dynamically changes the buffer so that
(1) link utilization is optimized and
(2) queuing delay is kept at the minimum that is required to achieve optimum throughput considering the CC.
Finally, we develop a prototype of Congestion Control Aware (cocoa) and show that it can achieve these objectives for the most common loss-based CCAs, Reno [9] and Cubic [7]. These CCAs operate by continuously increasing the number of bytes that are allowed to be in the network (congestion window) if no packet loss is experienced and by sharply decreasing this number if a packet is lost (multiplicative decrease). With Cubic being the default CC in all major OS, we especially emphasize our evaluation on improving its performance. Contrasting to the aforementioned CCAs, recently proposed BBR [3] does not continuously increase and then sharply decrease when packets are lost but instead uses periodic measurements to estimate the available bandwidth

Workshop on Buffer Sizing, December 2-3, 2019, Stanford, California, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as well as the minimal RTT and then tries to stay at this point of optimal bandwidth and minimal delay. BBR is thus considered to be *model-based.* We demonstrate that cocoa also behaves well in interaction with BBR v1.

2 CONCEPT

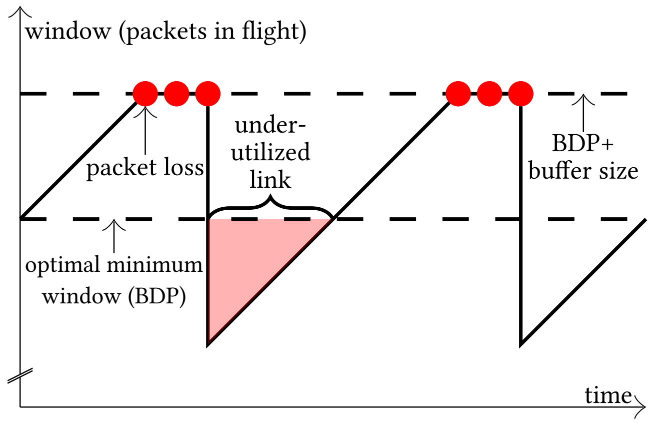

Figure 1: If the buffer is too small, loss-based CCAs will not be able to fully utilize the link since they send too few data following the multiplicative decrease that occurs after packet loss.

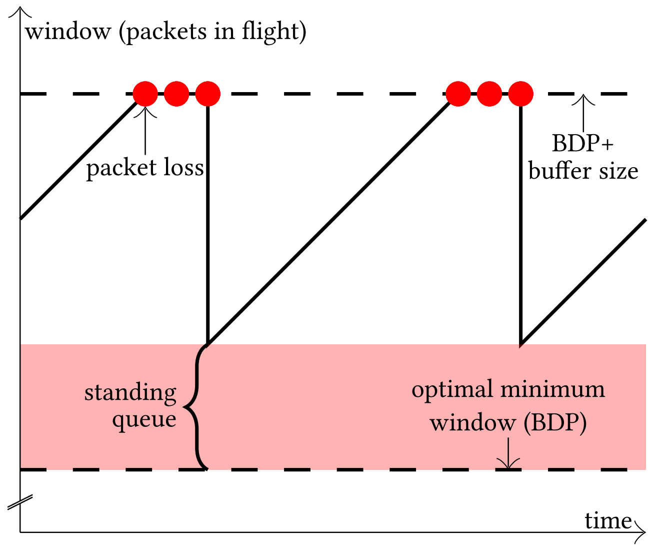

Figure 2: If the buffer is too large, loss-based CCAs will keep a unnecessary standing queue which is not necessary for achieving full link utilization.

Since the goal of our algorithm is to achieve optimal throughput irrespective of the CCA, we aim to avoid the scenario depicted in Figure 1: Here the buffer is too small, meaning that the CCAs never manages to achieve full utilization and periodically underutilizes the link, leading to, for example, a user waiting longer for a software update to finish or a game to download on their video game console.

The other case we want to avoid is a persisting standing queue, as depicted in Figure 2: In this case the buffer is too large meaning that optimal throughput is achieved but at the same time that an unnecessarily long standing queue is maintained. This oversized queue leads to RTT being larger than necessary and can result in unresponsive applications and reduced Quality of Experience. Furthermore, the buffer space has to be allocated in the bottleneck device and keeps the memory from serving a more useful purpose.

These considerations lead us to designing an algorithm that dynamically measures the CCA and adapts the buffer size to reach our goal of maximum throughput and minimal delay. The basic functionality of this algorithm would be to

(1) observe the CCA for a certain measurement period and then
(2) increase or decrease the buffer size depending on whether the scenario of Figure 1 or Figure 2 is true.

The challenge about this is to define the measurement period: As Figure 1 and Figure 2 show, we would like to use the measurement period that is the longest period between two packet losses. If this period is observed, it is possible to compute the standing queue as simply the minimum queue observed in this interval. For example, if the minimum queue is 3 packets it means, that there is a standing queue of 3 packets and the algorithm will reduce buffer size by 3 to eliminate the unnecessary queuing. Conversely, if the buffer is too small, the algorithm computes how long the link was underutilized and how many packets could have been transmitted during the idle period. The buffer size is then increased by this number of packets.

This is not so trivial because of the following consideration: If the measurement period is mistakenly not the longest interval shown in the figure, but the interval between two adjacent packet losses, the algorithm would assume an enormous standing queue and drastically reduce the buffer to be close to zero. This would then lead to severe underutilization afterwards.

The solution is thus to use a *Guard Interval (GI)*: The algorithm always waits for a specified minimum amount of time, picks the longest interval without packet loss in this guard interval and then applies its logic to reduce the standing queue on this *Longest Interval (LI).* We dynamically compute the GI as a multiple or a fraction (parameter of our algorithm) of the previous LI (Figure 3):

For example, we choose the multiplier to be $\frac{1}{2}$. Now, if the LI is in the previous GI was 100 ms, then the next GI is at least $\frac{1}{2}100\ ms$. During the GI we monitor all intervals and if the current interval is the longest one, it becomes the new

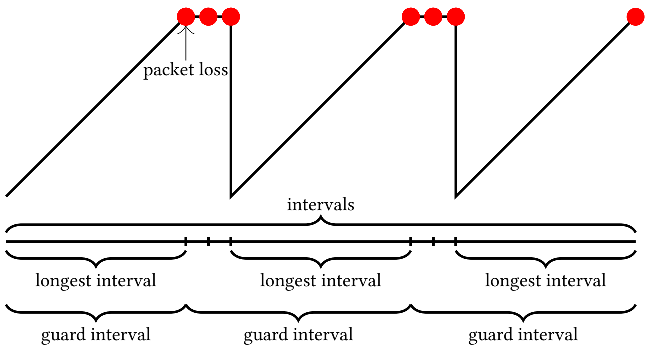

Figure 3: Illustration of the interval mechanism used by cocoa with a multiplier of $\frac{1}{2}$. Each *guard interval* (GI) contains one or more intervals of which the longest one is considered the *longest interval* (LI). The GI is always at least the length of the previous LI times the multiplier. A GI ends at the first packet loss after the previous LI times the multiplier.

LI. The GI ends when the first packet loss occurs after the end of the GI.

Almost all of the logic of our algorithm is executed when a new packet is received to be enqueued. The full algorithm is depicted as Algorithm 1.

Algorithm 1 Procedure that is executed when a new packet is received to be enqueued.

```
 1: function ENQUEUE(new_packet)
 2:     if queue is full then
 3:         if link was idle during this interval $\wedge$ the buffer wasn't already enlarged in this interval $\wedge$ this is not the first interval of this flow then
 4:             buffer_size ← buffer_size + (packets_transmitted_in_current_interval / time_active $\times$ time_idle)
 5:             add new_packet to queue
 6:         else if the buffer was enlarged in this interval $\vee$ this is the first interval of this flow then
 7:             start a new interval
 8:             start a new GI
 9:             drop new_packet
10:         else if current_time $\geq$ end_the_current_GI then
11:             if there was a standing queue in the LI of the GI then
12:                 buffer_size ← buffer_size - standing_queue
13:             end if
14:             start a new interval
15:             start a new GI
16:             drop new_packet
17:         else
18:             start a new interval
19:             drop new_packet
20:         end if
21:     else
22:         add new_packet to queue
23:     end if
24: end function
```

2.1 Choice of the multiplier

The purpose of the multiplier, which is used to multiply the LI of previous GI to define the new GI, is to prevent that the short gap between two subsequent packet losses is used as the interval to determine whether the buffer size has to be adapted. This means that the multiplier must be large enough so that it allows to skip over the packet losses that occur when the buffer is full. For Reno and Cubic, the loss-free period is very large compared to the period during which packet loss occurs. Specifically, the period with packet loss has the length of one RTT since Reno and Cubic reduce their sending rate as soon as packet loss occurs, thus ending the packet loss. This means that the multiplier can be very small in order to work for these CCAs. We conducted experiments with the multiplier being 0.5 and have never encountered the problem that the interval between adjacent packet losses is considered the longest interval and that the buffer is cut to virtually nothing because a standing queue is assumed. However, BBR v1 uses a completely different approach compared to Reno and Cubic to probe for bandwidth: It periodically increases its sending rate and then reduces it again to see if it can achieve more throughput. This happens with the following pattern: $[\frac{5}{4}, \frac{3}{4}, 1, 1, 1, 1, 1, 1]$. The sending rate is increased to probe for more bandwidth, then decreased to reduce the standing queue potentially formed and after that BBR is simply keeping the sending rate to match the estimated bandwidth. Each of these eight phases lasts one RTT. BBR uses a random cyclic permutation of this pattern with the only condition being that it cannot start with $\frac{3}{4}$. The following pattern could thus occur: $[1, 1, 1, 1, 1, 1, \frac{5}{4}, \frac{3}{4}, \frac{5}{4}, \frac{3}{4}, 1, 1, 1, 1, 1, 1]$. If the GI ends after the first 6 phases and also the LI was these six phase, with a factor of 0.5, the next GI would thus end at the end of phase 8. Then, the LI would be the 8th cycle. The next GI would then be of the duration of 0.5 phases and would then end in the middle of cycle 9. This would be erroneous because this is a probing phase during which packet loss occurs. The aforementioned problem of erroneously and significantly reducing the buffer could thus occur. This means that for BBR v1 the multiplier must be larger than 1 since it can happen that non-probing phases are followed by probing phases of the same length. From these considerations it follows that for our experiments we generally use a multiplier of 1.25. The behavior is different for BBR v2 which drastically changed the probing phase and now consists of a random 2-3 s phase of constant sending rate (cruising) followed by a quick probing phase followed by a decrease phase [4, 5]. We thus conjecture that a multiplier of 0.5 is sufficient for BBR v2. However, we could not finally confirm this since BBR v2 is still not completely finished.

2.2 Maximum buffer increase

Another potential problem is the following: If the available share of bandwidth suddenly increases drastically, our algorithm would sharply increase the buffer size. In this case it can happen that the buffer becomes much too large. We thus limit the maximum buffer increase to be 2 times the previous buffer size. This parameter is only relevant in the case of a sudden increase of available bandwidth or if a flow's throughput is application limited and not during stable state behavior.

2.3 Maximum GI

One more challenge is posed by the following consideration: If a delay-based CCA is in its stable state, no packet loss should occur. The LI could thus become very large (like tens of seconds) and also the next GI would be extremely large. This can be a problem if a delay-based CCA like Vegas [2] is used for a long flow and then after several seconds, the bandwidth drastically lowers. Then suddenly packet loss could occur but the GI would be so long that nothing would happen for a long time. To counter this, we add a parameter that specifies the maximum duration of the GI. We set this parameter to 1 s. This means that we also assume that no flows are handled by cocoa which have an RTT larger than 1 s. Another possibility for setting this parameter is to set it to a multiple of the RTT, for example 2, 4 or 8 times the most recent RTT. RTT could be measured using TCP timestamps [1] in case of TCP flows or the spin bit [10] for QUIC.

3 IMPLEMENTATION

We implement cocoa as an extension of the “fq” qdisc [6]. This means that when using cocoa also all features offered by fq are available. We add three configuration parameters to the qdisc: the multiplier (default 1.25), the maximum buffer increase (default 2.0) and the maximum GI duration (default 1.0 s). We make the source code of our implementation freely available to enable reproducibility and encourage experimentation: https://github.com/CN-TU/cocoa-qdisc.

4 EVALUATION

We evaluate cocoa using the `py-virtnet` (https://github.com/CN-TU/py-virtnet) toolkit to build a virtual network using Linux's network namespaces. We initialize the buffer size for cocoa to 100 packets per flow like in fq. The testbed consists of a sender and a receiver connected via a switch. We runs the qdisc being tested on the interface that connects the switch to the receiver. We introduce delay with `netem` and limit bandwidth with `htb`.

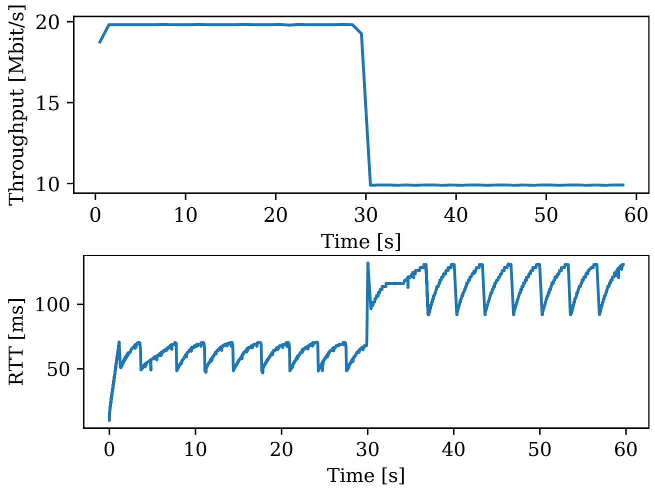

Figure 4: Throughput and delay of a Cubic flow with fq at the bottleneck. The initial bandwidth is 20 Mbit/s but it is halved after 30 s. The delay is 10 ms. It is clear that there is a standing queue and after the bandwidth halves, the minimum delay is 100 ms even though 10 ms would be possible.

First we evaluate if cocoa is able to maintain a small buffer like it is necessary in case of small Bandwidth Delay Products (BDPs). As can be seen in Figure 4 the regular fq qdisc maintains a standing queue and unnecessarily leads to a significant increase in RTT (100 ms minimum, when the real minimum is 10 ms). In contrast, Figure 5 maintains the full throughput while at the same time keeping the delay minimal and not keeping a standing queue. Also, when drastically changing the bandwidth by halving it (Figure 5) or doubling it (Figure 6) cocoa rapidly adapts to the new conditions and returns to the optimum state.

Next we compare cocoa against fq_codel. Figure 7 shows that fq_codel keeps the queuing delay under 5 ms. Also it shows that keeping the queuing delay that low is detrimental for achieving full throughput when using the Cubic CCA. With cocoa we achieve throughput that is more than 10% higher overall, while only slightly increasing the average delay. In addition to Cubic, we performed experiments with Reno. Here, over a 240 s flow of 100 Mbit/s with a delay of

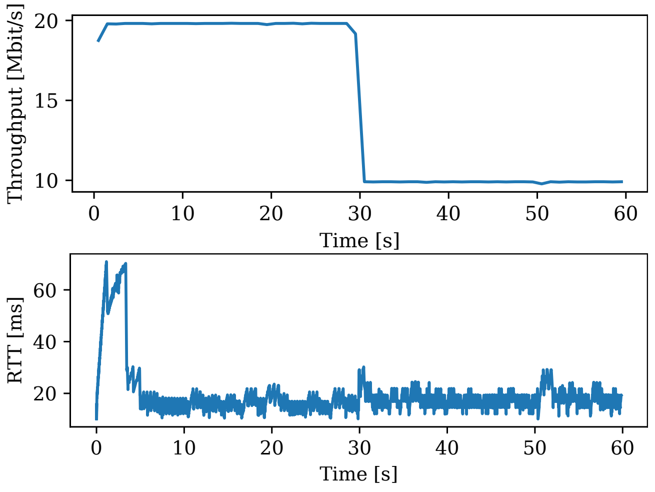

Figure 5: Throughput and delay of a Cubic flow with cocoa at the bottleneck. The initial bandwidth is 20 Mbit/s but we halve it after 30 s. The delay is 10 ms. cocoa keeps the queuing delay minimal while achieving the same throughput as with an oversized buffer.

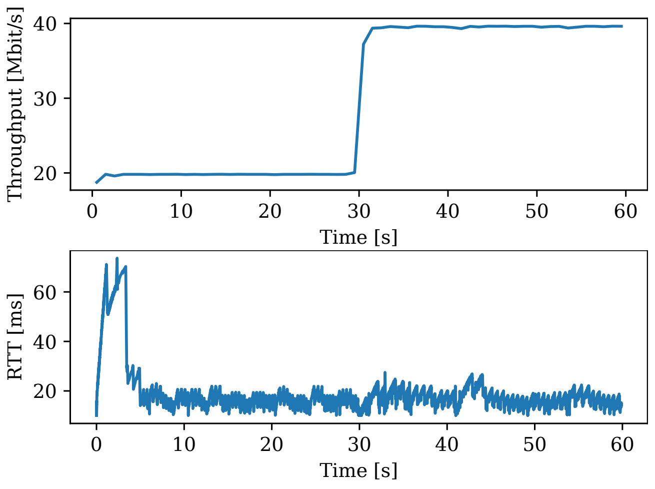

Figure 6: Throughput and delay of a Cubic flow with cocoa at the bottleneck. The initial bandwidth is 20 Mbit/s but we double after 30 s. The delay is 10 ms. Also a sudden large increase of bandwidth is handled well by cocoa.

50 ms, cocoa achieves more than 20% higher throughput than fq_codel. The average RTT for fq_codel is 51.34 ms while it is 96.78 ms for cocoa. Table 1 shows that cocoa reaches link utilizations close to 100% for both Cubic and Reno in large BDP scenarios. RTT can be higher than for other qdiscs if this is required to achieve optimum throughput for the CCA for the given link speed and base RTT.

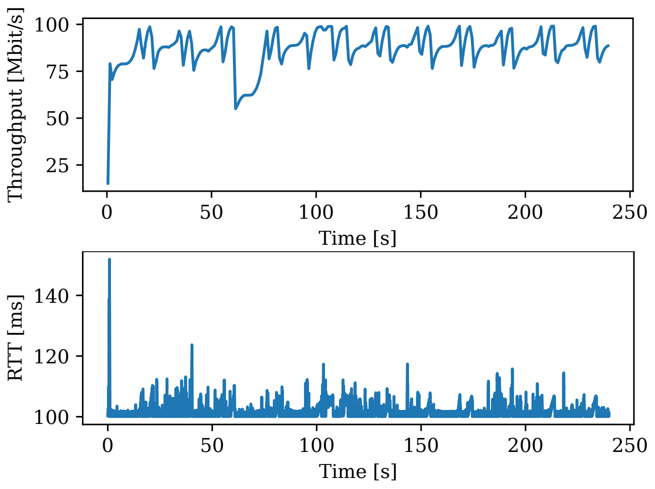

Figure 7: Throughput and delay of a Cubic flow with fq_codel at the bottleneck. The bandwidth is 100 Mbit/s. The delay is 100 ms. The average RTT is 101.61 ms. Total throughput is 2615 MB.

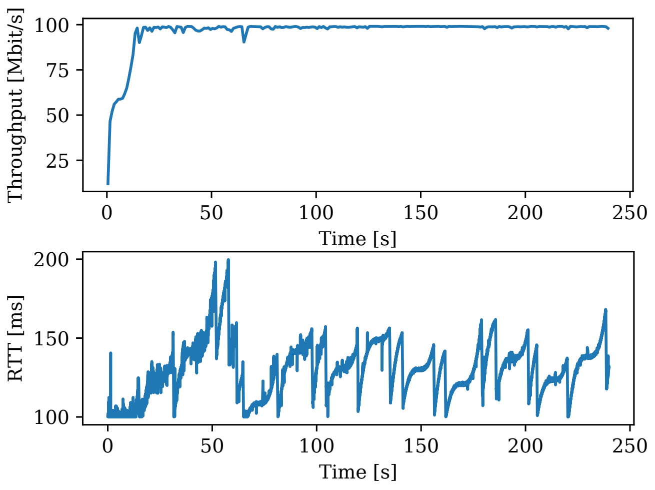

Figure 8: Throughput and delay of a Cubic flow with cocoa at the bottleneck. The bandwidth is 100 Mbit/s. The delay is 100 ms. The average RTT is 130.36 ms. Total throughput is 2893 MB.

Besides fq_codel possibly leading to lower throughput as we have shown, another problem is that for links with a small BDP fq_codel can keep a standing queue akin to fq since fq_codel only wants queuing delay to fall below 5 ms once every 100 ms. We also ran experiments for these low BDP scenarios: For a link speed of 20 Mbit/s and a base RTT of 1 ms for a Cubic flow both cocoa and fq_codel achieve 98% utilization. The average RTT is 6.86 ms for cocoa and 7.31 ms for fq_codel.

Table 1: Comparison of qdiscs for loss-based CCAs on a link of 100 Mbit/s with an RTT of 50 ms. Utilization in % of maximum capacity; RTT in milliseconds. For Cubic fq achieves good performance. This is because its default buffer size of 100 packets coincidentially works well for this specific link speed and RTT.

	fq		fq_codel		cocoa	
	Util.	RTT	Util.	RTT	Util.	RTT
Cubic	96.2	54	92.6	52	98.5	66.5
Reno	84.1	52.7	81.1	51.5	97.9	98.2

We also ran experiments with the cake qdisc. However, results were very similar to those of fq_codel, which is no surprise since cake is based on fq_codel.

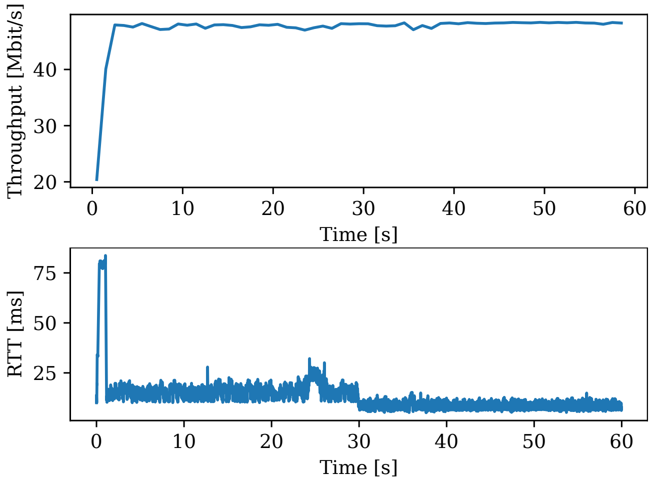

Figure 9: Throughput and delay of a BBR v1 flow with cocoa at the bottleneck. The bandwidth is 50 Mbit/s. The delay is 10 ms and we halve it after 30 s.

Also for BBR v1, throughput quickly reaches its maximum while no standing queue is forming (Figure 9).

5 DISCUSSION

Our goal was to design a qdisc which achieves maximum throughput while keeping the queuing delay as small as possible. The results of the experiments, which we perform with a prototype of cocoa, show that there are certain common scenarios in which current state-of-the-art qdiscs like fq, fq_codel and cake fall short of the optimum throughput by a significant margin while our approach succeeds in leading to full utilization of the link at the bottleneck (with a potential increase in delay). Furthermore, the aforementioned qdiscs also suffer from standing queues in scenarios with small BDP which we can also mitigate with our approach.

We see this work as a initial step towards flow-adaptive queuing at bottlenecks. An interesting further improvement could be to not statically initialize the buffer with a constant 100 packets but instead use experience from previous flows. Moreover, we think that a promising direction for future work might be to explore the use of reinforcement learning. Such an approach would fingerprint a flow and choose the buffer size accordingly, maximizing a chosen objective such as high throughput with minimal delay.

ACKNOWLEDGEMENTS

We thank Gernot Vormayr for providing us with `py-virtnet`, a toolkit for easily building virtual networks.

REFERENCES

[1] David Borman, Richard Scheffenegger, and Van Jacobson. 2014. TCP Extensions for High Performance. https://tools.ietf.org/html/rfc7323

[2] L.S. Brakmo and L.L. Peterson. 1995. TCP Vegas: end to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications* 13, 8 (Oct. 1995), 1465–1480. https://doi.org/10.1109/49.464716

[3] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53. http://queue.acm.org/detail.cfm?id=3022184

[4] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Ian Swett, Victor Vasiliev, Bin Wu, Matt Mathis, and Van Jacobson. 2019. BBR v2: A Model-based Congestion Control IETF 105 Update. (2019), 21.

[5] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Ian Swett, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, Matt Mathis, and Van Jacobson. 2019. BBR v2 A Model-based Congestion Control IETF 104. (2019), 36.

[6] Eric Dumazet. 2013. pkt_sched: fq: Fair Queue packet scheduler [LWN.net]. https://lwn.net/Articles/565421/

[7] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* 42, 5 (July 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[8] T. Høiland-Jørgensen, D. Täht, and J. Morton. 2018. Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways. In *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 37–42. https://doi.org/10.1109/LANMAN.2018.8475045

[9] V. Jacobson. 1988. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols (SIGCOMM '88)*. ACM, New York, NY, USA, 314–329. https://doi.org/10.1145/52324.52356 event-place: Stanford, California, USA.

[10] Mirja Kühlewind and Brian Trammell. 2018. The QUIC Latency Spin Bit. https://tools.ietf.org/html/draft-ietf-quic-spin-exp-01

[11] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay. *Queue* 10, 5 (May 2012), 20:20–20:34. https://doi.org/10.1145/2208917.2209336

[12] M. Shreedhar and G. Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking* 4, 3 (June 1996), 375–385. https://doi.org/10.1109/90.502236

[13] D. Taht, Jim Gettys, T. Hoeiland-Joergensen, Toke Hoeiland-Joergensen, Eric Dumazet, J. Gettys, E. Dumazet, and P. McKenney. 2018. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. https://tools.ietf.org/html/rfc8290