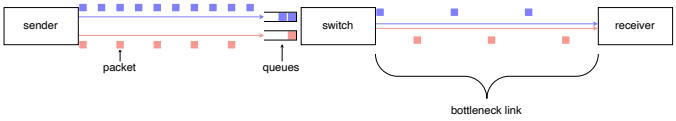
Fair Queuing Aware Congestion Control

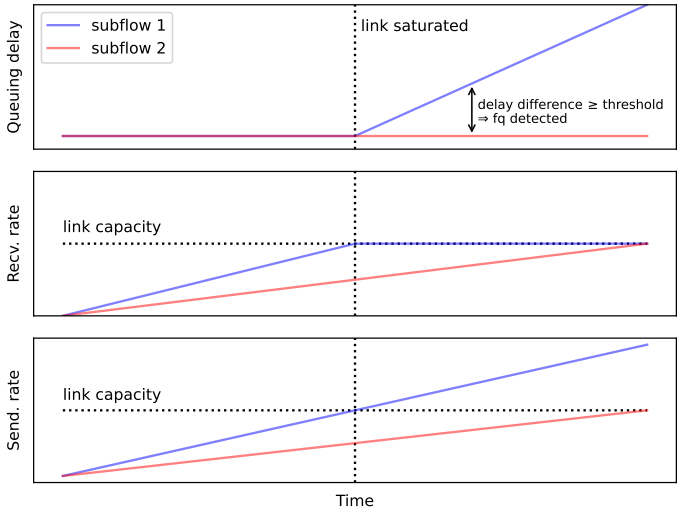
Maximilian Bachl

https://github.com/muxamilian/fair-queuing-aware-congestion-control

***Abstract*—Fair queuing is becoming increasingly prevalent in the internet and has been shown to improve performance in many circumstances. Performance could be improved even more if endpoints could detect the presence of fair queuing on a certain path and adjust their congestion control accordingly. If fair queuing is detected, the congestion control would not have to take cross traffic into account, which allows for more flexibility. In this paper, we develop the first algorithm that continuously checks if fair queuing is present on a path, with very high accuracy. When fair queuing is detected, a different congestion control can be chosen, which can result in reduced latency. Also, each flow can then specify how much queuing delay it allows, meaning that it can choose its own tradeoff between throughput and latency.**

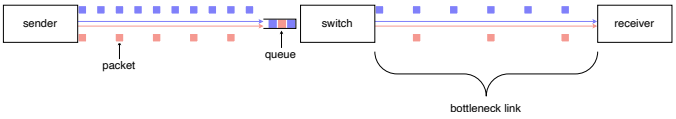


(a) A host sending data to a receiver. There is a bottleneck link with a switch which applies **fair queuing**.

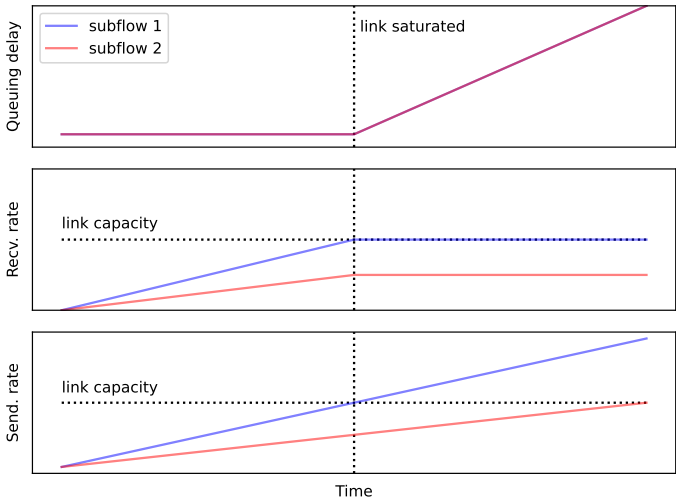


(b) Queuing delay in case of **fair queuing**.

Fig. 1: In case there is **fair queuing**, when the bottleneck link is congested, subflow 1, which sends more, sees increasing queuing delay. Subflow 2 doesn't see increased queuing delay as it sends less than its fair share and thus its packets don't have to wait at the bottleneck link.



(a) A host sending data to a receiver. There is a bottleneck link with a switch which applies **no fair queuing**.



(b) Queuing delay in case there is **no fair queuing**.

Fig. 2: In case there is **no fair queuing**, when the bottleneck link is congested, a queue builds up. Since both subflows share the same queue, they both see increased queuing delay.

I. Background

When different applications send packets over the internet, one application can send more than the other and thus unfairly take a larger share of bandwidth. Also, a large queue can form if one flow sends more than it should. This can result in unfairness and bad user experience. Several different approaches have been proposed to address this [1], [2]: One is to make sure every network flow is "well-behaved" (also known as TCP friendly). Another one is to enforce fairness at switches and routers, called "fair queuing" or "flow queuing" [3].

While fair queuing was proposed decades ago, it only gained popularity in the last couple of years because of implementations in the Linux kernel [4], [5] and also in Apple's macOS. Applications can benefit from increasing deployment of fair queuing: It makes sure that not the most aggressive one wins. It would be even better if applications could know if the connection they're sending on is managed by fair queuing. Then they could be sure that they can use their preferred congestion control mechanism, while not bothering or being bothered by other network flows. We proposed the first such approach in previous work [6] but our previous approach had some shortcomings upon which we improve in this paper.

In our previous work we proposed a technique which determines the presence of fair queuing at flow startup, which works as follows: If fair queuing is successfully detected at flow

startup, a congestion control was used which aimed to keep queuing delay low (delay-based congestion control). While this delay-based congestion control achieved high throughput and low delay, similar to the Vegas congestion control algorithm [7], it was vulnerable to be outcompeted by other network flows sending more aggressively, such as [8]–[10]. This means that our delay-based congestion control performed well but only when it wouldn't have to compete with other flows. Thus it is only used if fair queuing is detected. If the absence of fair queuing is detected, our previous approach uses a more aggressive congestion control (PCC [9]), which can compete better with other, more aggressive network flows, but doesn't keep delay as low as our delay-based congestion control.

While our previous approach had high detection accuracy (98%), it also had some limitations:

- It would **only** detect fair queuing only at **flow startup**. But if the bottleneck link changes during a flow, it could be that the previous bottleneck had fair queuing while the new one doesn't. This wouldn't have been detected.
- It would detect fair queuing only after filling the queue at the bottleneck completely, **causing packet loss**.

We would like to have an approach which

- **continuously checks** for the presence or absence of fair queuing, not only at flow startup.
- **doesn't cause packet loss** while trying to determine if there is fair queuing or not.
- can be **transparently used** on top of any congestion control algorithm.
- lets each application **choose how much delay** it allows in case fair queuing is detected. Allowing a higher queuing delay can result in higher throughput.

II. Concept

Our new approach presented here is a mechanism which performs measurements to determine if there is fair queuing. Our mechanism can be added on top of any existing congestion control algorithm. The approach of performing measurements in congestion control became popular in the last couple of years and was already followed by [8], [9], [11], [12].

The core concept of our approach, which we call ***Tonopah***, is that a network flow is separated into **two subflows**. One flow constantly sends more data (**dominant subflow**), while the other one sends less (**non-dominant subflow**). When the total sending rate reaches the link capacity and fair queuing is present, **fair queuing** is going to **limit** the throughput of the **dominant subflow** (Figure 1). Since its throughput is limited, a queue is going to build up and delay is going to increase. If the delay only increases for the dominant subflow but not for the non-dominant one, this means that there's fair queuing.

On the other side, if there is no fair queuing, when the sending rate exceeds the capacity of the bottleneck link, a queue is going to build up and queuing delay is going to rise for **both** the dominant and the non-dominant subflow (Figure 2). In this case, Tonopah doesn't behave different from a network flow which doesn't have fair queuing detection. This can be a potential advantage which a Tonopah-enabled congestion control can have over other congestion control algorithms such as BBR, which has been shown to be unfair to other flows sometimes [13], [14].

Thanks to the use of Multipath QUIC [15] the application does not see any difference because all data are recombined to one flow by Multipath QUIC.

An interesting aspect of fair queuing detection is that the presence of fair queuing can be interpreted as a *congestion signal*: Only if there is congestion, fair queuing can be detected. If the link is underutilized, flows don't "fight" for bandwidth and thus fairness doesn't need to be enforced at the bottleneck link. Thus, in this paper, we define a new congestion signal: The presence of fair queuing. Other already known congestion signals are, for example: packet loss, queue length, change of queue length, Explicit Congestion Notification [16], [17].

A. Details

The dominant subflow receives $\frac{2}{3}$ of the total allowed sending rate, while the non-dominant one receives the remaining $\frac{1}{3}$.

After each round-trip time, Tonopah checks if the queuing delay of dominant subflow was on average larger than the one of the non-dominant subflow. As the threshold we use 5 ms: If the average queuing delay in the last round-trip time was 5 ms higher for the dominant subflow compared to the non-dominant one, fair queuing is detected. However, it is worth noting that each application can choose this threshold individually. For example, a cloud gaming application might choose a threshold of 5 ms and thus prioritize low delay over high throughput. However, a video chat application might want to emphasize high bandwidth more and thus choose a threshold of 20 ms to get maximum throughput.

Both subflows are controlled by the same congestion window. The differences in sending rates are achieved by using pacing. The basis of the implementation of Tonopah is *Picoquic*[1]. Picoquic includes support for Multipath QUIC and pacing, which we need for the implementation of Tonopah. Also, Picoquic implements several congestion control algorithms. We base Tonopah on the NewReno implementation of Picquic. When fair queuing is detected, the congestion window is reduced by $\frac{1}{8}$. This worked well for our proof-of-concept implementation but one could also specify any other behavior in case fair queuing is detected or even do nothing at all.

III. Evaluation

Tonopah was implemented on top of Picoquic in C. It was added on top of the NewReno congestion control algorithm but it could also be added on top of other congestion control algorithms as well. By basing our implementation on Picoquic, Tonopah also supports Explicit Congestion Notification.

We evaluated Tonopah on an Apple MacBook Air from 2011 with an Intel Core i5-2557M CPU at 1.7 GHz. By

[1] https://github.com/private-octopus/picoquic/tree/master/picoquic

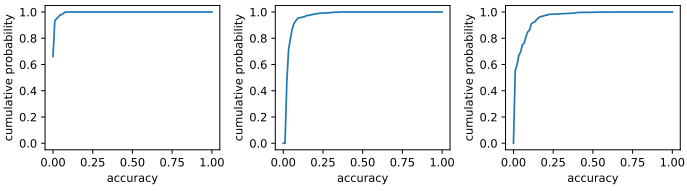
choosing old hardware for experimentation, we can show that our algorithm doesn't use excessive compute and can also run well on weak hardware.

We emulate the network using *mininet* and use Linux's *pfifo* queuing discipline in the case there is no fair queuing and *fq fq_codel* for fair queuing. For the evaluation we choose a simple network in which the sender is connected to the receiver via a switch.

All the source code and evaluation code of this paper is freely available (refer to the very top of this paper).

In the following evaluations we always run each network flow for 90 s. Tonopah measures whether there is fair queuing or not continuously. If in the evaluation scenario there is fair queuing and Tonopah detects fair queuing correctly for 80 s and fails to detect it for 10 s, we state the accuracy as $\frac{80}{90} = 89\%$. We evaluate the performance of Tonopah over a range of delays from 10 to 100 ms and link capacity from 10 to 100 Mbit/s in a grid-like fashion. For each combination of delay and link capacity, multiple experiment runs are performed and the median, the quartiles or the average of the results are shown. As the buffer size at the bottleneck we set the bandwidth delay product but at least 100 packets when no fair queuing (pfifo) is used. When using fq for fair queuing, we set the buffer size for each flow using the same rule. When using fair queuing with fq_codel, we use it in its default configuration.

A. Detection accuracy



(a) pfifo (b) fq_codel (c) fq

Fig. 3: Empirical cumulative distribution functions for detection of the abscence/presence of fair queuing.

Figure 4 shows that if there is no fair queuing, Tonopah detects this correctly virtually always. If there is fair queuing with fq_codel (Figure 5) or fq (Figure 6), Tonopah detects this correctly very often but not in every case. Figure 3 shows the empirical cumulative distribution functions.

B. Throughput and Queuing Delay of Tonopah

While the goal of this paper is to implement a mechanism to detect fair queuing in an online fashion, we also want to show that fair queuing detection can be used to lower the queuing delay drastically while not impairing throughput. Table I shows that Tonopah can lower queuing delay by more than two thirds while keeping throughput unchanged, when it is implemented on top on NewReno. However, we want to emphasize that Tonopah is not a congestion control algorithm but just a mechanism to measure fair queuing which can be used with any congestion control algorithm.

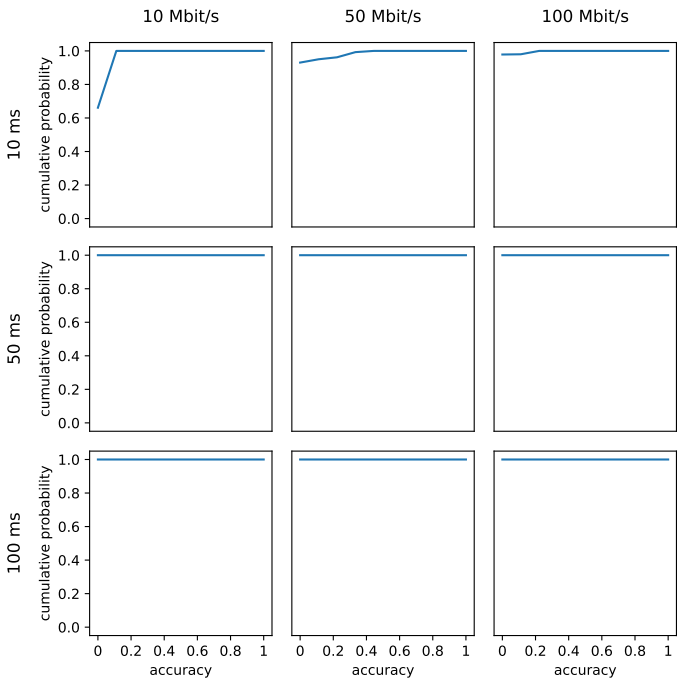


Fig. 4: Detection accuracy in case there's **no fair queuing** (pfifo). The overall accuracy is 99.4%.

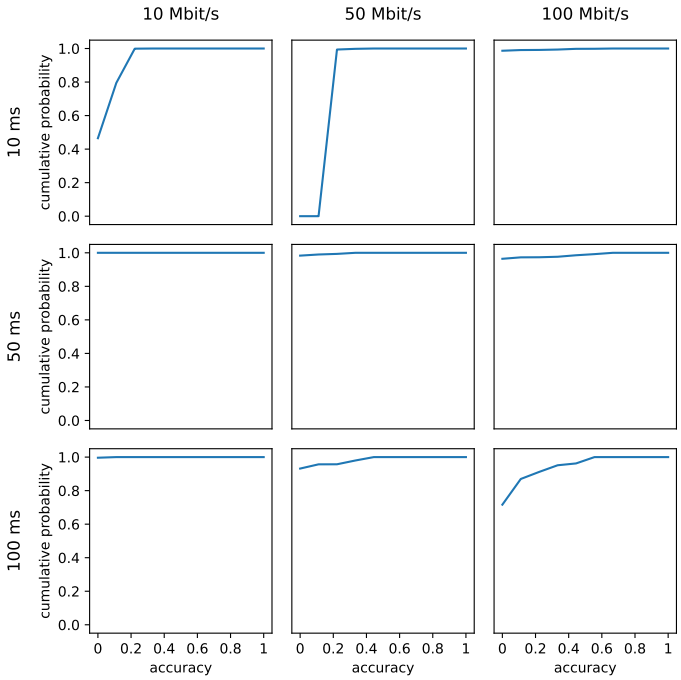


Fig. 5: Detection accuracy in case there is **fair queuing** (fq_codel). The overall accuracy is 95.8%.

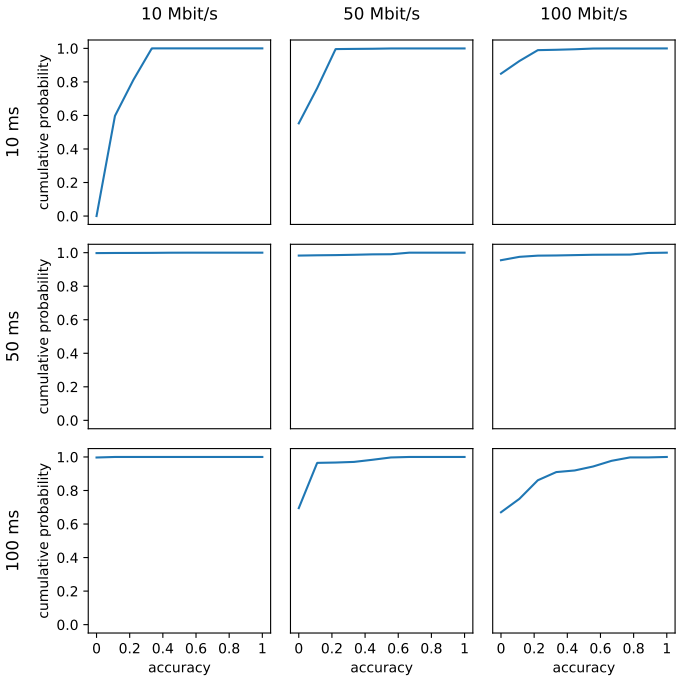
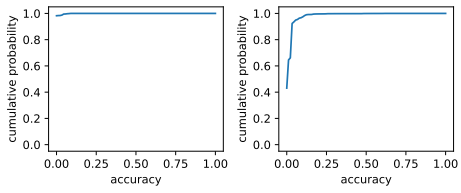


Fig. 6: Detection accuracy in case there is **fair queuing** (fq). The overall accuracy is 95.4%.

	Link utilization	Queuing delay
NewReno	95.6%	36.5 ms
Tonopah	91.1%	3.3 ms

TABLE I: NewReno's link utilization is 105% Tonopah's. But the **queuing delay** of NewReno is **1111%** Tonopah's. This means that Tonopah can deliver the same or slightly less throughput while causing a lot less queuing delay. The differences in **queuing delay** are **highly significant** (p-value (Welch's t-test) $< 10^{-21}$). The experiment setup is the same as in Figure 6.

C. *Cross-traffic*



(a) pfifo (b) fq

Fig. 7: Empirical cumulative distribution functions for detection of the abscence/presence of fair queuing when there is cross-traffic.

To evaluate Tonopah's performance under cross-traffic, we use iperf3 with the NewReno congestion control. We start iperf 4 s before Tonopah so that it has time to saturate the link. Figure 8 and Figure 9 show that also in the presence of cross-traffic Tonpah's detection accuracy is virtually unchanged or even slightly better. We tried to use fq_codel as in Figure 6 but

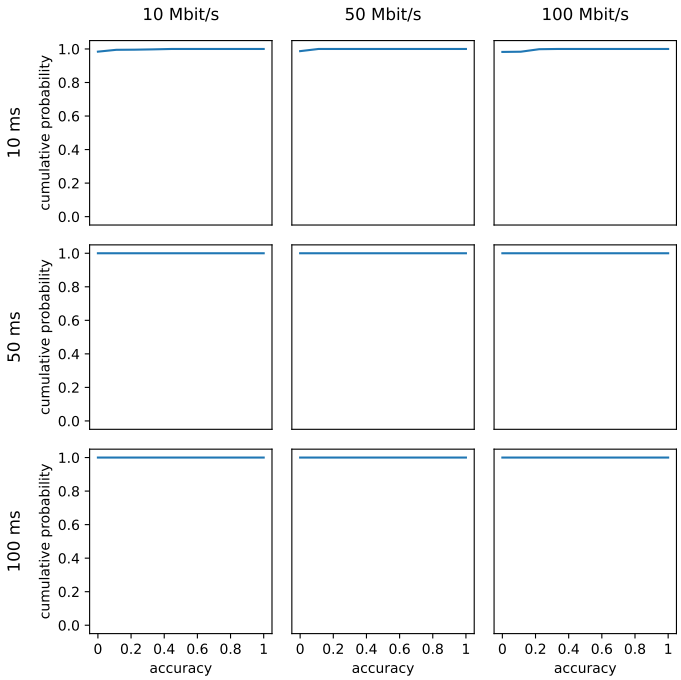


Fig. 8: Detection accuracy in case there's **no fair queuing** under the presence of **cross-traffic**. The overall accuracy is 99.9%.

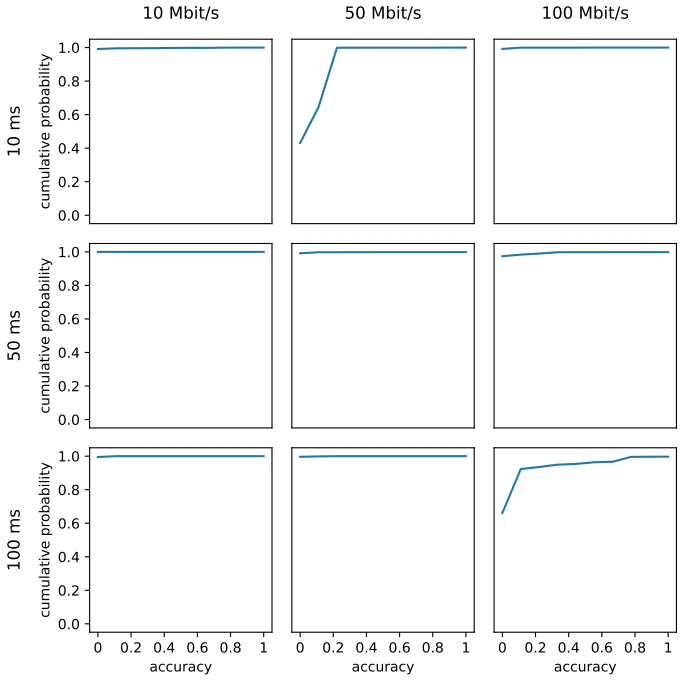


Fig. 9: Detection accuracy in case there is **fair queuing** (fq) under the presence of **cross-traffic**. The overall accuracy is 98.1%.

we encountered problems when using fq_codel in combination with iperf3. Frequently, iperf3 would lose all packets, have a timeout after a couple of seconds and stop. Since we could not resolve this issue, we used fq instead of fq_codel, for which this issue didn't occur.

Figure 7 shows the empirical cumulative distribution functions when there is cross-traffic.

IV. Discussion

We showed that continuous fair queuing detection is feasible and can achieve a high detection accuracy. We also showed that it can work with alternative versions of fair queuing such as fq_codel although experiments showed that accuracy depends on the configuration of the queue manager, the link capacity, base delay etc. Further experiments could be done to finetune Tonopah and also experiment with other queue managers such as *cake* [18], *PIE* [19], *cocoa* [20] or *LFQ* [21].

Another potentially fruitful line of research would be to integrate Tonopah into other congestion control algorithms such as BBR.

Acknowledgements

We thank Sebastian Moeller, who suggested to add CDF plots and to use fq_codel with a target value of 5 ms, which is the default on Linux.

References

[1] L. Brown, G. Ananthanarayanan, E. Katz-Bassett, A. Krishnamurthy, S. Ratnasamy, M. Schapira, and S. Shenker, "On the Future of Congestion Control for the Public Internet," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020.

[2] R. Ware, M. K. Mukerjee, S. Seshan, and J. Sherry, "Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019.

[3] J. Nagle, "On Packet Switches With Infinite Storage," Request for Comments RFC 970, Internet Engineering Task Force, 1985.

[4] E. Dumazet, "pkt_sched: fq: Fair Queue packet scheduler [LWN. net]," 2013.

[5] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The flow queue codel packet scheduler and active queue management algorithm," tech. rep., 2018.

[6] M. Bachl, J. Fabini, and T. Zseby, "Detecting Fair Queuing for Better Congestion Control," Tech. Rep. arXiv:2010.08362, arXiv, Feb. 2021. arXiv:2010.08362 [cs] type: article.

[7] L. Brakmo and L. Peterson, "TCP Vegas: end to end congestion avoidance on a global Internet," *IEEE Journal on Selected Areas in Communications*, 1995.

[8] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, 2016.

[9] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance," 2015.

[10] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, 2008.

[11] P. Goyal, A. Narayan, F. Cangialosi, S. Narayana, M. Alizadeh, and H. Balakrishnan, "Elasticity Detection: A Building Block for Internet Congestion Control," Tech. Rep. arXiv:1802.08730, arXiv, 2020.

[12] D. A. Hayes, M. Welzl, S. Ferlin, D. Ros, and S. Islam, "Online Identification of Groups of Flows Sharing a Network Bottleneck," *IEEE/ACM Transactions on Networking*, 2020. Publisher: IEEE.

[13] R. Ware, M. K. Mukerjee, S. Seshan, and J. Sherry, "Modeling BBR's Interactions with Loss-Based Congestion Control," in *Proceedings of the Internet Measurement Conference*, 2019.

[14] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of BBR congestion control," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct. 2017.

[15] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Multipath Extension for QUIC," Internet Draft draft-ietf-quic-multipath-01, Internet Engineering Task Force, 2022.

[16] M. Mathis, "Relentless Congestion Control," 2009.

[17] D. A. Hayes and G. Armitage, "Revisiting TCP Congestion Control Using Delay Gradients," in *NETWORKING 2011*, pp. 328–341, 2011.

[18] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of CAKE: a comprehensive queue management solution for home gateways," in *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2018.

[19] 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 *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, July 2013.

[20] M. Bachl, J. Fabini, and T. Zseby, "Cocoa: Congestion Control Aware Queuing," in *Proceedings of the 2019 Workshop on Buffer Sizing*, 2019.

[21] M. Bachl, J. Fabini, and T. Zseby, "LFQ: Online Learning of Per-flow Queuing Policies using Deep Reinforcement Learning," in *2020 IEEE 45th Conference on Local Computer Networks (LCN)*, Nov. 2020.