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. In case fair queuing is detected, each flow can specify how much queuing delay it allows, meaning that each flow can choose its own tradeoff between throughput and latency.**

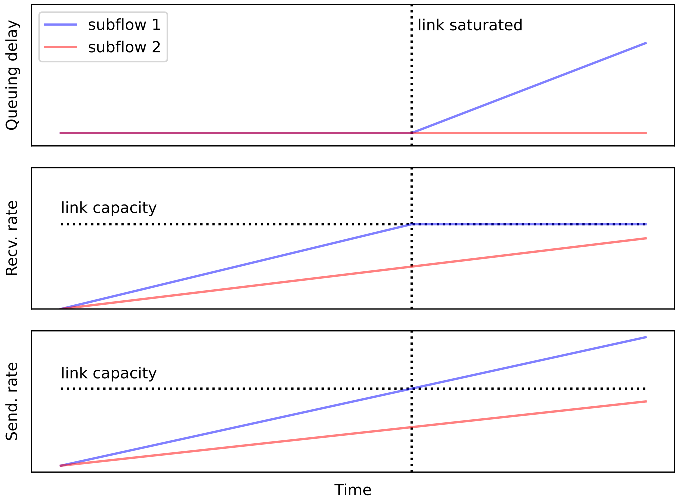

Fig. 1: In case there is **fair queuing**, when the bottleneck link is congested, subflow 1, which sends more, sees increasing queuing delay. Subflow 2 doesn't see increased queuing delay as it sends less than its fair share and thus its packets don't have to wait at the bottleneck link.

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.

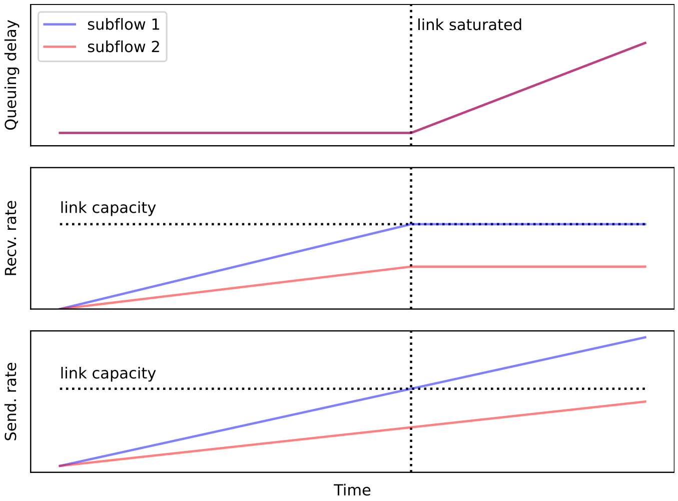

Fig. 2: In case there is **no fair queuing**, when the bottleneck link is congested, a queue builds up. Since both subflows share the same queue, they both see increased queuing delay.

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 (specifically 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 thus 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.

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**. **Alternatingly**, one flow sends more (**dominant subflow**), while the other one sends less (**non-dominant subflow**). After a certain time interval the dominant subflow becomes the non-dominant one and vice versa. 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). This can be measured by observing the *delivery rate* [1] of both subflows. If there is no fair queuing, the subflow which sends more also achieves higher throughput (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].

By alternating which subflow is dominant and which one is non-dominant, we prevent the build-up of a standing queue at the bottleneck: When the dominant subflow builds up a queue at the bottleneck link, it is going to be the non-dominant subflow in the next time interval and then it is going to send less and the queue is going to be reduced.

The congestion window of the flow applies to both subflows together. This means that while always the dominant subflow sends more than the non-dominant one, in sum they never send more than what is allowed by the congestion window.

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 bandwidh 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 $\frac{6}{10}$ of the total allowed sending rate, while the non-dominant one receives $\frac{4}{10}$. The dominant subflow and the non-dominant subflow alternate every $\max(50, \min(100, rtt))$ milliseconds. This time period is also called an ***interval***. To determine the presence of fair queuing, the data of the last four intervals are evaluated. We look at more than one interval because this reduces the effect of measurement noise. We found four intervals to be a suitable number.

We determine that fair queuing is present if the receiving rate was more fair than not fair. It would be completely fair if both flows get $\frac{1}{2}$ of the total throughput. It would be completely unfair if the dominant subflow got $\frac{6}{10}$ and the non-dominant one $\frac{4}{10}$. If the dominant one got a fraction of 0.549 of the total throughput, it would be just a little bit more fair than unfair. In this case we would say that there is fair queuing. If, on the other side, the dominant subflow got a fraction of 0.551 of the total throughput, we would say that there is no fair queuing.

Another safeguard against measurement noise is that Tonopah only take a decision regarding fair queuing if the dominant flow actually really sent more data than the non-dominant one. Specifically, if the dominant flow sends a fraction of throughput smaller than 0.575, Tonopah doesn't take a decision because it means that probably there wasn't enough data to send. This case only occurs when the underlying application has no data to send or when the sending rate is very low ($<$ 10 Mbit/s).

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*[2]. Picoquic includes support for Multipath QUIC and pacing, which we need for the implementation of Tonopah. Also, it implements 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.

[1] The delivery rate (a concept popularized by BBR [8]) can be measured by observing the rate at which acknowledgement packets are sent back to the sender by the receiver. If the sender sends more than the link capacity, the delivery rate is never going to exceed the link capacity since the receiver cannot receive packets faster than the link capacity allows.

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

	10 Mbit/s	50 Mbit/s	100 Mbit/s
10ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
50ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
100ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%

TABLE I: Detection accuracy in case there's **no fair queuing**. The overall median accuracy (true negative rate) is 100%, the first quartile is 100% and the third quartile is 100%.

	10 Mbit/s	50 Mbit/s	100 Mbit/s
10ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 99% 3rd quart.: 100%
50ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
100ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%

TABLE II: Detection accuracy in case there is **fair queuing**. The overall median accuracy (true positive rate) is 100%, the first quartile is 100% and the third quartile is 100%.

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 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* for fair queuing. For 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.

	Link utilization	Queuing delay
NewReno	95.6%	33.6 ms
Tonopah	94.3%	10.5 ms

TABLE III: NewReno's link utilization is 103% Tonopah's. But the **queuing delay** of NewReno is **321%** Tonopah's. This means that Tonopah can deliver the same throughput while causing a lot less queuing delay. The differences in **queuing delay** are **highly significant** (p-value[1] $< 10^{-5}$). The experiment setup is the same as in Table II.

	10 Mbit/s	50 Mbit/s	100 Mbit/s
10ms	Median: 99% 1st quart.: 98% 3rd quart.: 99%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
50ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
100ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 98% 3rd quart.: 100%

TABLE IV: Detection accuracy in case there's **no fair queuing** under the presence of **cross-traffic**. The overall median accuracy (true negative rate) is 100%, the first quartile is 100% and the third quartile is 100%.

A. Detection accuracy

Table I shows that if there is no fair queuing, Tonopah detects this correctly virtually always (true negative rate ≥99%). If there is fair queuing (Table II), Tonopah detects this correctly very often but not in every case (true positive rate in the high 90s).

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 III 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.

C. Cross-traffic

To evaluate Tonopah's performance under crosstraffic, we use iperf with the NewReno congestion control. We start iperf 4 s before Tonopah so that it has time to saturate the link. Table IV and Table V show that also in the presence of cross-traffic Tonpah's detection accuracy is virtually unchanged.

D. Other variants of fair queuing: fq_codel

We also evaluated the performance of our fair queuing detection on a bottleneck managed by fq_codel [5]. We chose a default target queuing delay of 10 ms following Apple's

[1] Using Welch's t-test

	10 Mbit/s	50 Mbit/s	100 Mbit/s
10ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
50ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
100ms	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%

TABLE V: Detection accuracy in case there is **fair queuing** under the presence of **cross-traffic**. The overall median accuracy (true positive rate) is 100%, the first quartile is 100% and the third quartile is 100%.

	10 Mbit/s	50 Mbit/s	100 Mbit/s
10ms	Median: 85% 1st quart.: 84% 3rd quart.: 85%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
50ms	Median: 91% 1st quart.: 88% 3rd quart.: 92%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 98% 1st quart.: 97% 3rd quart.: 99%
100ms	Median: 9% 1st quart.: 7% 3rd quart.: 13%	Median: 100% 1st quart.: 100% 3rd quart.: 100%	Median: 64% 1st quart.: 0% 3rd quart.: 75%

TABLE VI: Detection accuracy in case there is the ***fq_codel*** variant of **fair queuing** with a target delay of 10 ms. The overall median accuracy (true positive rate) is 95%, the first quartile is 83% and the third quartile is 100%.

implementation[3] because we argue that Apple probably spent a considerable amount of time fine-tuning their implementation and came to the conclusion that 10 ms work best as the default target delay. Also, Apple's implementation of fq_codel seems to be the most widespread worldwide[4] with currently two billion Apple devices presumably supporting Apple's fq_codel.

Table VI shows that Tonopah doesn't detect fair queuing as well when there's fq_codel compared to when there is fq. However, median detection accuracy (true positive rate) is still reasonably high at 95%.

	10 Mbit/s	50 Mbit/s	100 Mbit/s
10ms	Median: 39% 1st quart.: 37% 3rd quart.: 80%	Median: 100% 1st quart.: 99% 3rd quart.: 100%	Median: 100% 1st quart.: 100% 3rd quart.: 100%
50ms	Median: 36% 1st quart.: 36% 3rd quart.: 36%	Median: 100% 1st quart.: 99% 3rd quart.: 100%	Median: 99% 1st quart.: 98% 3rd quart.: 99%
100ms	Median: 16% 1st quart.: 14% 3rd quart.: 17%	Median: 99% 1st quart.: 98% 3rd quart.: 100%	Median: 97% 1st quart.: 64% 3rd quart.: 99%

TABLE VII: Detection accuracy in case there is the ***fq_codel*** variant of **fair queuing** with a target delay of 5 ms. The overall median accuracy (true positive rate) is 98%, the first quartile is 37% and the third quartile is 100%.

1) fq_codel with a target delay of 5 ms: Although Apple's fq_codel implementation with its 10 ms default target delay seems to be the most widespread, fq_codel is also the default queuing manager on multiple Linux distributions. On Linux, fq_codel has a default target delay of 5 ms, half of Apple's value of 10.

Since the target delay of 5 ms is quite small, we had to slightly modify Tonopah: We fixed the interval of Tonopah to 50 ms. As a reminder, the interval is the time duration after which the dominant subflow becomes the non-dominant one and vice versa. As a reminder, for the previous experiments, the interval was the current round trip time but at least 50 ms and no more than 100 ms.

Table VII shows the performance of Tonopah with Linux's implementation of fq_codel. Median detection accuracy (true positive rate) is 98%. A problem is that accuracy is pretty low for small bandwidths ($\leq$10 Mbit/s) but otherwise fair queuing can be recognized quite accurately.

Since we changed how Tonopah works to make it work better with Linux's implementation of fq_codel by setting the interval to 50 ms, we also have to make sure that this modification didn't lower the true negative rate (accuracy in case there is no fair queuing). Our experiments show that with the constant 50 ms interval Tonopah even works slightly better, giving a true negative rate of 100%, meaning it can always detect the absence of fair queuing correctly.

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 also include experiments with fq_codel with a target value of 5 ms, which is the default on Linux.

[3] https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/bsd/net/if.h#L262

[4] https://blog.cerowrt.org/post/state_of_fq_codel/

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.