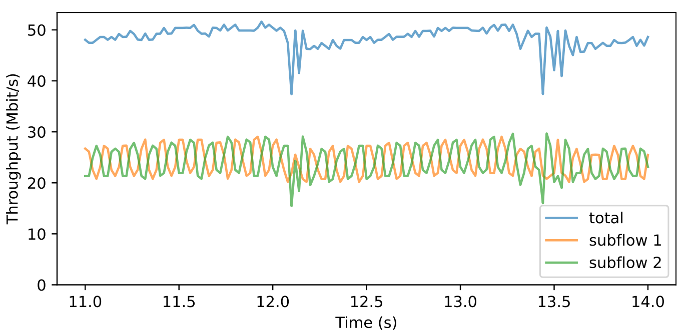
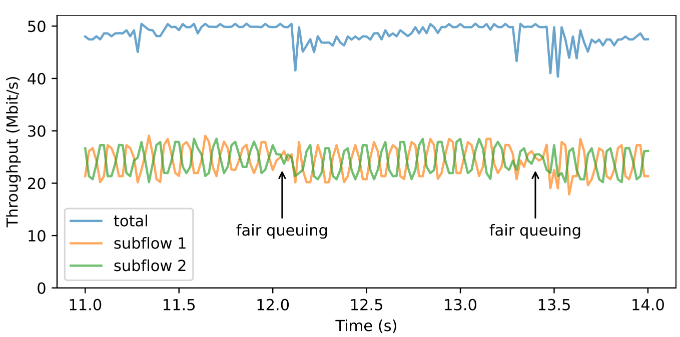
Fair Queuing Aware Congestion Control

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. When fair queuing is detected, a different congestion control is chosen, which results in reduced latency. Unlike an algorithm proposed in a previous paper of us, the approach presented here does not only detect the presence of fair queuing once at flow startup but it does so continuously. We show that by adding fair queuing detection to an existing congestion control algorithm, queuing delay can be reduced by more than two thirds.**

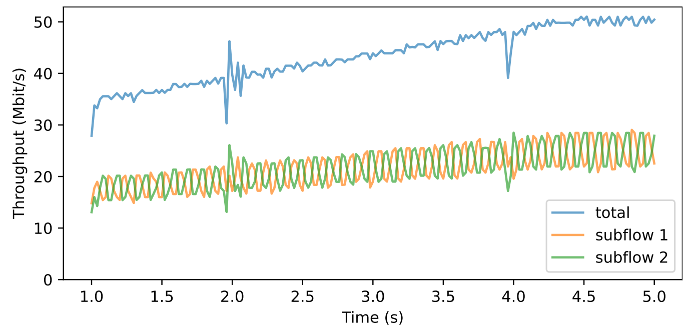


(a) **Sending** rate (fair queuing)

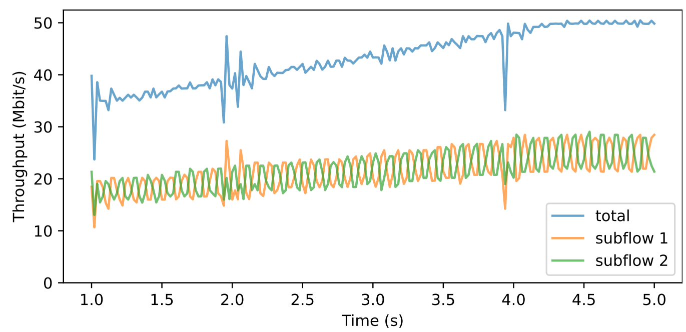


(b) **Receiving** rate (fair queuing)

Fig. 1: Figure 1a shows the sending rate, 1b the receiving rate in case there's **fair queuing**. Around seconds 12 and 13.5, the sending rate reaches the maximum of the link – 50 Mbit/s – and fair queuing starts to limit the throughput of the flow that is sending more. Thus, in the lower figure, the dominant subflow and the non-dominant subflow achieve approx. the same receiving rate even though the dominant subflow sends more (dominant subflow). This is the effect of fair queuing.



(a) **Sending** rate (no fair queuing)



(b) **Receiving** rate (no fair queuing)

Fig. 2: Figure 2a shows the sending rate, 2b the receiving rate in case there's **no fair queuing**. Even though the sender sends too much and a queue builds up, still the flow that sends more data also has a higher receiving rate. This is in contrast to Figure 1, where both flows have the same receiving rate once there is congestion at the bottleneck, thanks to fair queuing.

I. Background

When different applications send packets on the internet, one application can send more than the other and thus unfairly take a larger share of bandwidth. 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]. 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 the can use a common 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.

Our approach is a congestion control mechanism which also performs measurements to determine if there is fair queuing. This approach of performing measurements in congestion control became popular in the last couple of years and was already followed by [7]–[10].

II. Introduction

In our previous work we proposed a technique which determines the presence of fair queuing at flow startup [6], which worked 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, it was vulnerable to be outcompeted by other network flows sending more aggressively, such as [7], [8], [11], similar to the Vegas congestion control algorithm [12]. This means that our delay-based congestion control performed well but only when it wouldn't have to compete with other flows. Thus is only used if fair queuing is detected. If it is detected that there is no fair queuing, our approach uses a more aggressive congestion control (specifically PCC [8]), which can compete better with other 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 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.

III. Concept

The core concept of our approach, which we call ***Tonopah***, is that a network flow is separated in **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 a higher throughput (Figure 2), which can be considered unfair. 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 always send the same amount of data, as allowed by the congestion window.

Thanks to the use of Multipath QUIC [13] 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 [14], [15].

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 $\max(50, \min(100, \mathit{rtt}))$ 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 we 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 less than 57.5%, we

[1] The delivery rate (a concept popularized by BBR [7]) 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.

	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 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 is 100%, the first quartile is 100% and the third quartile is 100%.

don't take a decision because the pacing apparently wasn't accurate enough. This case only occurs when the sending rate is very low (< 10 Mbit/s) and thus pacing in inaccurate.

Both subflows are controlled by the same congestion window. The differences in sending rates are implemented 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.

IV. Evaluation

Tonopah was implemented on top of Picquic in C. It was added on top of the NewReno congestion control algorithm but it could easily 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 1.70GHz. 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.

A. 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

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

	Link utilization	Queuing delay
NewReno	95.6%	10.5 ms
Tonopah	94.3%	33.6 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 cause a lot less queuing delay. The differences in **queuing delay** are **highly significant** (p-value[1] $< 10^{-5}$).

[1]Using Welch's t-test

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

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

B. Cross-traffic

C. Other variants of fair queuing

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 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.

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

	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 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**. The overall median accuracy is 95%, the first quartile is 83% and the third quartile is 100%.

V. Discussion

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*, (Virtual Event USA), pp. 30–37, ACM, Nov. 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*, (Princeton NJ USA), pp. 17–24, ACM, Nov. 2019.

[3] J. Nagle, "On Packet Switches With Infinite Storage," Request for Comments RFC 970, Internet Engineering Task Force, Dec. 1985. Num Pages: 9.

[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] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, September-October, pp. 20 – 53, 2016.

[8] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance," pp. 395–408, 2015.

[9] 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, Feb. 2020. arXiv:1802.08730 [cs] type: article.

[10] 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*, vol. 28, no. 5, pp. 2229–2242, 2020. Publisher: IEEE.

[11] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 64–74, July 2008.

[12] L. Brakmo and L. Peterson, "TCP Vegas: end to end congestion avoidance on a global Internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 1465–1480, Oct. 1995. Conference Name: IEEE Journal on Selected Areas in Communications.

[13] 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, Mar. 2022. Num Pages: 28.

[14] M. Mathis, "Relentless Congestion Control," p. 4, 2009.

[15] D. A. Hayes and G. Armitage, "Revisiting TCP Congestion Control Using Delay Gradients," in *NETWORKING 2011* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, eds.), vol. 6641, pp. 328–341, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.