http://www.diva-portal.org

Preprint

This is the submitted version of a paper presented at *29th IEEE Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC). Bologna, Italy. 9-12 september, 2018.*.

Citation for the original published paper:

Grazia, C A., Patriciello, N., Høiland-Jørgensen, T., Klapez, M., Casoni, M. et al. (2018)
Adapting TCP Small Queues for IEEE 802.11 Networks
In: *2018 IEEE 29Th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)* IEEE
IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications workshops
https://doi.org/10.1109/PIMRC.2018.8581048

N.B. When citing this work, cite the original published paper.

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-68631

Adapting TCP Small Queues for IEEE 802.11 Networks

Carlo Augusto Grazia*, Natale Patriciello†, Toke Høiland-Jørgensen‡, Martin Klapez*
Maurizio Casoni* and Josep Mangues-Bafalluy†
*Department of Engineering *Enzo Ferrari*, University of Modena and Reggio Emilia
via Pietro Vivarelli 10, 41125 Modena, Italy. Email: carloaugusto.grazia@unimore.it
†Centre Tecnològic de Telecomunicacions de Catalunya (CTTC/CERCA)
Av. Carl Friedrich Gauss 7, 08860 Castelldefels, Barcelona, Spain.
‡Department of Mathematics and Computer Science, Karlstad University, 651 88 Karlstad, Sweden

***Abstract*—In recent years, the Linux kernel has adopted an algorithm called TCP Small Queues (TSQ) for reducing queueing latency by controlling buffering in the networking stack. This solution consists of a back-pressure mechanism that limits the number of TCP segments within the sender TCP/IP stack, waiting for packets to actually be transmitted onto the wire before enqueueing further segments. Unfortunately, TSQ prevents the frame aggregation mechanism in the IEEE 802.11n/ac standards from achieving its maximum aggregation, because not enough packets are available in the queue to build aggregates from, which severely limits achievable throughput over wireless links. This paper demonstrates this limitation of TSQ in wireless networks and proposes Controlled TSQ (CoTSQ), a solution that improves TSQ so that it controls the amount of data buffered while allowing the IEEE 802.11n/ac aggregation logic to fully exploit the available channel and achieve high throughput. Results on a real testbed show that CoTSQ leads to a doubling of throughput on 802.11n and up to an order of magnitude improvement in 802.11ac networks, with a negligible latency increase.**

***Index Terms*—Frame Aggregation, IEEE 802.11, Small Queues, TCP, Throughput.**

I. Introduction

The Linux networking stack has seen many improvements and much research activity in recent years, with the goals of reducing congestion and latency, and mitigating the bufferbloat phenomenon [1], [2]. This research has included the introduction of novel Active Queue Management (AQM) algorithms such as Codel [3] and PIE [4] as well as novel TCP congestion control algorithms like BBR [5]. Another essential solution, enabled by default in the Linux kernel since 2012, is TCP Small Queues (TSQ), a fine-grained backpressure mechanism that limits each TCP flow to 1 ms of data at the current rate in the TCP/IP stack; subsequent packets from the same flow are enqueued only after the actual NIC transmission. However, not much analysis of TSQ is available in the scientific literature; in fact, only a few research articles have at most cited this algorithm [6]–[8] spending few words, or nothing at all, on the TSQ mechanism itself. To the best of our knowledge, only Guo et al. in [9] investigated TSQ by interacting with the algorithm and changing the threshold for experimental purposes. Unfortunately, this work is tailored for cellular networks only, as it focuses entirely on latency where the TSQ threshold is reduced in order to eliminate queueing as much as possible. The paper concluded that reducing the TSQ threshold has a negligible impact on the firmware queueing delay. In addition, the TSQ tuning provided in [9] cannot be applied to the new Linux kernel versions in which the TSQ policy has been hard-coded in the kernel and cannot be modified by the user. In wired networks, TSQ largely achieves its goal of latency reduction without affecting the maximum achievable throughput. However, the same does not hold for wireless networks, such as IEEE 802.11n/ac WLANs, and this problem has never been discussed in the literature. The contributions of this paper are the following: (i) we demonstrate the throughput degradation in IEEE 802.11n/ac WLANs caused by TSQ; (ii) we propose Controlled TSQ (CoTSQ), a solution to modify and control the TSQ behaviour through a Linux kernel patch and (iii) we evaluate our solution on a real testbed providing an extensive set of tests that show how CoTSQ doubles the achievable throughput in 802.11n networks and improves it by up to an order of magnitude in 802.11ac networks, in both cases with a negligible latency increase. The rest of the paper is organised as follows: Section II details the TSQ algorithm, while Section III presents our CoTSQ solution. Section IV describes the testbed used to produce the results available in Section V. Finally, Section VI concludes the paper.

This work was partially funded by Spanish MINECO grant TEC2017-88373-R (5G-REFINE) and Generalitat de Catalunya grant 2017 SGR 1195.

II. TCP Small Queues in a Nutshell

This section describes the TSQ mechanism, first implemented by Eric Dumazet and available since Linux kernel versions 3.6 [10]. To understand TSQ, we refer to

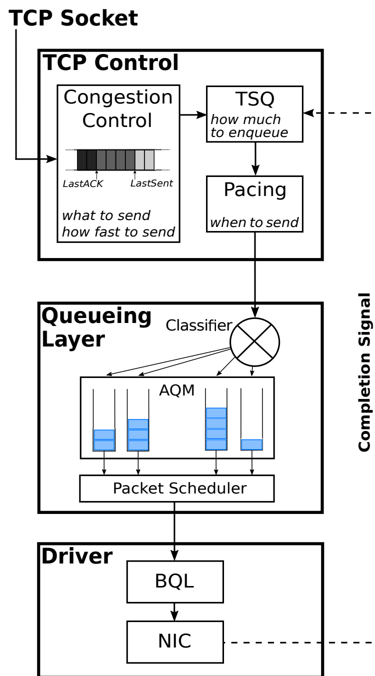


Fig. 1: Linux TCP sender architecture.

Figure 1 that shows the path of TCP packets, from the TCP socket of the host through the different layers of the Linux networking stack: The TCP layer, the queueing layer and the device driver.

At the top of Figure 1 is the TCP layer. When an application transmits data through a TCP socket, it is queued in the socket buffer. The TCP control system determines when packets are dequeued from the socket buffer and moved into the lower levels of the network stack. This control system consists of the Congestion Control algorithm, the TSQ logic and the Pacing mechanism, which can each limit the flow of packets into lower layers. The Congestion Control algorithm is the core engine of the protocol, and each TCP variant (such as Cubic, BBR, Vegas, New Reno, etc.) define different behaviour for this congestion control module. The module implements the standard ACK logic of TCP, maintains the congestion window, etc. TSQ and Pacing have two different goals: the former has the role of limiting the amount of data of any given socket in the stack, while the latter smoothes out transmission of individual packets by spacing them equally according to the current estimated rate, thus preventing big bursts of packets put onto the medium at once. The pacing mechanism is optional and can be activated by the congestion control module or by the sending application, while TSQ and the selected congestion control is always active.

Inside the queueing layer, packets are managed according to the queueing discipline deployed. The queueing layer can implement various queueing mechanisms, ranging from a simple FIFO queue, to elaborate queueing schemes with per-flow queueing and fairness enforcement. In addition, AQM can be implemented in this layer, either as a simple mechanism on top of a FIFO queue, or as part of a more elaborate multi-queue system.

A lot of the scientific work seeking to address bufferbloat has focused on the queueing layer. The Linux kernel implements the queueing layer as a generic *queueing discipline* (qdisc) which is used for all networking interfaces. The current state of the art qdisc (which is used by default in many deployments of Linux) is FQ-Codel [11], which is a fairness queueing algorithm that uses the CoDel AQM to control each queue, and a Deficit Round Robin-based scheduling of the queues. However, for supported WiFi drivers, the kernel instead uses an integrated queueing system based on the FQ-CoDel algorithm, but tailored specifically for WiFi [12].

Once the queueing layer dequeues a packet, the latter moves to the last block in the figure which is the device driver controlling the Network Interface Card (NIC). Here there is a buffer that manages the packets waiting to be sent on the physical link, and the Linux kernel deploys solutions to mitigate bufferbloat also at this low level of the network stack. This solution is named Byte Queue Limits (BQL) and puts a cap on the amount of data that can be enqueued and waits to be delivered by the NIC [13]. The last block also contains the implementation of the IEEE 802.11n/ac aggregation logic, which is implemented either in the device driver or in the device firmware itself. Aggregation combines several packets into a single link-layer frame, reducing overhead and increasing throughput [14], [15].

As seen above, there are queues in several different layers of the network stack, and thus several places in which bufferbloat can occur, each requiring its own solution. TSQ is complementary to the other solutions, and is targeted specifically at applications on the local host that use TCP; by providing backpressure, these applications can better react to congestion signals for their traffic, and achieve lower application-level latency. TSQ controls dequeueing from the socket buffer, and so it limits the enqueued data of any TCP flow regardless of where the data is queued. When the limit is reached, new data of a flow can be enqueued only when previously enqueued packets from the same flow have been dequeued by the NIC, and the packet memory freed. This works by triggering a completion signal that notifies the TSQ logic when the driver frees the memory allocated for a data packet.

The amount of queue allowed by the TSQ logic is adjusted dynamically as a function of the TCP rate; the amount of data that can be enqueued for each flow corresponds to the amount of data that can be transmitted in 1 ms at the current rate of the flow. This policy is fixed in the kernel, and the TSQ logic can be neither disabled nor tuned.

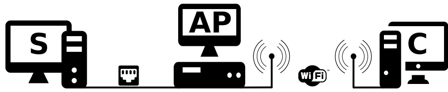


Fig. 2: Physical testbed layout.

III. Controlled TSQ

To overcome the inflexible behavior of TSQ, we patched the kernel to expose the TSQ core parameters and make it possible to disable or tune the TSQ logic. We name this solution Controlled TSQ (CoTSQ) [16]. The CoTSQ logic relies on 3 parameters:

- `bytes` is the TSQ parameter that limits the amount of data, expressed in bytes, for each TCP flow. It is used to impose an upper bound on the queue;
- `ms` limits the amount of data in the stack as a function of the latency. It gives the dynamic threshold, autotuning the number of bytes to enqueue as a function of the current rate;
- `pkts` sets a *lower* bound on the number of packets queued for each TCP flow.

$$B = \min\Big(\texttt{bytes}, \max\big(\texttt{ms}, \texttt{pkts}\big)\Big) \quad (1)$$

These three parameters are combined into the queue limit `B` enforced by TSQ as given in Equation (1). The `ms` and `pkts` parameters are first converted to bytes, by multiplying the former by the current flow rate and the latter by the TCP Maximum Segment Size (MSS). We chose Equation (1) because it is easy to define `bytes` as upper bound (maximum amount of data to be enqueued), `pkts` as lower bound (minimum amount of packets to be enqueued) and `ms` as the core value. The latter gives the actual amount of data to enqueue as a function of the rate. We can express the standard TSQ behavior by setting `bytes`=128KB, `ms`=1 and `pkts`=0 in the CoTSQ parameters. Similarly, we also defined a way to completely disable TSQ by setting `bytes`=-1 (or any negative value) for testing purposes.

IV. Testbed

To evaluate our solution we deployed the physical testbed depicted in Figure 2. It is composed of three nodes: a wired server `S`, a wireless client `C` and an access point `AP` that acts as a router.

Nodes characteristics. All the three testbed nodes are running the Arch Linux distribution, equipped with the latest (at the time of writing) Linux kernel version 4.14.14-1. The end nodes `C` and `S` use CUBIC as their TCP congestion control algorithm. We tested three different wireless devices, a USB dongle and two types of PCIe cards. For all the device types, one device is used to create the Access Point router on the `AP` node and one other is used to provide wireless connectivity to `C`. The three setups are labelled by the WiFi device driver names as ath9k_htc, ath9k, and ath10k, and are summarised in Table I.

TABLE I: Testbed setups

setup name	chipset	protocol
ath9k_htc	Atheros AR9271 (1x1 MIMO)	802.11n
ath9k	Atheros AR9580 (3x3 MIMO)	802.11n
ath10k	Atheros QCA9880v2 (3x3 MIMO)	802.11ac

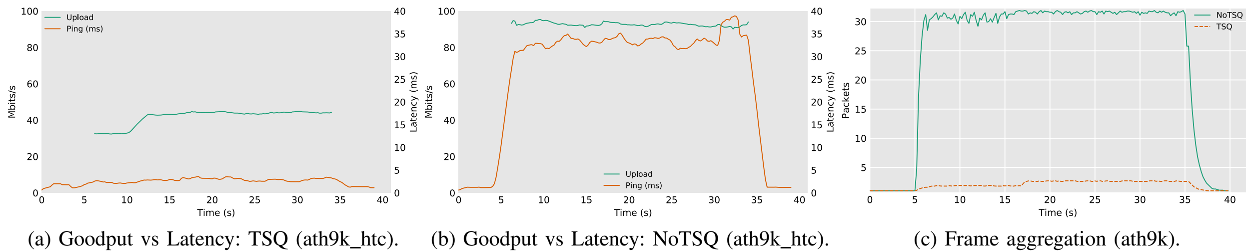
TABLE II: Parameters

parameter	value
Kernel version	4.14.14-1
TCP Congestion Control	Cubic
TSQ type	TSQ, NoTSQ and xTSQ
Queueing discipline	FQ_Codel
Tests	1-4 TCP Uploads, RRUL

The ath9k_htc setup uses USB devices which contain Qualcomm Atheros Communications AR9271 chipsets. The ath9k setup uses PCIe devices containing Qualcomm Atheros Communications AR9580 chipsets with 3x3 MIMO antennas. These first two setups are used for testing the IEEE 802.11n protocol. The ath10k setup uses PCIe devices containing Qualcomm Atheros Communications QCA9880v2 chipsets with 3x3 MIMO antennas. This latter setup is used for testing the IEEE 802.11ac protocol. These three setups, due to the different drivers, have different queueing systems. The ath9k_htc driver uses the standard qdisc-based queueing layer. The qdisc is configured to use FQ-CoDel as the queueing discipline, which is the default in Arch Linux and it is also the best option for preventing bufferbloat [17]. The ath9k and ath10k drivers instead use the integrated WiFi-specific queueing layer mentioned previously. The experiments on the ath9k and ath10k setups were carried out at Karlstad University, in a setup similar to the one shown in Figure 2, except that tests were run between `AP` and `C`.

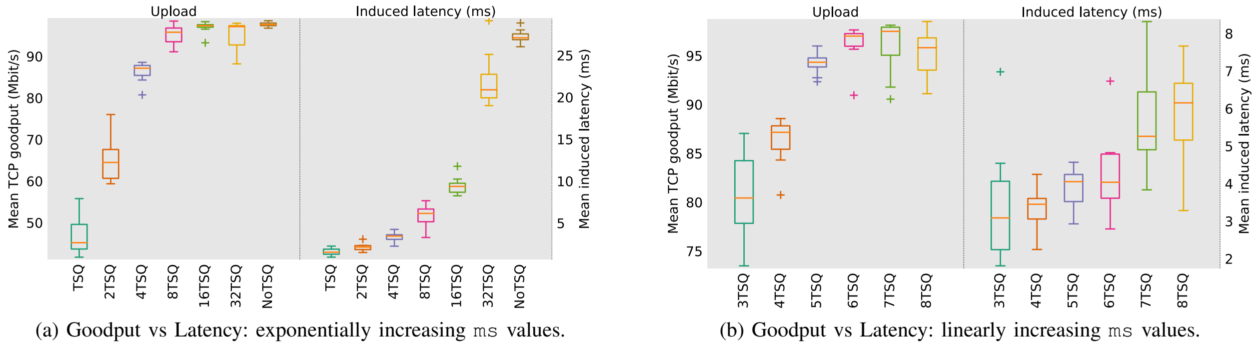
TSQ is the main variable parameter for our tests and can have different values:

- TSQ: the standard TSQ behavior that is obtained by setting `bytes`=128KB, `ms`=1 and `pkts`=0;
- NoTSQ: that completely disable TSQ logic and is obtained by setting `bytes`=-1;
- xTSQ: where `x` is the value, expressed in milliseconds, of the amount of data to enqueue for each TCP flow. It is obtained by setting `bytes`=10MB, `ms`=`x` and `pkts`=1. This expresses the tunable nature of the CoTSQ patch. We have set up a lower bound of one packet and an upper bound of 10MB of data, while the amount `x` rules the CoTSQ behavior by determining the amount of data to enqueue as a function of the TCP rate, autotuning the threshold like the original TSQ implementation. These bounds have been selected simply to make sure only the `ms` value controls the queue.



(a) Goodput vs Latency: TSQ (ath9k_htc). (b) Goodput vs Latency: NoTSQ (ath9k_htc). (c) Frame aggregation (ath9k).

Fig. 3: Goodput Vs latency and frame aggregation, 1 TCP upload on ath9k_htc and ath9k: TSQ vs NoTSQ.



(a) Goodput vs Latency: exponentially increasing `ms` values. (b) Goodput vs Latency: linearly increasing `ms` values.

Fig. 4: Goodput Vs Latency during a TCP flow upload on ath9k_htc.

Test configuration. We performed our tests using Flent [18], a flexible network tester able to configure different kinds of traffic and collect the results. We performed tests with a variable number of flows: starting from a single flow in upload, then moving to a more congested network with up to four flows in upload. Each test consists of one minute of actual TCP flow; a 5-seconds period of only latency measurement is included before and after each test to better show the impact of TCP on the queueing delay. We also run a test named Real-time Response Under Load (RRUL) designed by the bufferbloat community for analyzing network performance under heavy workloads; it consists in 8 TCP flows, 4 in download and 4 in upload, that compete with ICMP and UDP traffic. The RRUL test has been included in order to measure the impact on bidirectional traffic. All the testbed characteristics are summarised in Table II.

V. Results

In this section, we analyse the results obtained by running the aforementioned tests on the physical testbed. Only a subset of the results are presented here, but the full dataset is available together with the patch itself, and the test scripts, in [16]. Where not explicitly stated otherwise, the results are from the ath9k_htc setup.

The frame aggregation problem. We first present Figure 3 that encapsulates, in a sense, the motivation of this paper. The Figure shows the goodput achieved by a single TCP flow in upload (from the client to the server) competing with a ping flow. Figure 3a, shows the default configuration with TSQ enabled, while Figure 3b shows the result with the TSQ logic completely disabled (labeled NoTSQ).

This Figure has several interesting features:

- The default configuration with TSQ enabled gives a very low latency that oscillates between 1 and 4 ms. At the same time, the goodput is firmly bounded at 44Mbit/s;
- The patched configuration with NoTSQ (TSQ disabled) results clearly in a higher goodput that oscillates between 90 and 95 Mbit/s, which is the channel bandwidth limit with the equipped 802.11n hardware of the ath9k_htc setup. At the same time, the latency is increased to around 35ms.

The goodput/latency tradeoff between TSQ and NoTSQ is evident from this test, as well as the TSQ limitation in channel exploitation over an 802.11n environment. The reason why TSQ does not provide optimal goodput lies in the achieved frame aggregation, which is shown in Figure 3c; the presence of a limited number of packets queued in the driver limits the size of the aggregates that can be built, which severely limits the achievable throughput. On the other hand, disabling TSQ (with the NoTSQ test) lets the system reach the maximum goodput, better utilising the channel, but inducing a latency that reaches 35ms, ten times higher than with TSQ enabled. The goodput limit, in this case, strongly depends on the WLAN cards hardware.

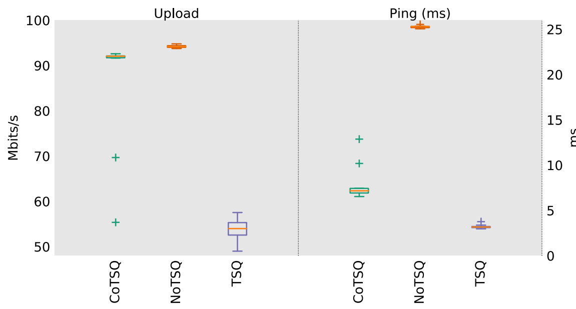


Fig. 5: Goodput vs Latency, 4 TCP flows in upload.

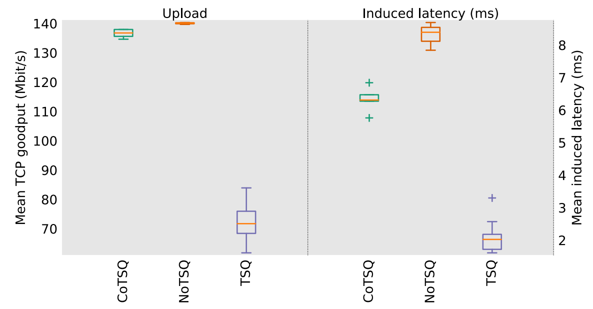


Fig. 7: Goodput vs Latency, 1 TCP upload: ath9k setup.

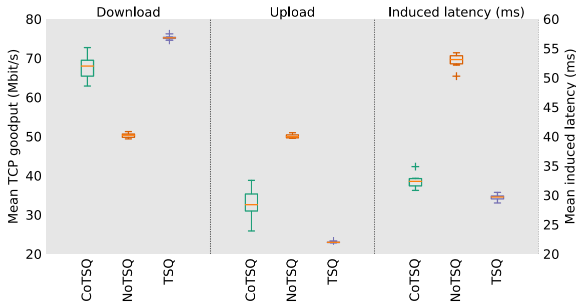


Fig. 6: Goodput vs Latency, RRUL test.

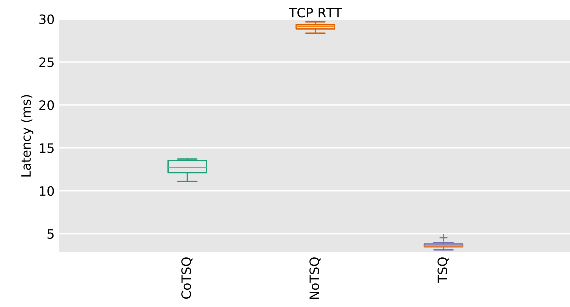


Fig. 8: TCP RTT, 1 TCP upload: ath9k setup.

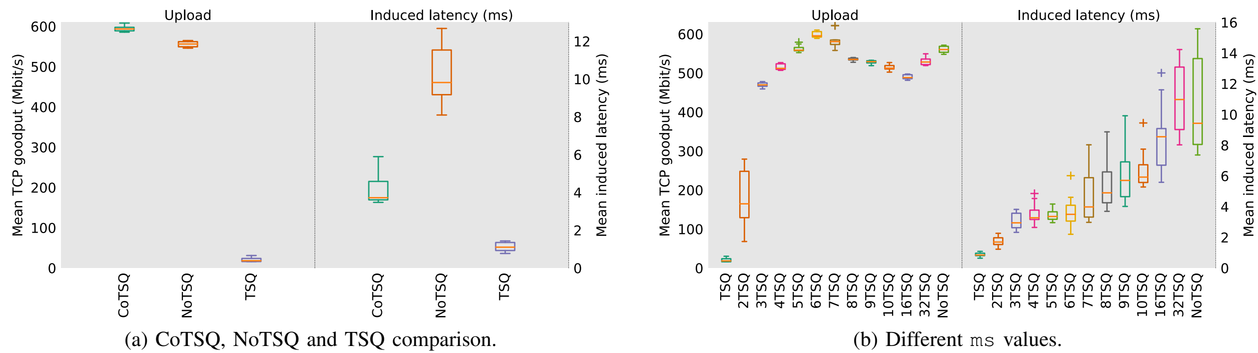
The natural question at this point is: is it possible to fully utilise the channel, reaching high goodput through frame aggregation while maintaining a low latency?

The CoTSQ solution. To explore the tradeoff between latency and throughput, we performed additional experiments while varying the `ms` parameter. Figure 4a shows the results of exponentially increasing the parameter value, as well as the effect of completely disabled TSQ. The figure shows that while the latency keeps increasing with the allowed queue space, throughput only increases up to the 8TSQ configuration. We then proceeded to a fine-grained evaluation of xTSQ switching to a linear increment of values in the range between three and eight `ms`. Figure 4b shows that the optimum tradeoff is found at 6 ms, which we will refer to as CoTSQ for the rest of this paper. Here, throughput reaches the maximum value while maintaining a latency only 2ms higher than the standard TSQ.

To examine the impact of CoTSQ in a scenario with more congestion, we perform an additional evaluation in a scenario where 4 simultaneous TCP uploads are active. Figure 5 shows the combined goodput of the four flows, and the latency measured while they were active. The first thing to notice is that the trend is maintained, CoTSQ and NoTSQ almost double the goodput also in a more congested environment with four transmitting flows. The second important thing to consider is that the latency induced by the CoTSQ solution is not increased by the presence of more TCP flows, and remains at 6 ms.

What happens in the presence of bidirectional traffic is shown by the RRUL test. In this case, TSQ only impacts the upstream traffic. This is because the client TSQ logic does not impact sending of ACKs from `C`, and because the bottleneck is at the `AP`, so there is no queue at `S`, making TSQ irrelevant. Figure 6 shows the results of the RRUL test run, and includes the goodput value of both download and upload streams, together with the induced latency. A very first thing to notice is the difference between the goodput in the download and upload directions. When standard TSQ is in place, the download is unfairly favoured; the difference is a factor of 3 with more than 75 Mbit/s for the download and less than 25 Mbit/s for the upload. The reason for such unfairness is that while the upload traffic is limited by TSQ, the download traffic is not, and so has no problem reaching its maximum achievable throughput. CoTSQ reduces the difference between upload and download, while keeping latency close to the value achieved by TSQ. While NoTSQ achieves perfect fairness between upload and download, this comes at the cost of almost twice the latency compared to TSQ and CoTSQ.

To show the impact of CoTSQ with a different queueing structure, Figure 7 shows the goodput of a single TCP upload together with the induced latency on the ath9k configuration. The first difference to notice is that the ath9k hardware achieves a higher maximum throughput due to its 3x3 MIMO setup. However, the same relative impact of TSQ remains, with CoTSQ doubling the achievable throughput relative to TSQ. As before, the latency increases to 6 ms with CoTSQ. The second difference to notice with Figure 5, is that the mean induced latency on NoTSQ is reduced from almost



(a) CoTSQ, NoTSQ and TSQ comparison.

(b) Different ms values.

Fig. 9: Goodput Vs Latency achieved during 1 TCP upload: ath10k setup

25 ms in the ath9k_htc setup, to less than 10 ms in the ath9k setup. This is because the WiFi-specific queueing structure used by the ath9k driver more effectively isolates the latency measurement (ping) flow from the queue induced by the TCP flow. To show the benefit of CoTSQ compared to NoTSQ in this setup, we also show the RTT of the TCP flow itself during the same test, in Figure 8. From this it is clear that CoTSQ helps the TCP sender to maintain a lower RTT while still reaching almost optimal goodput; the difference between the CoTSQ RTT and the NoTSQ RTT is nearly 20 ms.

802.11ac. Figure 9a shows the goodput of a single TCP upload together with the induced latency for the ath10k setup. This clearly shows the increased maximum bandwidth of 802.11ac compared to 802.11n. The remarkable thing to notice is that the goodput improvement from TSQ to CoTSQ in this setup is even larger, showing an order of magnitude improvement. Figure 9b shows the repeat of the experiment exploring different parameter values for TSQ. From this, it is clear the optimal value of `ms` is still 6 ms of data to enqueue. However, increasing the `ms` beyond this does not lead to a stable maximum goodput. We attribute this to interactions between queueing in the driver and queueing in the firmware, and plan to investigate this further in the future.

To conclude this section, we have demonstrated with CoTSQ that it is possible to achieve high TCP goodput over wireless 802.11 networks with a negligible latency induction. Based on our results, we have submitted a patch to the Linux kernel which modifies the TSQ parameters for all TCP flows going out over wireless interfaces. The patch has been accepted for inclusion and will be part of the upcoming version 4.17 of Linux.

VI. Conclusions

We have designed and implemented CoTSQ, a solution that modifies the TSQ logic of the Linux kernel to allow each flow to enqueue enough data to avoid breaking the frame aggregation logic of wireless network interface cards. We have tested our solution in both 802.11n and 802.11ac networks, with different levels of traffic congestion. The results show that TSQ prevents full channel utilisation in wireless networks due to the inability to aggregate packets. Our CoTSQ solution doubles the achieves goodput in 802.11n networks, and improves it by an order of magnitude in 802.11ac networks, in both cases with a negligible increase in latency. Future work will investigate the benefit of CoTSQ with a wider range of different wireless drivers as well as with different TCP congestion control algorithms.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *Queue*, vol. 9, no. 11, p. 40, 2011.

[2] Y. Gong *et al.*, "Fighting the Bufferbloat," in *Computer Comm. Workshops, 2013 IEEE Conference on*, April 2013, pp. 411–416.

[3] K. Nichols and V. Jacobson, "Controlling Queue Delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[4] R. Pan *et al.*, "PIE: A lightweight control scheme to address the Bufferbloat problem," in *HPSR, 2013 IEEE International Conference on*. IEEE, 2013, pp. 148–155.

[5] N. Cardwell *et al.*, "BBR: Congestion-based Congestion Control," *Comm. of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[6] A. Saeed *et al.*, "Carousel: Scalable traffic shaping at end hosts," 2017, pp. 404–417.

[7] B. Briscoe *et al.*, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 3, pp. 2149–2196, 2016.

[8] R. Lübben *et al.*, "On characteristic features of the application level delay distribution of TCP congestion avoidance," 2016.

[9] Y. Guo *et al.*, "Understanding on-device Bufferbloat for cellular upload," vol. 14-16-November-2016, 2016, pp. 303–317.

[10] J. Corbet, "TCP Small Queues," https://lwn.net/Articles/507065/, July 2012.

[11] T. Høiland-Jørgensen *et al.*, "FlowQueue-CoDel," https://tools.ietf.org/html/rfc8290, January 2018.

[12] T. Høiland-Jørgensen *et al.*, "Ending the anomaly: Achieving low latency and airtime fairness in wifi," in *USENIX ATC 17*, 2017.

[13] J. Corbet, "Network transmit queue limits," LWN Article, August 2011. [Online]. Available: https://lwn.net/Articles/454390/

[14] J. Saldana *et al.*, "Frame aggregation in central controlled 802.11 WLANs: The latency versus throughput tradeoff," *IEEE Communications Letters*, vol. 21, no. 11, pp. 2500–2503, 2017.

[15] Y. Kim *et al.*, "Throughput enhancement of IEEE 802.11 WLAN via frame aggregation," vol. 60, no. 4, 2004, pp. 3030–3034.

[16] "CoTSQ: Linux Kernel patch and source scripts," http://netlab.ing.unimo.it/sw/cotsq.zip, March 2018.

[17] T. Høiland-Jørgensen *et al.*, "The Good, the Bad and the WiFi: Modern AQMs in a residential setting," *Computer Networks*, vol. 89, pp. 90 – 106, 2015.

[18] ——, "Flent: The FLExible Network Tester," *ValueTools 2017*, 2017.