Bufferbloat mitigation in the WiFi stack - status and next steps

Toke Høiland-Jørgensen
Karlstad University
toke.hoiland-jorgensen@kau.se

Abstract

Over the last year, patches for bufferbloat mitigation and airtime fairness have landed in the WiFi stack. The bufferbloat patches improve performance across the board, and airtime fairness further improves performance in the presence of slow stations.

We provide an overview of these improvements and explain some of the peculiarities of WiFi that guided the design. Following this, we outline some of the proposed next steps. This includes proposals for further reducing latency, a mechanism to set different policies for the airtime usage of stations, as well as a revamped queueing mechanism for dynamic QoS priority scheduling.

1 Introduction

Linux has been the primary target platform for community efforts to mitigate the unwanted network queueing latency known as bufferbloat. Numerous improvements have gone into the Linux networking stack, such as the FQ-CoDel qdisc, first introduced in Linux 3.5. However, the effects of applying these improvements to a WiFi link has been limited until recently. This has changed dramatically over the last year, during which patches for bufferbloat mitigation and airtime fairness have landed in the WiFi stack. The bufferbloat mitigation patches reduce latency under load by an order of magnitude across the board, and an airtime fairness scheduler significantly improves network performance in the presence of slow stations. These changes are integrated directly into the mac80211 subsystem and disable the qdisc layer entirely on WiFi interfaces, to be able to effectively deal with the constraints imposed by the 802.11 MAC protocol.

In this paper we outline the properties of WiFi that have served as the basis for this design, and also describe the new queueing structure itself and the performance benefits resulting from it. In addition, we describe some of the current and planned features that are made possible because of the queueing structure. These include proposals for further reducing queueing latency, as well as a mechanism to set different policies for the airtime usage of stations, and a revamped queueing mechanism for dynamic QoS priority scheduling.

The work presented here is the result of a community effort, and I can by no means take credit for all of it. This exposition is meant as a summary of work that has mostly been published elsewhere, most notably in our previous paper [2], and on the linux-wireless and make-wifi-fast mailing lists.

An important purpose of this paper and its associated talk is to solicit feedback from the community on the future directions. This also means that several of the ideas for future work are very much preliminary and subject to change.

The rest of this paper first summarises some of the constraints imposed by the 802.11 protocol, then outlines the previous improvements and their benefits, and finally outlines some ideas for future improvements.

2 802.11 MAC protocol constraints

There are three main constraints to take into account when designing a queueing scheme for WiFi. First, we must be able to handle aggregation; in 802.11, packets are assigned different Traffic Identifiers (TIDs) (typically based on their DiffServ markings [5]), and the standard specifies that aggregation be performed on a per-TID basis. Second, we must have enough data processed and ready to go when the hardware wins a transmit opportunity; there is not enough time to do a lot of processing at that time. Third, we must be able to handle packets that are re-injected from the hardware after a failed transmission; these must be re-transmitted ahead of other queued packets, as transmission can otherwise stall due to a full Block Acknowledgement Window.

The need to support aggregation, in particular, has influenced the new queueing design in mac80211. A generic packet queueing mechanism, such as that in the qdisc layer, does not have the protocol-specific knowledge to support the splitting of packets into separate queues, as is required for aggregation. And introducing an API to communicate this knowledge to the qdisc layer would impose a large complexity cost on this layer, to the detriment of network interfaces that do not have the protocol-specific requirements. So rather than modifying the qdisc layer, our queue management scheme bypasses it completely, and instead incorporates the smart queue management directly into mac80211. The main drawback of doing this is, of course, a loss of flexibility. With this design, there is no longer a way to turn off the smart queue management completely; and it does add some overhead to the packet processing. However, the performance benefits by far outweigh the costs.

Algorithm 1 802.11 queue management algorithm - enqueue.

```
1:  function ENQUEUE(pkt, tid)
2:      if queue_limit_reached() then                    ▷ Global limit
3:          drop_queue ← FIND_LONGEST_QUEUE()
4:          DROP(drop_queue.head_pkt)
5:      queue ← HASH(pkt)
6:      if queue.tid ≠ NULL and queue.tid ≠ tid then
7:          queue ← tid.overflow_queue                   ▷ Hash collision
8:      queue.tid ← tid
9:      TIMESTAMP(pkt)                    ▷ Used by CoDel at dequeue
10:     APPEND(pkt, queue)
11:     if queue is not active then
12:         LIST_ADD(queue, tid.new_queues)
```

3 New queueing structure

This section is based on the description and evaluation in our previous paper [2] describing these changes. Please see that for further details and a more thorough evaluation.

The new WiFi queue management scheme is built on the principles of the FQ-CoDel qdisc. Because FQ-CoDel allocates a number of sub-queues that are used for per-flow scheduling, simply assigning a full instance of FQ-CoDel to each TID is impractical. Instead, we modify FQ-CoDel to make it operate on a fixed total number of queues, and group queues based on which TID they are associated with. When a packet is hashed and assigned to a queue, that queue is in turn assigned to the TID the packet is destined for. In case that queue is already active and assigned to another TID (which means that a hash collision has occurred), the packet is instead queued to a TID-specific overflow queue.[1]

The mac80211 stack does a lot of per-packet processing specific to the 802.11 protocol, which involves building headers, assigning sequence numbers, optional fragmentation, encryption, etc. Some of these operations are sensitive to reordering (notably, if sequence numbers or encryption IVs are out of order, packets will be discarded by the receiver). Since per-flow queueing can cause reordering when more than one flow is active, doing this packet processing before the queueing step would result in packet loss. On the other hand, applying everything at dequeue takes up valuable time in the loop that builds aggregates, which is time sensitive. To strike a balance between these factors, the mac80211 transmission handling is split into two parts, with the latter part (containing all reorder-sensitive actions) applied after the dequeue step, just before the packet is handed to the driver.

A global queue size limit is kept, and when this is exceeded, packets are dropped from the globally longest queue, which prevents a single flow from locking out other flows on overload. The enqueue logic is shown in Algorithm 1.

The lists of active queues are kept in a per-TID structure, and when a TID needs to dequeue a packet, the FQ-CoDel scheduler is applied to the TID-specific lists of active queues. This is shown in Algorithm 2.

[1] A hash collision can of course also mean that two flows assigned to the same TID end up in the same queue. In this case, no special handling is needed, and the two flows will simply share a queue like in any other hash-based fairness queueing scheme.

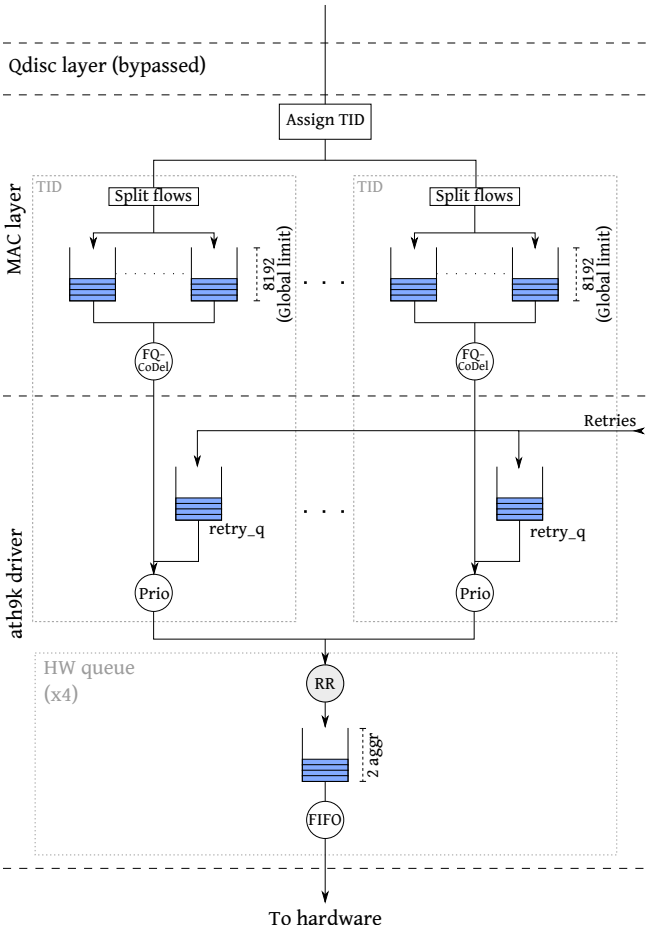

Figure 1: Our 802.11-specific queueing structure, as it looks when applied to the Linux WiFi stack.

The obvious way to handle the two other constraints mentioned above (keeping the hardware busy, and handling retries), is, respectively, to add a small queue of pre-processed aggregates, and to add a separate priority queue for packets that need to be retried. And indeed, this is how the ath9k driver already handled these issues, so we simply keep this. The resulting queueing structure is depicted in Figure 1.

3.1 Airtime fairness

Once we have the managed per-TID queues, it becomes straight forward to solve another long-standing WiFi performance problem: The so-called ”performance anomaly”. This anomaly is a well-known property of WiFi networks: if devices on the network operate at different rates, the MAC protocol will ensure *throughput fairness* between them, meaning that all stations will effectively transmit at the lowest rate. The anomaly was first described in 2003, and several mitigation strategies have been proposed in the literature (e.g., [3, 6]). However, none of these have been implemented in Linux until now.

Because we have a per-TID managed queueing struc-

Algorithm 2 802.11 queue management algorithm - dequeue.

```
1:  function DEQUEUE(tid)
2:      if tid.new_queues is non-empty then
3:          queue ← LIST_FIRST(tid.new_queues)
4:      else if tid.old_queues is non-empty then
5:          queue ← LIST_FIRST(tid.old_queues)
6:      else
7:          return NULL
8:      if queue.deficit ≤ 0 then
9:          queue.deficit ← queue.deficit + quantum
10:         LIST_MOVE(queue, tid.old_queues)
11:         restart
12:     pkt ← CODEL_DEQUEUE(queue)
13:     if pkt is NULL then                          ▷ queue empty
14:         if queue ∈ tid.new_queues then
15:             LIST_MOVE(queue, tid.old_queues)
16:         else
17:             LIST_DEL(queue)
18:             queue.tid ← NULL
19:         restart
20:     queue.deficit ← queue.deficit − pkt.length
21:     return pkt
```

Algorithm 3 Airtime fairness scheduler. The schedule function is called on packet arrival and on transmission completion.

```
1:  function SCHEDULE
2:      while hardware queue is not full do
3:          if new_stations is non-empty then
4:              station ← LIST_FIRST(new_stations)
5:          else if old_stations is non-empty then
6:              station ← LIST_FIRST(old_stations)
7:          else
8:              return
9:          deficit ← station.deficit[pkt.qoslvl]
10:         if deficit ≤ 0 then
11:             station.deficit[pkt.qoslvl] ← deficit + quantum
12:             LIST_MOVE(station, old_stations)
13:             restart
14:         if station's queue is empty then
15:             if station ∈ new_stations then
16:                 LIST_MOVE(station, old_stations)
17:             else
18:                 LIST_DEL(station)
19:             restart
20:         BUILD_AGGREGATE(station)
```

ture, solving the performance anomaly is simply a matter of scheduling transmissions from each of the TIDs in a manner that ensures that each station gets its fair share of airtime. The managed queues will then automatically provide the needed back-pressure on flows to each station to ensure latency stays low. The implementation in the ath9k driver adds accounting of the airtime used by each station, using a time deficit counter that is analogous to the FQ-CoDel byte deficit assigned to each flow. The airtime is reported by the hardware on TX completion, and on the receive side it is calculated from the packet size and the rate it was transmitted at.

Once this deficit is in place, an airtime fairness scheduler can be added to the driver. This is shown in Algorithm 3. It is similar to the the FQ-CoDel dequeue algorithm, with stations taking the place of flows, and the deficit being accounted in microseconds instead of bytes. The two main differences are (1) that the scheduler function loops until the hardware queue becomes full (at two queued aggregates), rather than just pulling a single packet from the queue; and (2) that when a station is chosen to be scheduled, it gets to build a full aggregate rather than a single packet.

3.2 Performance

We evaluate our modifications in a testbed setup consisting of five PCs: Three wireless clients, an access point, and a server located one Gigabit Ethernet hop from the access point, which serves as source and sink for the test flows. The evaluation is repeated for these four scenarios:

FIFO: The 4.6 kernel from kernel.org modified only to collect the airtime used by stations, running with the default `pfifo_fast` qdisc installed on the wireless interface.

FQ-CoDel: As above, but using the FQ-CoDel qdisc on the wireless interface.

FQ-MAC: Kernel patched to include the FQ-CoDel based intermediate queues in the MAC layer.

Airtime fair FQ: As FQ-MAC, but additionally including our airtime fairness scheduler in the ath9k driver.

We show only three sets of results here: The latency improvements from the queue management changes, and the achieved fairness and gain in total network throughput from the airtime fairness scheduler. For the full evaluation, see the previously published paper [2].

Figure 2 shows the results of our ICMP latency measurements with simultaneous TCP download traffic. Here, the FIFO case shows several hundred milliseconds of latency when the link is saturated by a TCP download. FQ-CoDel alleviates this somewhat, but the slow station still sees latencies of more than 200 ms in the median, and the fast stations around 35 ms. With FQ-MAC, this is reduced so that the slow station now has the same median latency as the fast one does in the FQ-CoDel case, while the fast stations get their latency reduced by another 45%. The airtime scheduler doesn't improve further upon this (other than to alter the shape of the distribution slightly for the slow station, but retaining the same median), so we have omitted it from the figure.

Figure 3 shows the airtime usage of each of the three stations in the different scenarios. When the airtime scheduler is enabled, perfect fairness is achieved. Figure 4 shows how this translates to throughput for downstream TCP traffic. For this case, the fast stations increase their throughput as fairness goes up, and the slow station decreases its throughput. The total effect is a net increase in throughput. The increase from the FIFO case to FQ-CoDel and FQ-MAC is due to better aggregation for the fast stations. This was observed for UDP as well in the case of FQ-MAC, but for FQ-CoDel the slow station would occupy all the queue space in the driver, preventing the fast station from achieving full aggregation. With the TCP feedback loop in place, this lock-out behaviour is lessened, and so fast stations increase their throughput.

As these results clearly show, there are significant performance gains from the changes already included in the kernel.

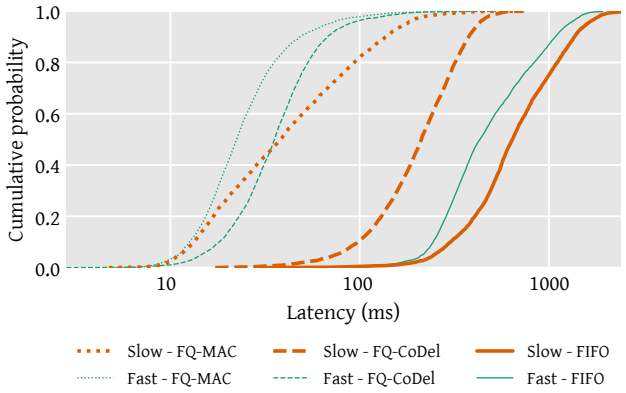

Figure 2: Latency (ICMP ping) with simultaneous TCP download traffic.

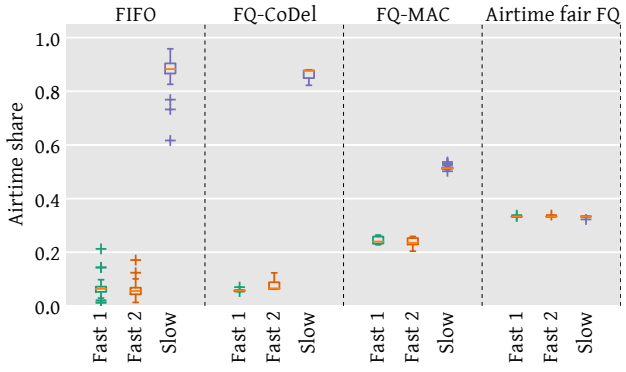

Figure 3: Airtime usage for each of the three stations in the different scenarios (UDP traffic).

4 Next steps

Currently, only the ath9k and ath10k drivers (and the out-of-tree mt76 driver) benefit from the intermediate queue structure, and the airtime fairness scheduler is ath9k only. There is work underway to remedy this; patches exist to move the airtime scheduling into mac80211,[2] and Johannes Berg has outlined a plan for converting the mac80211 code completely to using the intermediate queueing structure,[3] which will allow all drivers to benefit from it.

Integration into the common mac80211 layer is important to ensure that these features, and further developments based upon them, can benefit all drivers with as little effort as possible. The rest of this section outlines some of the ideas for future improvements that I and others in the community have discussed. These are all mostly on the idea stage for the time being, and part of the purpose of listing them here is to gather feedback on their usefulness and viability.

The upcoming improvements are grouped into three parts: Further reducing latency, Airtime policies and QoS handling.

[2] https://lkml.kernel.org/r/20171016160902.8970-1-toke@toke.dk

[3] https://lkml.kernel.org/r/1507217947.2387.60.camel@sipsolutions.net

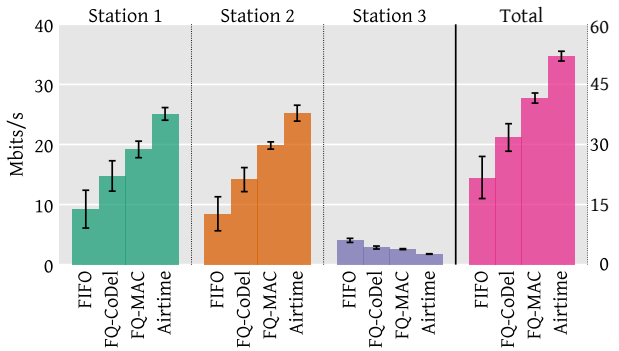

Figure 4: Throughput for TCP download traffic (to clients).

4.1 Further reducing latency

While we have reduced queueing latency by an order of magnitude, there are still some areas where additional reductions should be feasible. This can be seen by the fact that it is still possible to lower latency by aggressively shaping the traffic on top of the wireless interface (thus moving the bottleneck somewhere else). While not all these ideas are applicable to all devices (because some of the required functionality may be in firmware), it can help where implementing it is possible. The areas of potential improvement are:

Minimising hardware queue length. Currently, most drivers keep a fixed amount of data buffered ready to send to the hardware. For drivers that handle aggregation in firmware, this is in the form of a fixed number of packets, while for drivers that build aggregates in software (mainly ath9k), this is a fixed number of full aggregates. In the former case, a BQL-like mechanism is needed to limit this buffering to a minimum. Drivers that do aggregation in software have enough information available that another approach may be viable: Deferring scheduling of the next aggregate until right before the expected completion time of the previous one. CPU scheduling granularity may be a limiting factor in this, and getting the timing right in general can be tricky. In both cases, getting an interrupt from the hardware when transmission **starts** would be helpful to achieve more precise timing.

Reducing the number of retransmissions. Instead of using a fixed number of maximum retransmissions for all packets, dynamically setting the number of retransmissions based on the data rate can reduce latency by reducing head of line blocking while the retransmissions are ongoing. This has been shown to work in a fairly simple implementation [4]. Using a time based limit and optionally including the queue time as recorded by CoDel could be a way to achieve this.

Dynamic aggregate sizing. Rather than always attempting to build the largest possible aggregates, when there are many stations with data outstanding it may be worthwhile to reduce the size of the aggregates (to, say, 1 ms) to trade off lower latency for airtime efficiency (and so bandwidth). Exploration of when this is a good tradeoff remains to be done.

4.2 Airtime policies

Having airtime accounting information available when scheduling stations makes it possible to implement other scheduling options than strict fairness. For instance, in some cases it may be desirable to allow a slow station to exceed its fair share of airtime in order to get it above a minimum bit rate. Or it may be desirable to limit a subset of active stations to a fraction of the available airtime (to implement a limited guest network, for instance).

There are multiple ways to extend the scheduler to support such use cases. The two most straight forward are the following:

Grouping queues and scheduling the groups. The current deficit-based scheduler simply goes round-robin between all active queues until it finds one that has a positive airtime deficit, which is then scheduled (while the others have their deficit increased by one quantum every time the scheduler passes over them). Introducing groups that are scheduled together (possibly recursively and possibly with weights) is a way to enable many different policies and allows configuration by letting userspace configure the groups. The drawback is that it requires a non-trivial configuration to achieve the desired results, especially as stations come and go.

Arbitrarily dividing airtime between stations. Different policies could be implemented by dividing credits unevenly, allowing userspace to specify an arbitrary division of credits. However, specifying a good API for this is difficult. Using BPF could be an option, but without being ale to loop over the active stations, it may not be possible to realise the arbitrary division of airtime that is needed for this to work.

Performance issues Adding features to the scheduler to make it support policies is bound to make necessary to do more work on each invocation of the scheduler, which can be problematic in the fast path. A way to fix this is to queue the airtime usage information on per-CPU queues in the fast path, and have a separate thread collect the information and perform scheduling. For this, it may be necessary to switch the scheduler to use airtime credits that are divided between stations, similar to [1], as that would simplify the decision making in the fast path.

4.3 QoS handling

At present, the four different 802.11e QoS levels are basically scheduled in strict priority order, mapping to QoS queues is done by diffserv marking and there is no admission control for the higher priority levels. At the same time, with the improved queueing structure, the end-to-end latency for a voice flow marked as best-effort can be as good as for one that uses the VO queue.

Improved network efficiency. There is a potential for improving network efficiency by including queued VO packets at the head of aggregates sent at a lower priority level (since the VO level doesn't allow aggregation). Using current queue occupancy as a factor in deciding when to do this is important, but information on the level of media contention is also needed, as the more aggressive media contention parameters of the VO priority is important in high-contention scenarios. More experimentation is needed to figure out exactly in which cases this optimisation makes sense.

Admission control. Another issue with QoS handling is that it is based on diffserv markings and that there is no admission control. This means that it is quite possible to send large data flows at the higher priority levels, which can hurt performance of the network. Previous work in the Cake scheduler[4] have shown promising results in applying a soft admission control, where flows that build a large queue at the higher priority levels are dynamically demoted to best effort (or even background). Compared to a strict rate-based admission control, this has the advantage that it adapts automatically to the current performance of the network, at the cost of less predictability for applications.

Interaction with airtime fairness. In some cases, a station with outstanding high priority packets outstanding needs to be throttled to achieve fairness, even though all other stations only have packets queued of a lower priority. In this case, QoS prioritisation and airtime fairness enforcement conflicts with each other: If fairness is enforced, lower priority packets will be transmitted before those of higher priority. In most cases it is likely beneficial to enforce fairness first (which can be achieved by first selecting the station to transmit to, and the picking the highest priority queue active for that station), but this may lead to problems with regulatory compliance. We plan to explore this further going forward.

4.4 Configurability and extraction of statistics

The queueing structure and airtime fairness scheduler as they are currently implemented offers very little in configurability and statistics on their operation. What little there is available is only accessible through the debugfs interface, which is often disabled on production systems. While a lot of emphasis has been put on making sure that these features work in the absence of operator tuning, in some cases tuning can still be necessary. Having tuning and statistics available in a way that integrates well with existing tooling is something that not a lot of thought has been put into as of now. The available options and statistics are as follows:

- Configuration (per phy):
  - FQ knobs (packet/memory limit, quantum)
  - Airtime flags (whether to count airtime on TX/RX)
- Statistics:
  - FQ per-tid stats (drops/marks/bytes etc; per sta)
  - FQ multicast stats (per vif)
  - Airtime stats (TX/RX usecs, deficit; per sta)

[4] `https://github.com/dtaht/sch_cake`

Going forward, these will all be incorporated into the nl80211 netlink interface (with the possible exception of the airtime flags, which is mostly a debug setting). For some things, this is fairly straight forward (for instance, airtime statistics can be straight-forwardly included along with the byte statistics already available in the mac80211 netlink interface and shown by the `iw` tool), while less so for others. As an examples the latter, a new API needs to be hashed out for the upcoming policy aspects of the airtime scheduler (which will require userspace support to set the policy).

5 Conclusion

This paper summarises the work that has gone into the WiFi stack over the last year to reduce bufferbloat and improve airtime fairness. These improvements have been realised by replacing the queueing structure and station scheduling of the mac80211 layer. We have also outlined some promising avenues for future work based on these new features, which we plan to pursue in the coming months.

6 Acknowledgements

Several people were indispensable in making this work happen. First and foremost my co-authors on the original paper, Michał Kazior, Dave Täht, Per Hurtig and Anna Brunstrom. Michał wrote most of the code for the mac80211 queueing structure, and the others were instrumental in helping with testing, debugging, experimental design and feedback. Johannes Berg, Kalle Valo and Felix Fietkau have offered excellent feedback on the code itself, and many members of the OpenWrt/LEDE and Make-wifi-fast communities provided invaluable feedback and testing. Finally, a general thank you to the whole netdev community for being welcoming and generally awesome, in virtual as well as in real life.

References

[1] Garroppo, R. G.; Giordano, S.; Lucetti, S.; and Tavanti, L. Providing air-time usage fairness in IEEE 802.11 networks with the deficit transmission time (DTT) scheduler. 13(4):481–495.

[2] Høiland-Jørgensen, T.; Kazior, M.; Täht, D.; Hurtig, P.; and Brunstrom, A. 2017. Ending the anomaly: Achieving low latency and airtime fairness in wifi. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 139–151. Santa Clara, CA: USENIX Association.

[3] Joshi, T.; Mukherjee, A.; Younghwan Yoo; and Agrawal, D. Airtime fairness for IEEE 802.11 multirate networks. 7(4):513–527.

[4] Nomoto, M.; Wu, C.; Ohzahata, S.; and Kato, T. 2017. Resolving bufferbloat in tcp communication over ieee 802.11 n wlan by reducing mac retransmission limit at low data rate. *ICN 2017* 80.

[5] Szigeti, T., and Baker, F. 2016. DiffServ to IEEE 802.11 Mapping. Internet Draft (standards track).

[6] Tan, G., and Guttag, J. V. Time-based fairness improves performance in multi-rate WLANs. In *USENIX Annual Technical Conference, General Track*, 269–282.