Airtime Fairness

Handling massive client download and fair airtime

Overview

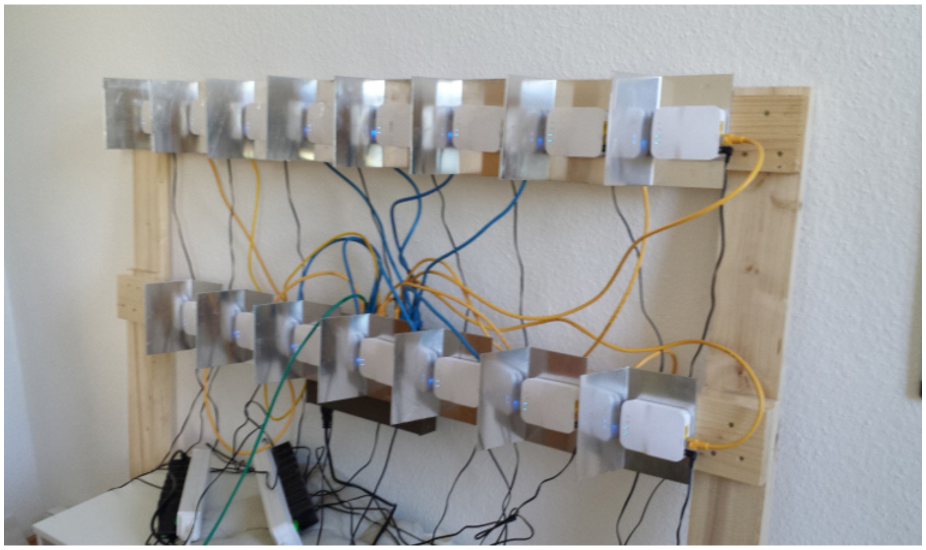
Many clients connected to one access point is a scenario often seen at conferences or airports. Throughput performance often drops and latencies go up when many clients use the WiFi simultaneously, but its hard to test these kind of scenarios in a lab to reproduce these problems. For this purpose we built a test wall of 30 WiFi clients (OM2P-LC access points) to simulate these kind of scenarios and tune WiFi drivers and schedulers. We would like to support a high number of simultaneous clients while maintaining good throughput, low/acceptable latencies and fair distribution of resources among stations.

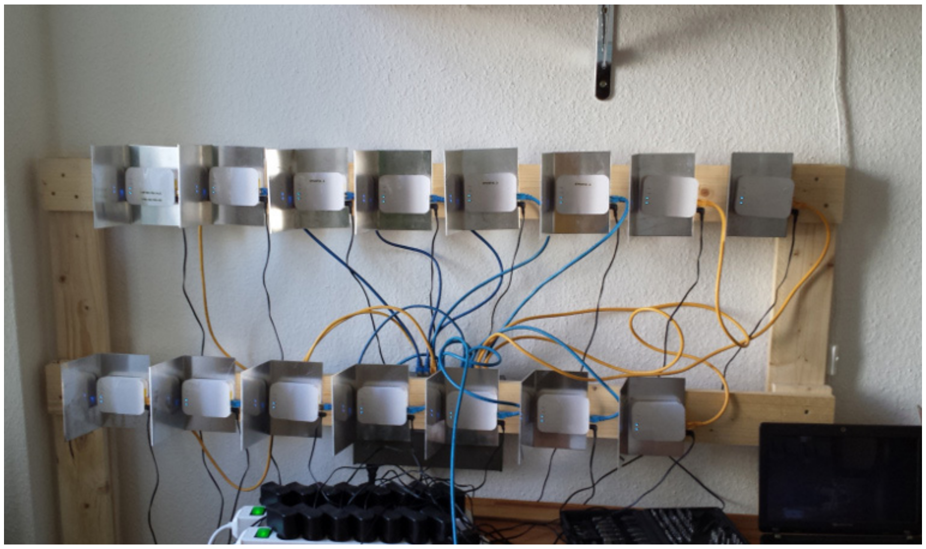
One approach to tackle the mentioned problems is to reduce bufferbloat by limiting the time packets stay in queues of the access point before sent to their destination. This will reduce latencies which are often seen to rise to multiple seconds in busy WiFi networks. The CoDel algorithm was implemented in the Linux kernel to mitigate this problem, which attempts to keep latencies below certain target values.

Another problem is that clients may be served at different WiFi rates (from 1 Mbit/s up to 300 Mbit/s or more), depending on their connection quality or signal strength. Frames on low datarates take much longer time in the air to transmit a fixed amount of data compared to frames at high datarates. A phenomena which can be observed quite often is that the “slow” clients operating with low datarates will hog most of the airtime, leaving the fast clients only a fraction of time to transmit data. However, if airtime is distributed equally among clients regardless of their datarates used, the “fast” clients can send much more data in their share of airtime and the throughput sum will increase. Aggregate throughput improvements of 200% and more can be achieved using this technique. We have implemented an experimental airtime-based scheduler in ath9k to balance airtime and improve the overall throughput.

Test setup

Our test setup consists of a two test walls with 30 OM2P-LC access points mounted on them.





All of them connect to one access point. Client number 1 (the “slow client”) is configured at a fixed rate of 1 Mbit/s (by using TKIP and limiting the legacy rates to 1 Mbit/s), while other clients may use any datarate (up to 72 Mbit/s at MCS7/HT20).

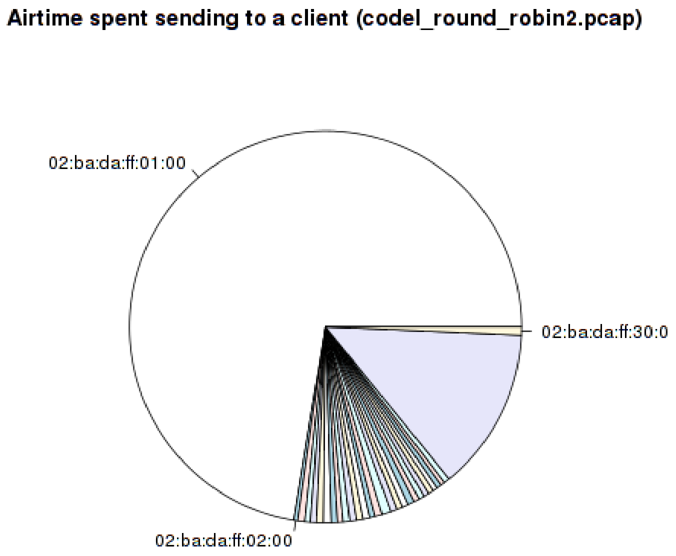
Clients run a netperf server, and our test script downloads data from a machine behind the access point over the WiFi to the clients. At the same time, the slow client and two fast clients measure the latency using ping.

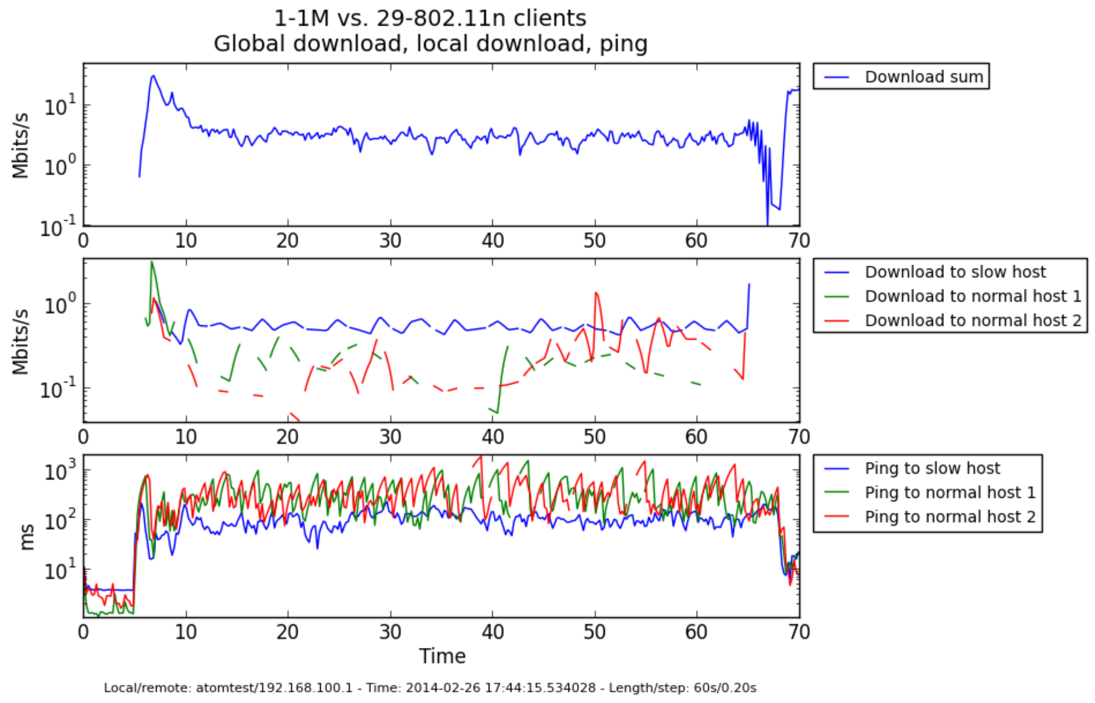
The AP runs OpenWRT trunk (r39639) with a few custom patches to re-enable pfifo_fast for comparison and with the experimental fair airtime patch. ath9k schedulers are either round robin (the current default scheduler) and fair airtime. Either pfifo_fast or fq_codel qdiscs are used on the WiFi interface.

Test results

fq_codel + round robin

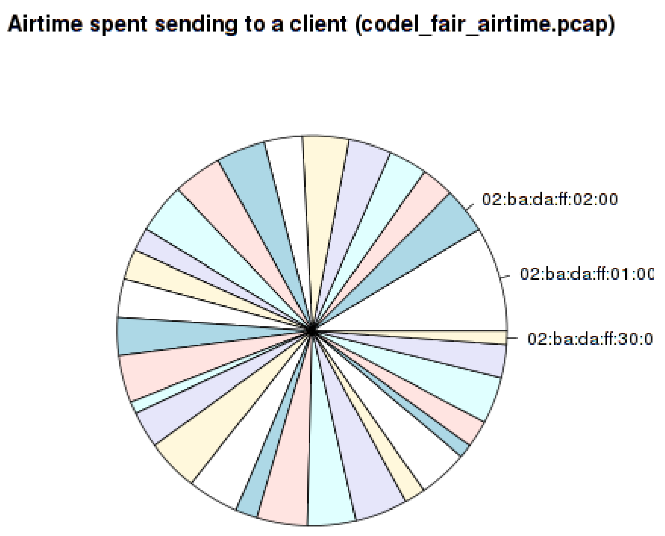
This is the current default configuration in OpenWRT. We can observe that the latency limits are not too bad, but the slow client is given way too much airtime compared to the other fast clients. This results in stable latency and throughput graphs for the slow client, but poor performance for the fast clients. Note that various tests have been performed and most don't look as stable as the one in the graphs below.

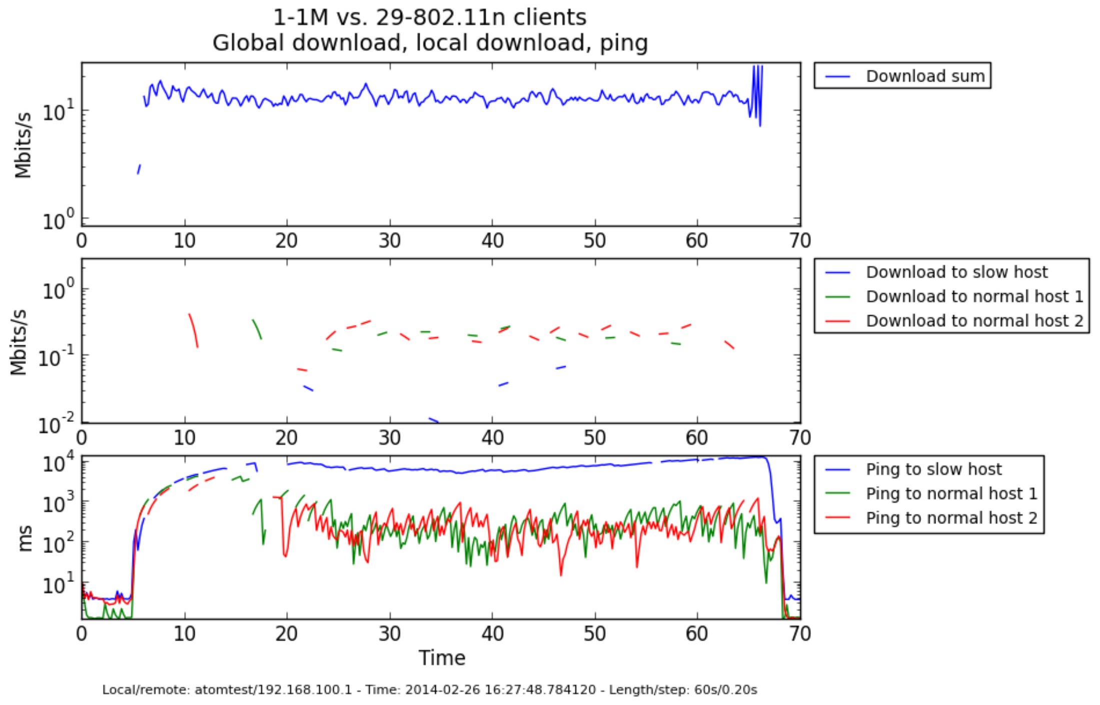




fq_codel + fair airtime

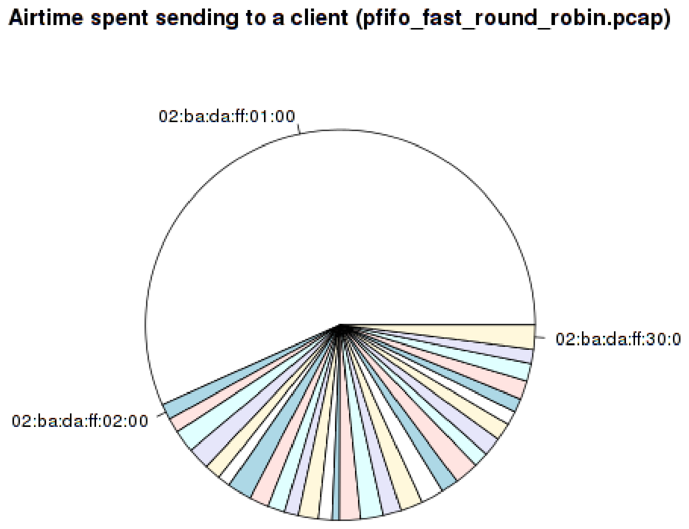
Compared to round robin, the airtime is much more equally distributed on the clients. The slow client is not favored, but its latency times are too high to be acceptable (around 1s). A lot of frames for this client are buffered for a long time in the ath9k-internal queues, and the client is not scheduled very often anymore due to its high airtime consumption. The aggregate bandwidth is much higher compared to round robin, as desired.

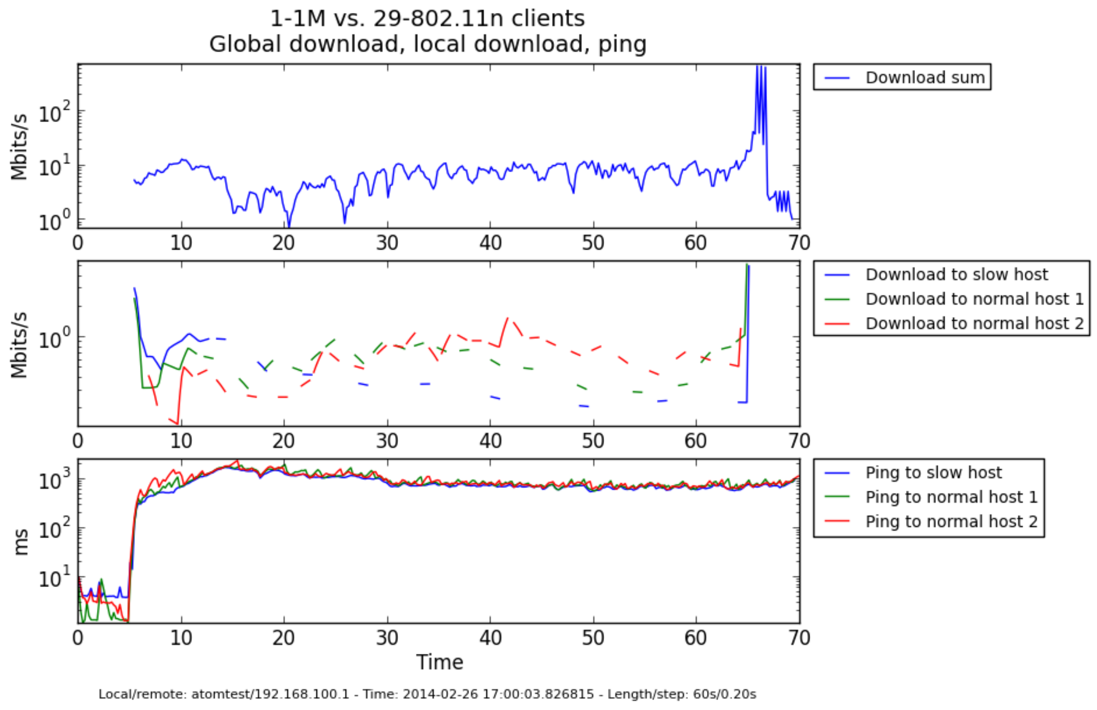




pfifo_fast + round robin

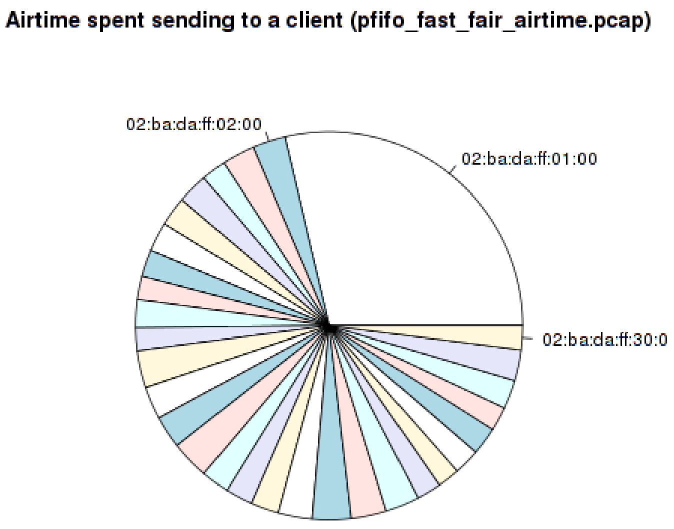
Throughputs are rather shaky in this test, and latencies are above acceptable levels. The airtime distribution is more fair compared to fq_codel + round robin.

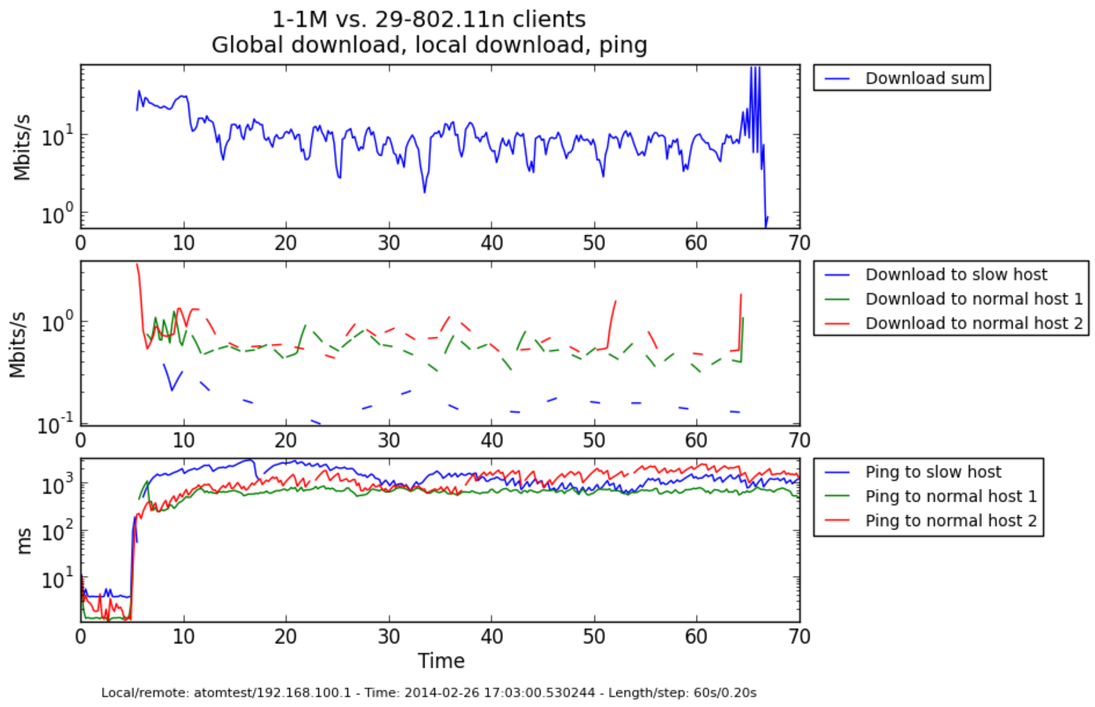




pfifo_fast + fair airtime

Throughputs are still shaky but higher than for the round robin scheduler. The airtime is distributed more fairly compared to round robin, but the slow client still hogs more than 25% of the total airtime. Latencies are still beyond acceptable levels.





Summary

The current fair airtime implementation holds its promise and distributes the airtime fairly among the clients. Also the aggregate throughput has increased, as expected. However, the slow client now has an unacceptable high latency - that is because packets for this client now stay in the ath9k-internal queues, out of reach for CoDel. Also not all clients are really fast - some of them don't get enough packets into ath9k to form large aggregates, and therefore only transmit few packets with lower datarates (but similar airtime, though).

It seems the key to solving that would be to improve the communication between the driver (ath9k) and the upper layers (mac80211 or qdiscs), so that the driver can report which stations could use more packets and the upper layers can deliver packets for exactly these stations. More on that in the next section.

On a side node, it is interesting to see how codel and the current scheduler surprisingly seem to favor the slow client instead of the fast one.

Next steps

One major problem observed while implementing the fair airtime scheduler is the supply of frames from higher levels (qdisc). ath9k keeps separate queues per station, but some are getting filled with frames while others stay empty. There is currently no way to signal upper levels which station has airtime credits left and could use more frames, and which are already full - the only signaling mechanism is to "wake" the upper layers to deliver more frames for the whole interface (which could be for any station) or to stop the delivery.

There are currently three ideas how to fix this problem:

1. Separate queues per station

Instead of having a single queue per WMM access class, split queues per station. That would allow separate qdiscs per station as well, which can be woken/stopped from the underlying driver directly. The separate queues could be either created dynamically[1], statically (the limit is 2007 stations, times 8 for each TID) or with a fixed amount of queues and a hash. Each of these ideas has drawbacks (too much memory waste, complicated queue handling) and all mentioned alternatives will probably require rather complex changes in mac80211 and drivers.

2. Inform qdiscs about delivery
qdiscs currently stop caring about frames after dequeueing them. They are not informed when the packet actually leaves the underlying driver and/or hardware. One way to solve this would be to add a callback to signal transmission completion[2]. For mac80211/ath9k and many soft mac drivers, skbs are returned to mac80211 after delivery to report the tx status (how many retries have been used on which rates, etc). Therefore a callback could be performed quite easily, however qdisc-internal state will be lost (skb->cb is overwritten by mac80211 and drivers).

3. Queue management in drivers
Drivers like ath9k which manage station queues separately could keep queues short by themselves, e.g. by dropping packets if they can't be delivered after some time. This scheme would be driver-dependent as qdisc code can't be reused at this point, and still does not interact properly with the "upper layer" qdiscs.

As for any of the ideas above, we'd be very interested in suggestions and comments.

External links:
[1] http://permalink.gmane.org/gmane.linux.kernel.wireless.general/80474
[2] http://permalink.gmane.org/gmane.linux.kernel.wireless.general/80476

For more information:

Marek Lindner	Simon Wunderlich
marek@open-mesh.com	simon@open-mesh.com