TCP over Large Buffers: When Adding Traffic Improves Latency

Tristan Braud, Martin Heusse, and Andrzej Duda
Grenoble Institute of Technology, CNRS Grenoble Informatics Laboratory UMR 5217, Grenoble, France
Email: [Tristan.Braud, Martin.Heusse, Andrzej.Duda]@imag.fr

***Abstract*—Excessive buffer sizes in access networks may result in long delays, the effect known under the name of *bufferbloat*. There exist mechanisms that can solve the problem, however their deployment may be difficult or not feasible. In this paper, we explore other approaches that may bring better performance if the known solutions cannot be applied. Our main finding is the paradoxical observation that adding low intensity traffic of short packets improves latency at the price of a slight decrease in throughput. The result comes from the fact that buffer sizes are generally limited to a number of packets, so that filling them with small packets improves the queueing delay. Such a countermeasure does not require any modification at the head of the bottleneck link so an end host can use it to improve its performance. We also notice that the reverse download traffic has beneficial influence on performance, which explains why the negative effect of bufferbloat sometimes appears as alleviated. The paper presents the results of extensive experiments in a realistic setup over an ADSL link as well as on a dedicated testbed to illustrate the performance gains and the effect of several TCP variants.**

I. Introduction

Bufferbloat is a recently analyzed phenomenon in which large buffers at routers lead to long delays [1], [2]. Even if there is no empirical evidence on how often bufferbloat happens [3], it can have detrimental effect on user experience. Consider for instance a 400kb/s residential uplink with a 50 packet buffer at its head: sending 50 (1500B) packets takes 1.5s while the default timeout for TCP SYN or a DNS query is 1s. Thus, a host may retransmit packets even before its first packet leaves the buffer! Moreover, a large uplink buffer greatly affects download throughput due to the interference between uploads and downloads [4].

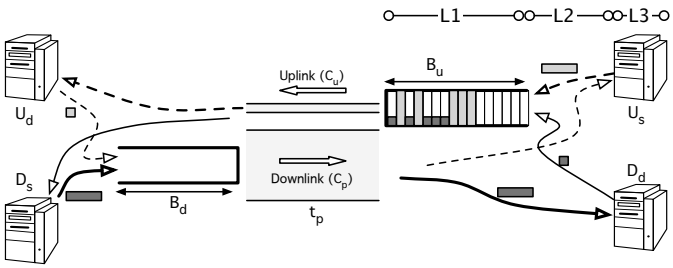

Figure 1. An excessive buffer size (B_u) at the uplink leads to high latency when the buffer holds large TCP data segments. Short (TCP ACKs) and large packets mix up in the buffer when there is traffic in both directions.

There are many approaches to solve the problem: the most effective one is to eliminate the problem by changing the queueing policy at the head of the uplink, which is the bottleneck in such a setup, *e.g.* by giving priority to short packets. This level of action is illustrated as L1 in Figure 1 (the figure presents the common case of an asymmetric residential DSL link with possibly two TCP connections in opposite directions: download and upload). As this solution is not always practical, it is possible to move the bottleneck closer to the upload source to a point where tweaking the queueing policy is easier (illustrated as L2 level in the figure). Such a solution requires to give up a bit of capacity and be able to estimate the bottleneck link capacity in real time. Then, by sending traffic into the uplink at a rate that remains below the capacity, the buffer does not fill up; the new bottleneck should have less buffer space or implement a more advanced queueing policy than FIFO. Even further upstream, one can act at the source host (L3) by avoiding to saturate the uplink with large packets. This paper focuses on this last level of action, because it is under the user control unlike the other solutions.

One possible solution at L3 is to use a congestion control algorithm such as TCP Vegas that will not use all buffer slots. Since Vegas keeps the queueing delay to a low level, it is not sensitive to the buffer size, as its approach is to maintain a low buffer occupancy while keeping the link active. Another approach is to make sure that a significant fraction of buffer slots contain small packets, which will decrease the queueing delay. Such small packets may appear as a side effect of other traffic, for instance as TCP ACKs generated by a download connection. This beneficial influence of ACKs from the download connection on delay may explain why bufferbloat is not necessarily as prevalent as sometimes thought [3]. Testing the network for excessive buffers shows that they are common, but in many cases, using the network lowers their impact.

In this paper, we propose to improve latency by adding small packets to the upload queue on purpose. Such a scheme may appear paradoxical, but we show through extensive analyses and experiments that the method results in a significant improvement of latency at the price of a slightly lower throughput.

II. Adding Traffic to Improve Latency

In most cases, buffers are structured as arrays of packets regardless of the packet size, *i.e.* there is a maximal number of packets in a buffer. So, the queueing delay greatly depends

978-0-9836283-9-2

on the size of the packets present in the buffer as illustrated in Figure 2.

Figure 2. More short packets in a buffer lead to a lower queueing delay.

A. Motivating Experiment

Consider a regular (BeWAN 700G) ADSL modem with an uplink buffer of 30 packets, 3815kb/s downstream and 854kb/s upstream bit rates at the ATM layer. Table I presents the measured performance in two cases.

Table I
MOTIVATING EXPERIMENT

	Without UDP	With UDP
Queueing delay	346±84ms	124±50ms
TCP goodput	728kbit/s	660 kbit/s
Single download TCP goodput	3.26Mb/s	
Download TCP goodput when contending with TCP upload	2.80Mb/s	3.06Mb/s

First, we measure the queueing delay with a single TCP upload. Sending UDP packets at a rate of 125 packets/s (1 byte UDP payload) makes the upload goodput marginally drop to 660kb/s, whereas the average queueing delay is more than halved to 124ms.

Second, we consider a long lasting TCP download flow that gets 3.26Mb/s by itself on the link. It obtains 2.80Mb/s against one upload and 3.06Mb/s when adding the UDP traffic.

We can observe that adding the stream of UDP packets lowers and stabilizes the delay without significantly altering throughput. It also alleviates the upload impact on the download performance. Both effects improve network responsiveness.

B. Assumptions and Notation

We consider that the uplink buffer is large enough so that the congestion control manages to keep the link always busy. We also consider buffer occupancy before the buffer overflows.

We assume the following:

- TCP connection of sending data rate D_{T}, packet size S_{T}, and packet rate $R_{\mathrm{T}} = \dfrac{D_{\mathrm{T}}}{S_{\mathrm{T}}}$,
- UDP stream of throughput D_{U}, packet size S_{U}, and constant packet rate $R_{\mathrm{U}} = \dfrac{D_{\mathrm{U}}}{S_{\mathrm{U}}}$,
- $S_{\mathrm{U}} \ll S_{\mathrm{T}}$ and $D_{\mathrm{U}} \ll D_{\mathrm{T}}$,
- R_{T} is constant over one RTT, which is a fair assumption in the congestion avoidance phase,
- C_u, uplink bottleneck capacity,
- $Q(t)$, buffer occupancy (queue) in packets at time t.

C. Single TCP Flow

The queue growth during Δt is the difference between packet arrivals and transmissions on the uplink:

$$\Delta Q_{\mathrm{T}} = R_{\mathrm{T}}\Delta t - \frac{C_u}{S_{\mathrm{T}}}\Delta t \tag{1}$$

D. TCP and UDP Flows

For the both flows, the similar expression is the following:

$$\Delta Q_{\mathrm{T,U}} = Q_{\mathrm{T,U}}(t+\Delta t) - Q_{\mathrm{T,U}}(t) = (R_{\mathrm{T}}+R_{\mathrm{U}})\Delta t - \frac{C_u}{S}\Delta t, \tag{2}$$

where $\frac{C_u}{S}$ is the packet departure rate from the queue and S is the average packet size:

$$S = \frac{R_{\mathrm{U}}S_{\mathrm{U}}}{R_{\mathrm{U}}+R_{\mathrm{T}}} + \frac{R_{\mathrm{T}}S_{\mathrm{T}}}{R_{\mathrm{U}}+R_{\mathrm{T}}} \tag{3}$$

As $D_{\mathrm{U}} = R_{\mathrm{U}}S_{\mathrm{U}} \ll D_{\mathrm{T}} = R_{\mathrm{T}}S_{\mathrm{T}}$, we have $S \approx \dfrac{D_{\mathrm{T}}}{R_{\mathrm{U}}+R_{\mathrm{T}}}$ and

$$\Delta Q_{\mathrm{T,U}} = (R_{\mathrm{U}}+R_{\mathrm{T}})\Delta t\ (1 - \frac{C_u}{D_{\mathrm{T}}}) \tag{4}$$

So,

$$\Delta Q_{\mathrm{T,U}} = \frac{(R_{\mathrm{U}}+R_{\mathrm{T}})}{R_{\mathrm{T}}}\Delta Q_{\mathrm{T}} \tag{5}$$

This expression indicates that with the additional UDP traffic, buffer occupancy increases by proportion $\frac{R_{\mathrm{U}}}{R_{\mathrm{T}}}$ compared to the single TCP case. So the buffer overflows earlier, the TCP congestion window grows to a lower value and queueing delay decreases accordingly. If we want to lower the queueing delay by factor α: $\frac{\Delta Q_{\mathrm{T,U}}}{\Delta Q_{\mathrm{T}}} = \alpha$, the UDP rate needs to be:

$$R_{\mathrm{U}} = R_{\mathrm{T}}(\alpha - 1), \tag{6}$$

knowing that in all cases, the TCP packet departure rate is $R_{\mathrm{T}} \approx \frac{C_u}{S_{\mathrm{T}}}$.

In the above calculations, we have neglected the link layer overhead. On an Ethernet link, for instance, the smallest frame size is 64B plus preamble and interframe spacing (20B), so the size ratio between 1500B TCP segments to the smallest UDP packet size is 18 to 1.

Otherwise, the smallest UDP packet carries 1B of payload, for a total of 29B with IP and UDP headers. So the size ratio is then 52 to 1, which allows to generate many UDP packets per TCP segment at a low overhead. From Eq. 6, the overhead of the additional UDP traffic is:

$$\mathrm{OH}_{\mathrm{U}} = \frac{1}{1 + \frac{S_{\mathrm{T}}}{S_{\mathrm{U}}}\frac{1}{\alpha-1}}. \tag{7}$$

So, for instance, the delay may be decreased by a factor 10 at the cost of 15% of capacity[1].

In general, the starting point to estimate the required UDP rate is to compute the TCP segment packet rate at the bottleneck link: matching this rate will halve queueing delay. If the rate is not known, one can either assume a buffer size of 50 packets and start from there or increase the UDP packet

[1]This is the case of DSL links that do not use PPPoE for instance.

rate step by step while monitoring latency. Automating this process is one of our future work. In this paper, we focus more on validating and analyzing the effect of adding UDP traffic.

E. Sending UDP bursts

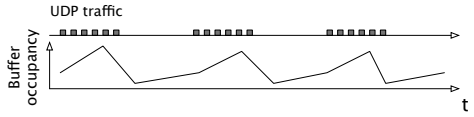

Figure 3. Sending periodic bursts of small UDP packets is enough to keep buffer occupancy to a low level

TCP connections always take time of the order of several RTTs to fill up a buffer. So, once a loss is triggered, the additional traffic that contributed to the loss becomes irrelevant for several RTTs (or a fraction of a second). It is thus sufficient to send the additional traffic by bursts of paced UDP packets. Typically, packets within a burst have the same rate as the continuous additional traffic, as the effect is similar, but they are sent with pauses that can be of the order of a second (cf. Figure 3).

III. Reverse Traffic Also Reduces the Impact of Excessive Buffers

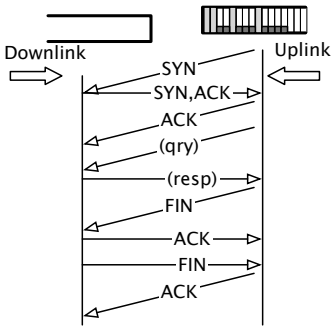

Figure 4. Each short TCP connection generates short packets that occupy the reverse buffer during 3 to 4 RTTs.

While adding UDP traffic is a means for improving latency, other traffic may have the same effect. For instance, a continuous TCP download generates a stream of ACKs in the uplink buffer that can reduce its effective size.

Although uploads may hurt downloads, the interference is less likely when the buffers are structured as arrays of packets and delayed ACK is disabled, which increases the fraction of small packets in the bottleneck buffer [4].

Similarly, an intense traffic of short download connections may effectively lower queueing. To analyze this effect, let us consider a TCP connection carrying a short query and a single data segment response (see Figure 4). Neglecting all delays other than in the bottleneck queue, this connection will have one packet in the buffer at all time during 3 to 4 RTTs, depending on the side from which it is established.

Considering a web surfing type of usage, we consider a simplified traffic model [5] in which connection establishments follow a Poisson process of intensity $\lambda = 2s^{-1}$. We approximate the RTT with the value of 500 ms (it can easily happen with an oversized uplink buffer), which means that we have $3\lambda RTT = 3$ arrivals on the average during 3 RTTs. According to our previous explanations, the 3 arrivals will occupy the buffer with one packet each during 3 to 4 RTTs.

In the simplest scenario in which all connections are initiated by the server and are only one packet long without a response from the client, we have on the average 3 ACKs in the uplink buffer. As a non negligible number of connections is larger than 1 packet, we can expect to have more queue slots occupied by small packets, therefore, the queue size is virtually reduced. Moreover, if the DSL link is shared within for instance a family, we can expect to have a superposition of many Poisson arrivals, which is itself a Poisson process of parameter $\lambda = \sum_n \lambda_n$, meaning more ACKs in the queue so a smaller visible queue for all connections.

Consequently, download traffic will often cause bursts of small packets that keep the upload congestion window from growing excessively, which has a similar effect as sending additional UDP traffic. The experiments bellow illustrate this phenomenon.

IV. Testbed experiments

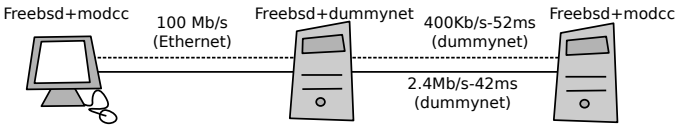

Figure 5. Testbed.

We have run a series of experiments on a testbed network composed of three computers (cf. Figure 5). One of them acts as a router interconnecting two others. The router and the hosts run FreeBSD 9.1, which allows us to test the most popular variants of TCP without switching operating systems. The Dummynet module on the router emulates the bottleneck link. Both hosts run a vanilla kernel, whereas we have recompiled the router kernel with faster context switching (option HZ set to 20,000) for accurate bottleneck link emulation. Indeed, for high bandwidth bottlenecks, the operating system scheduler may not be able to keep up with the packet departure rate on the emulated link, which results in bursts of packets sent over the network. By changing the context switching frequency to 10KHz, we support bottleneck link bit rates up to 20Mbit/s.

First, we have validated the testbed in a simple experiment. With Dummynet, the router emulates an asymmetric link: 5 Mbit/s bandwidth, 42 ms delay with a buffer of 120 packets for the downlink and 800 Kbit/s bandwidth, 52 ms delay, and a buffer of 120 packets for the uplink. We have used a TCP upload concurrently with a 250 pkt/s UDP stream and compare to the same TCP flow without UDP. The results appear in Table II.

Table II
VALIDATION RESULTS

	Without UDP Stream	With UDP Stream
TCP goodput	770 kbit/s	657 kbit/s
Link utilization	100%	85%
RTT	280 ms	113.7 ms

We observe similar results as in Section II-A: at the cost of a slight loss in throughput, the RTT is divided by almost 2.5.

We now turn our attention to more realistic traffic: bidirectional long-lasting connections and random load composed of many connections with a focus on responsiveness.

A. Long Lasting Connections

Table III
EXPERIMENT 1 PARAMETERS

τ_{download}	42ms	42 ms
τ_{upload}	52ms	52 ms
C_p	288kB/s	600kB/s
	(2.4Mb/s at L2)	(5Mb/s at L2)
C_u	48kB/s	96kB/s
	(400kb/s at L2)	(800kb/s at L2)
Uplink buffer	10 and 60 packets	20 and 120 packets
Downlink buffer	60 packets	12 packets
TCP variant	New Reno	New Reno

We consider the setup detailed in Table III. We first use an uplink buffer of 60 packets (an oversized buffer according to the rule of thumb that the buffer should contain the bandwidth $\times$ RTT product). In the first run, we start a long lasting connection in the downlink direction and then, one upload at a random time 20 to 40 seconds later. During the second run, we add a UDP stream with a fixed packet rate of 250 packets/s in the uplink direction. The experiment continues with a bursty UDP stream in the uplink direction. A burst composed of 100 packets at 250 packets/s is sent every second. Finally, we relaunch the first run with a well-configured buffer of 10 packets in the uplink direction (which leads to a more reasonable queuing delay of 300 ms). All experiments stop after 200s. We use `ipmt` suite [6] to generate connections, `tcpdump` to observe and `tcptrace` to analyze the traffic.

We observe the TCP behavior when both connections are in the congestion avoidance state and have stabilized the congestion windows (*i.e.* the transient behavior after a new connection arrival has passed and congestion window statistics stabilize): we look at the part of the trace between 50 and 100s during which both connections are running and stable.

As the experiments run in real conditions, we want to get rid as much as possible of small variations. Therefore, we launch each experiment 10 times and compute the average values and standard deviation.

B. Bufferbloat Effect

Figures 6 and 7 present the results of both experiments. For a large uplink buffer, one can observe the bufferbloat effect: the RTT attains up to 549/605 ms (slower link) and 551/600 ms (faster link), and when the upload reaches between 91 and 92% of the uplink capacity, the download drops down to around 40% of the downlink.

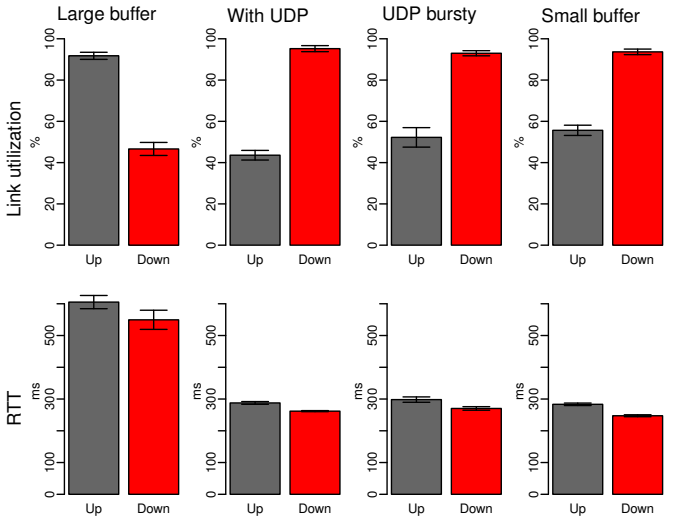

Figure 6. 1 download and 1 upload over an asymmetric link (2.4Mb/s *vs.* 400kb/s). Adding UDP traffic of 250 packets/s reduces the effective buffer size: downlink utilization increases and, more importantly, latency drops.

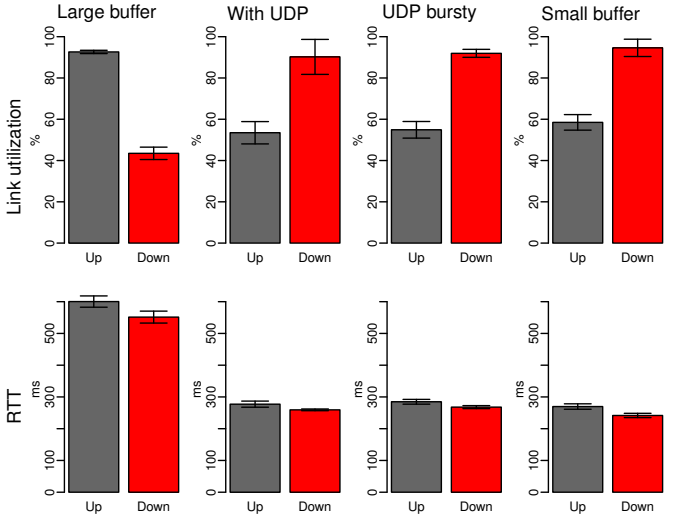

Figure 7. 1 download and 1 upload over an asymmetric link (5Mb/s *vs.* 800kb/s). Similar effect of adding UDP traffic of 250 packets/s: downlink utilization increases and latency drops.

For a correctly configured buffer, the RTT is divided by 2.5 and the download reaches around 95% of the downlink capacity, while the uplink stays around 55%.

When we add the UDP traffic, it reduces the effective buffer size, so we expect to obtain results that are similar to the case of a small buffer. Indeed, the RTT decreases by 2.5 and, even if the download does not reach 95% of the link capacity, the results are much improved without the need for changing the size of buffers.

Bursty UDP traffic leads to similar performance as the continuous UDP stream, while sending around 4 times less

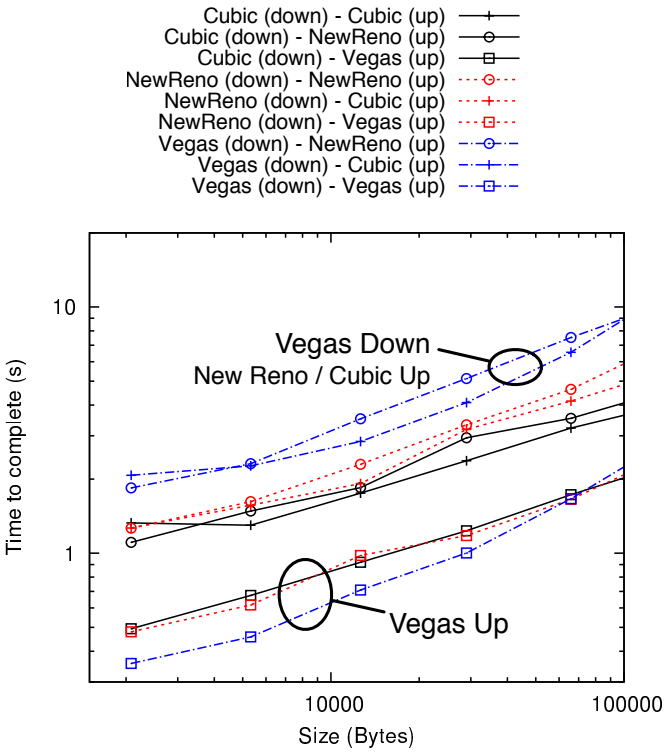

Figure 8. Conditional download response time *w.r.t.* transfer size for a 50% downlink load in presence of one continuous upload.

packets.

We cannot get lower than 0.3 seconds RTT, whatever the experiment, as the download buffer is dimensioned for having a maximum queuing delay of 300 ms.

C. Download Response Times

We then consider the case of a long lasting upload (1000s) competing with a more realistic downlink load composed of connections with Poisson arrivals and a heavy tailed size distribution. Download arrivals are modeled by a Poisson process with intensity $\lambda = 1s^{-1}$ and the transfer sizes follow a Zipf distribution with parameter $\alpha = 1.7$. This synthetic load corresponds to a downlink occupancy of approximately 50% with 1012 connections of average size 972 packets due to the presence of few large transfers.

Our objective is to capture the slowdown that a user experiences while typically browsing the Web as long as an upload is active. Note that uploads naturally occupy the link for a longer time given the link asymmetry.

We first run the experiments without additional UDP traffic. Then, we add the UDP traffic with a packet rate of 125 packets/s to the uplink and observe the difference.

Figures 8 and 9 show the conditional download response time (conditioned on the connection size) with a downlink load of respectively 50 and 17%, while Figures 10 and 11 present the download response times for the same downlink loads with additional UDP traffic.

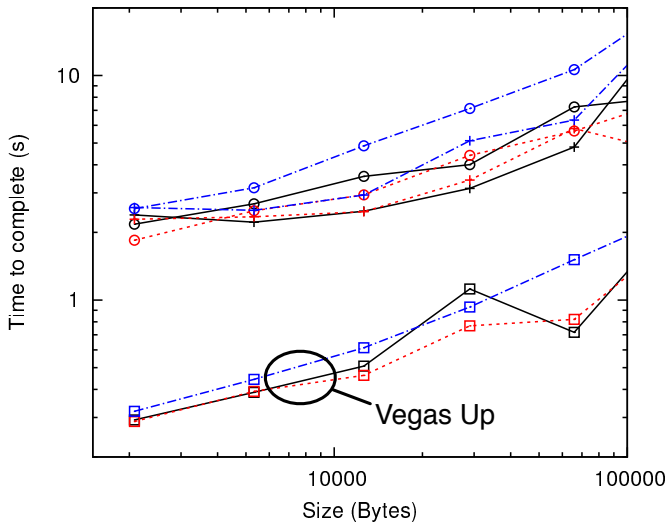

Figure 9. Conditional download response time with downlink load 17% in presence of one continuous upload.

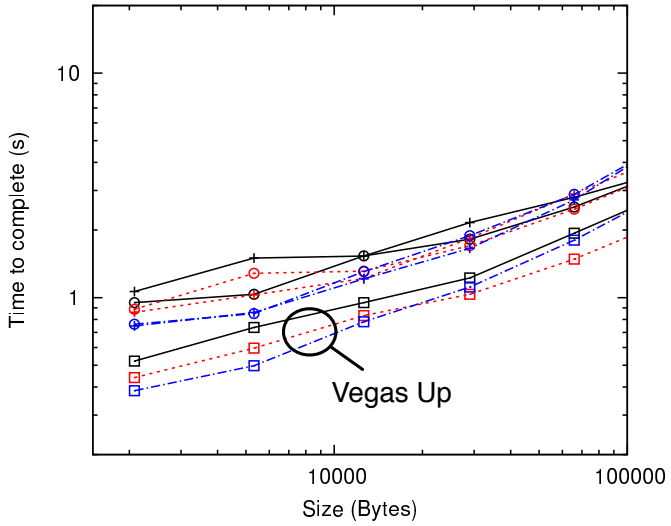

Figure 10. Conditional download response time with downlink load 50% with additional UDP traffic and in presence of one continuous upload.

Under high load (cf. Figure 8), Vegas used for the upload (the lines with square points) results in consistently low download response times, as downloads find mostly empty buffers and therefore complete much earlier. If we use Vegas for the download, we observe longer completion times, as Vegas sees longer RTT due to the upload, so it tends to lower its sending rate, thus increasing the response time.

Under lighter load (cf. Figure 9), as long as Vegas is used for the uploads, the connections complete much sooner, as they again experience consistently low delays. Furthermore, when competing against TCP Cubic or New Reno, the impact on downloads is more important under lighter load. The explanation is that the lighter downlink load in insufficient to perturb the upload as the stream of ACKs remains relatively

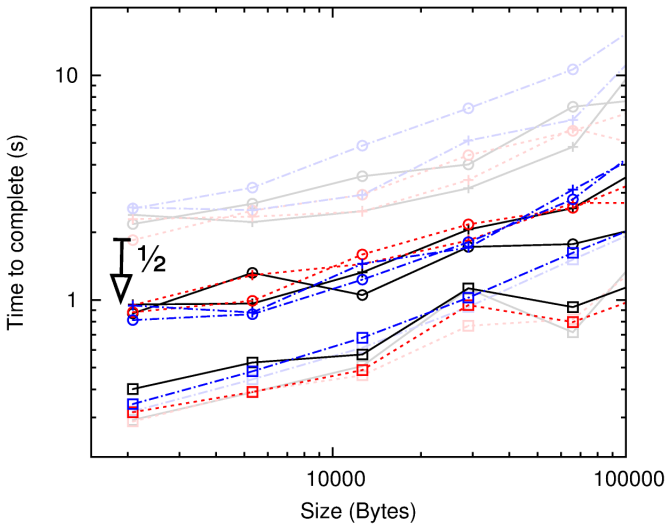

Figure 11. Conditional download response time with downlink load 17% with additional UDP traffic and in presence of one continuous upload. For reference, the results without UDP are shown grayed out in the background.

low. So the upload steadily occupies the uplink buffer leading to longer completion times.

UDP traffic (cf. Figures 10 and 11) results in significantly improved response times. The results are quite similar to the cases that use TCP Vegas on the uplink. They even go up to two times better for connections that use TCP Cubic or New Reno on the uplink (for load at 17%)

D. Realistic TCP mix

In this experiment, we run an upload to a distant TCP server behind a real ADSL link with 5440 kbit/s downstream and 352 kbit/s upstream bit rates. On the TCP server, we generate a random Poissonian/Zipf TCP download traffic as described in Section IV-C. With a Poisson intensity of $1s^{-1}$ and Zipf of 1.7, we generate 1061 download TCP connections over 1000 seconds with the average connection size of 660 packets and with 898 connections less than 10 packets. The upload connection goes through a buffer of 30 packets, while the download encounters an estimated 60 packets buffer. We expect to encounter the effects of bufferbloat on downloads. We use three versions of TCP (Vegas, NewReno, and Cubic) for upload and download. Then, we add upstream UDP traffic of 125 packets/s.

Furthermore, we give the results for an experiment with a higher load of 6 connections per second. We generate 5851 connections over 1000 seconds, with average size 413 packets and 5002 connections under 10 packets. We except to have better results than the first experiment according to part III.

The results are presented in Figures 12, 13 and 14. They confirm the results obtained in Section IV-C. With Vegas, download response time is much better than with other congestion control schemes. Indeed, TCP Vegas succeeds to keep the queue as empty as possible, so the queue is occupied mainly by UDP traffic and ACKs. Besides, with Vegas, the UDP

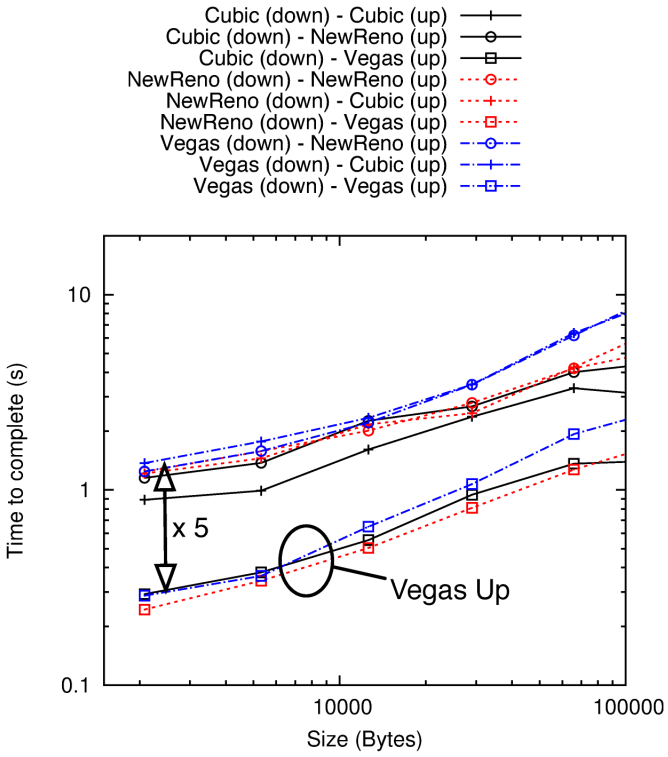

Figure 12. Conditional download response time with 1 connection per second in presence of one continuous upload, realistic TCP mix.

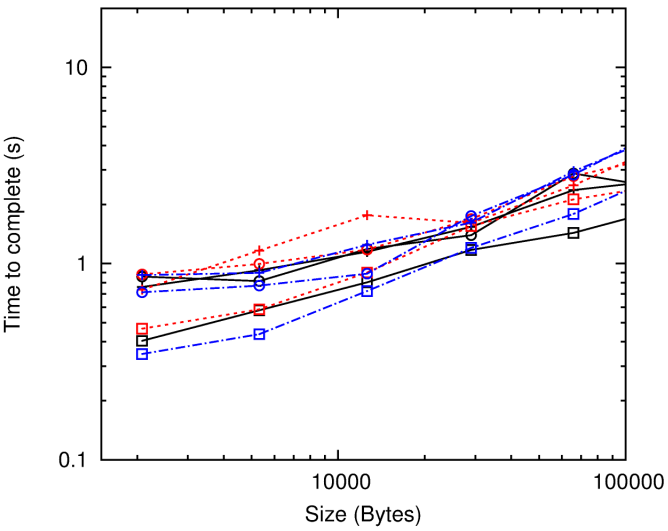

Figure 13. Conditional download response time with 1 connection per second in presence of one continuous upload and a 125 pkt/s UDP stream, realistic TCP mix.

traffic becomes in this case significant as it raises the queue occupancy, while the upload tries to decrease it. Anyways, as TCP Vegas already obviate the bufferbloat problem, adding an additional UDP traffic is a bit absurd, although mostly harmless.

Finally, with a more intense downlink traffic, it is interesting to note how TCP New Reno and Cubic get contrasted

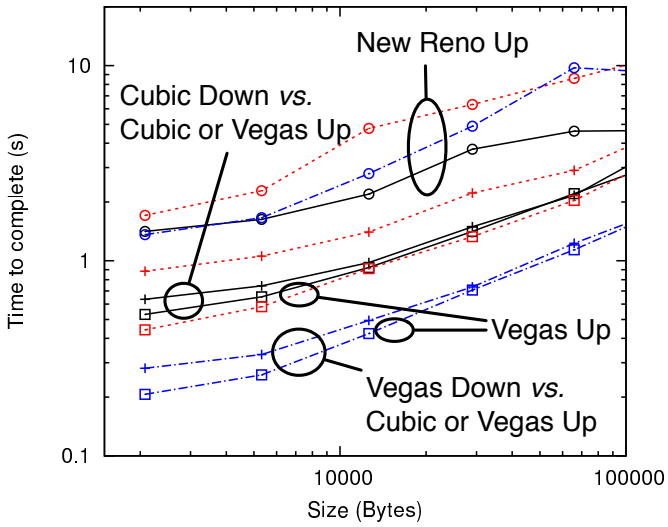

Figure 14. Conditional download response time with 6 connections per second in presence of one continuous upload, realistic TCP mix.

performance: New Reno for the upload leads to much higher response times, whereas all the other algorithm give better results. Here Vegas for the upload is still beneficial; unfortunately, New Reno a lot more frequent than Vegas on end hosts.

On this ADSL link, the performance differences are not as marked as those from the testbed experiments. On reason is that the buffer in the modem is only 30 packets (compared to 60 packets in the testbed experiment). Moreover, the link is much more asymmetric with the ratio of 15 compared to the ratio of 6 in the testbed. With such asymmetry, as soon as we saturate the downlink buffer, ACKs arrive at a faster rate, thus occupying the uplink buffer.

Considering the experiment under a higher load, we can observe that the average response time is lower due to a higher number of ACKs in the uplink queue.

V. Related work

RFC 970 [7] published in 1985 expressed the first concerns about excessive buffer sizes.

In 1996, Chesire wrote about the problem of latency in the Internet in a note titled *It's the Latency, Stupid* [8]. He proposed a few solutions given that *"once you have bad latency, you're stuck with it"*. Nevertheless, most of the solutions are development guidelines for a proper use of the network (compress traffic, caching, send less data etc.).

Researchers started to discuss the actual problems raised by excessive buffers in 2009 [9] and a workgroup on the bufferbloat began in 2011 [10]. The discussions proposed one immediate solution—correctly sized buffers, although it would require to not only change their size, but also their structure: for instance, a 3 packet uplink buffer results in poor performance, because it will overflow immediately in presence of a mix of ACKs and TCP data segments. Conversely, limiting the buffer capacity to a number of bytes and not to a number of packets (*e.g.* 5000B) reduces latency and improves the uplink and downlink utilization.

Recently, Nichols et al. proposed new queueing strategies such as CoDel [11] or FQ-CoDel [12] that schedule packets based on their sojourn times in the buffer. They are by now good solutions, but they take place in the core of the network and require heavy intervention while we propose a solution that can be set up by the end user.

End-to-end congestion control mechanisms can also help. For instance, TCP Vegas [13] computes the expected throughput based on *RTT* and congestion window *cwnd* to keep the queue occupancy as low as possible. The Linux Kernel integrated several modifications to counter bufferbloat [14]: a combination of fair queuing, TCP stack modifications limiting the amount of data queued in the network, as well as some recommended configuration.

VI. Conclusion

The paper explores solutions for limiting the negative effects of bufferbloat—increased latency due to excessive buffer sizes. Obviously, there exist mechanisms that can solve the problem, however their deployment may be difficult or not feasible. So, there is some space for other approaches that may bring better performance if the known solutions cannot be applied.

Our main finding is the paradoxical observation that adding low intensity traffic of short packets improves latency at the price of a slight decrease in throughput. We also notice that the reverse download traffic has beneficial influence on performance, which explains why the negative effect of bufferbloat sometimes appears as alleviated.

In this paper, we have concentrated on the effect of the additional traffic, neglecting the problems it may cause on subsequent links, if any. In fact, a further improvement of the method would be to send the additional packets with a small TTL to discard them immediately after crossing the bottleneck link. In this case, it would be beneficial that the ISP applied rate limiting to the generation of TTL-exceeded ICMP packets as it is often the case to fight TTL Expiry attacks.

In the future work, we propose to find a means for an unsupervised adjustment of the additional traffic parameters, for instance to suppress traffic when the uplink is lightly loaded.

We also need to consider the case of wireless links, especially in the uplink direction, for which the capacity may greatly vary and short packets could introduce too large overhead (actually, this holds for 802.11 links, but not for cellular networks that may concatenate frames for transmission).

References

[1] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling Bufferbloat in 3G/4G Networks. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 329–342, New York, NY, USA, 2012. ACM.

[2] CACM Staff. Bufferbloat: What's Wrong with the Internet? *Communications of the ACM*, 55(2):40–47, 2012. A discussion with Vint Cerf, Van Jacobson, Nick Weaver, and Jim Gettys.

[3] M. Allman. Comments on Bufferbloat. *ACM SIGCOMM Computer Communication Review*, January 2013.

[4] Martin Heusse, Sears A Merritt, Timothy X Brown, and Andrzej Duda. Two-way TCP Connections: Old Problem, New Insight. *ACM SIGCOMM Computer Communication Review*, 41(2):5–15, 2011.

[5] I.A. Rai, E.W. Biersack, and G. Urvoy-Keller. Size-based scheduling to improve the performance of short tcp flows. *Network, IEEE*, 19(1):12–17, Jan 2005.

[6] IPMT Test Suite. http://ipmt.forge.imag.fr.

[7] J. Nagle. RFC 970 On Packet Switches with Infinite Storage. http://www.ietf.org/rfc/rfc970.txt, December 1985.

[8] S. Cheshire. It's the Latency, Stupid. http://rescomp.stanford.edu/~cheshire/rants/Latency.html.

[9] B. Turner. Has AT&T Wireless Data Congestion Been Self-Inflicted? http://blogs.broughturner.com/2009/10/is-att-wireless-data-congestion-selfinflicted.html.

[10] J. Gettys. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing*, 15(3):96–96, May 2011.

[11] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5):20–34, 2012.

[12] T. Høiland-Jørgensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. FlowQueue-CoDel. IETF Internet draft, March 2014. draft-hoeiland-joergensen-aqm-fq-codel-00.

[13] Lawrence S. Brakmo, Sean W. O'malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM'94*, 1994.

[14] T. Høiland-Jørgensen. The State of the Art in Bufferbloat Testing and Reduction on Linux. http://www.ietf.org/proceedings/86/slides/slides-86-iccrg-0.pdf, March 2013.

978-0-9836283-9-2