Fighting Fire with Fire: Eliminating Standing Queues with Large UDP Packet Floods

Michael Welzl  
Department of Informatics  
University of Oslo, Norway  
Email: michawe@ifi.uio.no

Stein Gjessing  
Department of Informatics  
University of Oslo, Norway  
Email: steing@ifi.uio.no

Haakon Botnmark Jahre  
Department of Informatics  
University of Oslo, Norway  
Email: haakobja@ifi.uio.no

Abstract—“Bufferbloat”, extensive buffers in network equipment, can create severe delay for packets that traverse them. This can severely degrade the user experience. Bufferbloat has recently been addressed with various Active Queue Management (AQM) mechanisms, but these can only help if they are installed in the problematic device. When this device is not under control of the user who intends to fix the problem, there is not much that can be done. We present “Queue Flusher”, a tool that monitors TCP connections and tries to automatically detect the presence of standing queues anywhere along the paths traversed by the TCP connections that it sees. Upon detection, it sends a burst of UDP packets towards the destination of one of the TCP connections in question, in the hope that congestion controlled traffic traversing this queue reacts to the burst by reducing its rate. Evaluation results from a simple emulated network show that this mechanism can work surprisingly well, with a pronounced improvement in average delay at the expense of only a modest reduction in throughput.

I. INTRODUCTION

Excessive buffering in network equipment, sometimes called “buffer bloat”, can create very large delays for packets traversing the Internet [1]. This degrades the user experience, especially when communication is latency critical, for example with interactive multimedia applications (e.g. VoIP). Latency is also a key factor in web browsing, where it has a monetary value; this is underlined by several recent efforts to reduce Round-Trip Times (RTTs) for this type of application [2].

The bufferbloat problem has recently been addressed with Active Queue Management (AQM) mechanisms such as CoDel [3], [4], PIE [5] and the compound scheduling+AQM algorithm FQ_CoDel [6] – however, such mechanisms need to be installed in the device where the problem occurs. Sometimes, large uncontrolled buffers cause delay in a device that is far away from the user who suffers from the problem, and not under control of this user.

We present a simple software-only approach that can help in such a situation: a “Queue Flusher” that monitors the RTT of packets and, upon seeing a significant and sustained growth in delay, sends a burst of UDP packets in the direction where the problem is suspected. Such sustained delay growth can be caused by what is sometimes called a “standing queue”: a queue that maintains a certain permanent filling level. In the presence of more than a bandwidth × delay product of queuing, even standard TCP can generate a standing queue [4].

Figure 1 shows a scenario where a Queue Flusher is installed at the edge of a corporate network, and congestion is caused at a queue on a path somewhere in the Internet, possibly by an entirely “outside” traffic source (“flow 2” in the figure). If flow 2 is non-responsive, there is nothing that can be done about it from the position of the corporate network. If, however, flow 2 is a TCP flow or some other form of congestion controlled traffic (e.g. an adaptive multimedia application), it is bound to react to packet loss. Such loss is provoked by the Queue Flusher by sending a data burst towards the remote end of Flow 1.

We will explain our Queue Flusher implementation in the next section. In Section III, we provide performance evaluation results from a simple emulated testbed. After discussing related work in Section IV we conclude with a discussion of possible future research directions.

II. THE QUEUE FlushER

The scenario depicted in Figure 1 may look easy, but realistically supporting it brings about a number of complications. For example, the RTT of Flow 2 in the figure can be shorter or longer than the RTT of Flow 1; there can be many flows traversing the bottleneck, consisting of short web-like traffic or a mix of such traffic and long-term traffic. Flow 1 may itself be the culprit causing the standing queue or it may be an application-limited low-rate flow. Ensuring that the Queue Flusher operates correctly in such an environment requires considering a large variety of environment conditions, leading
to fine-tuning of the mechanism itself and its parameters.

Given the rather drastic nature of the idea, our goal is to simply see if “flooding” has any merit. Therefore, such extensive testing (and the ensuing complex algorithm design) is unnecessary. We have focused on a much simpler case, where Flow 1 in Figure 1 is the only one, and it is therefore the flow that generates the standing queue. To simplify RTT calculation, we also assumed that the Queue Flusher is run directly at the sender host and designed the mechanism accordingly.

The Queue Flusher operates as follows:

- It listens to all incoming and outgoing TCP packets using the “libpcap” library and maintains lists of necessary information of all packets, grouped by flow (identified via the 5-tuple: source and destination IP addresses and port numbers and the IP header’s protocol field).
- Whenever an ACK arrives, it checks if information from a corresponding data packet exists in the list. If so, it calculates an RTT sample and erases all information from data packets preceding the acknowledged one from its list. For each flow, three variables are kept, and updated upon ACK arrival:
  1) base_rtt, the minimum RTT seen during a fixed time interval. Since the minimum here is – similar to LEDBAT – used as an offset to delay from the queue growth, we use the same default duration for the interval as LEDBAT: 10 minutes [7]. As in LEDBAT, base_rtt is refreshed after this interval to avoid misinterpreting permanent changes in a path (e.g. due to a routing change).
  2) An Exponentially Weighted Moving Average (EWMA) (using $\alpha = 7/8$ as in TCP’s timeout calculation [9], where $\alpha$ is used as a weight for the previous value, i.e. new sample values are weighted with $(1-\alpha)$) of the flow’s RTT.
  3) The flow’s sending rate during a fixed interval in bits per second.
- A flushing decision is made by keeping track of these variables for a while: they are stored in a ring buffer of size 10 (as default), where new values are added whenever the variables are updated (i.e., when an ACK arrives). If the lowest RTT in this buffer exceeds $\text{threshold} \times \text{base}_\text{rtt}$, flushing is activated, sending out a Bandwidth×Delay-Product (BDP) worth of packets, where the bandwidth is the maximum throughput value in the ring buffer and the delay is set to $2 \times \text{base}_\text{rtt}$ (which would be the maximum delay seen in case of a BDP-sized queue). This amount is what we assume to be enough for TCP to notice and take action as it should truly “fill the link”. As a result, flushing takes around 2 RTTs. If the flushe stays in “flushing mode” (i.e. the RTT does not fall below $\text{threshold} \times \text{base}_\text{rtt}$) then the flushe has not sent enough, and the flushing amount is multiplied by 1.2 for subsequent flushes.

Some decisions made in this algorithm are somewhat arbitrary: e.g., the notion of a “sustained” increase in the RTT is here a matter of having pushed the EWMA up high enough and maintaining it for the next 10 samples. Clearly, whether fixed values such as the number of samples to monitor, the EWMA’s $\alpha$ value and the 1.2 increase factor are well chosen will depend on environment conditions. Accordingly, we chose these values to work reasonably well in the scenarios that we evaluated, but we leave further optimization of the algorithm itself as well as fine-tuning of its parameters for future work. The evaluated scenarios include a range of conditions, such as RTTs from 10 to 100 ms and various queue sizes in some tests, i.e. the algorithm was not “unfairly” tuned to work well in only one specific situation.

III. EVALUATION

We tested the Queue Flusher running on a host that sends a single TCP flow (with iperf) using 1500-byte packets through a 10 Mbit/s access link via a router and a 5 Mbit/s bottleneck to a receiver. This 3-host topology was emulated using the CORE network emulator in Linux, and the plotted RTT values were obtained with SPP\(^1\) (using tcpdump running on the sender and receiver host). Unless otherwise noted, the queue at the bottleneck is a DropTail (FIFO) queue with a length of 100 packets. This queue is “bloated”, and hence expected to yield a standing queue when TCP is run without the flushe: it exceeds the BDP in all our tests (the maximum network base RTT that we used was 100 ms, yielding a BDP of 41 packets).

Figures 2 and 3 show the effect that running the flushe has on the RTT of the TCP flow. In Figure 2, TCP produces a standing queue that corresponds with a permanent minimum delay exceeding 150 ms. Despite TCP halving its congestion window, the queue never completely drains. In Figure 3, we can see that the flushe kicked in at above 200 ms

\(^1\)http://caia.swin.edu.au/tools/spp
(2 × base\_rtt), briefly creating a large delay spike, but causing TCP to react and allowing it to fully drain the queue.

Figures 4 and 5 show the throughput corresponding with Figures 2 and 3, respectively. The influence of the flusher is clearly visible, but the overall reduction in throughput is relatively modest, given the significant benefit shown in the earlier figures: the standing queue was removed and all RTT measurements (with the exception of the actual flushing time) were shifted down by about 50 to 100 ms.

To better understand the effect that flushing has on the distribution of RTT samples, we ran experiments with varying queue lengths of 10, 20, 30, 100 packets and different network base RTTs of 10, 50 and 100 ms, using Cubic and Reno. Every run lasted five minutes. We also varied the base\_rtt multiplication factor threshold, which controls the delay tolerance of the Queue Flusher – a small threshold value means more aggressive flooding.

Figures 6, 7 and 8 show the Cumulative Distribution Function (CDF) of the “RTT gain” in these experiments: the values show the difference to the corresponding values in the case without the Queue Flusher. For example, a depicted gain of 50 ms means that the RTT was 50 ms smaller than in the case without the Queue Flusher, and a negative value means that the Queue Flusher made the delay worse. In all three cases, delay was worsened for only less than 20% of all samples, and the overall gains are quite large – for example, in the 10 ms base\_rtt case, at least half of the samples gained more than 130 ms (Reno) and 160 ms (Cubic) – more than ten times base\_rtt. Regarding the threshold value, considering these figures, a value of 1.7 appears to be a good compromise. However, we note that all threshold values saw a significant performance gain for the large majority of RTT samples.

We now fix the threshold value at 1.7 and “zoom into” the behavior with different queue lengths, with the medium base\_rtt value of 50 ms, for the case of Reno (tests with Cubic and other RTTs look similar). In the boxplots shown
in Figures 9 and 10, the upper and lower “hinges” correspond to the 25th and 75th percentiles. The upper and lower whisker extends from the hinge to the highest / lowest value that is within 1.5×IQR of the hinge (the Inter-Quartile Range IQR is the distance between the first and third quartiles). Data beyond the end of the whiskers are outliers and plotted as points. Figure 9 shows that delay generally grows with the queue length; however, when the Queue Flusher is active, the majority of values can be kept within a fixed range, and only the outliers extend towards the top of the RTT range. These outliers are caused by the flusher itself as it exceeds the queue length with the UDP packet burst. Naturally, this comes at a cost in throughput, but again, this degradation is quite modest and seems to have an upper bound that does not depend on the queue size for the majority of values (Figure 10).

Figure 11 depicts the RTT with and without the Queue Flusher as well as with CoDel and PIE. Since AQM mechanisms operate directly at the queue, it is without a doubt that they can perform better than the flusher – and they do, but the

flusher is surprisingly close in terms of RTT gain – and, as Figure 12 shows, CoDel’s greater RTT reduction comes at a larger cost in throughput than with the Queue Flusher.

IV. RELATED WORK

In addition to the aforementioned work on Active Queue Management (CoDel, FQ_CoDel and PIE), there is a large amount of earlier work on similar algorithms [10]. All of these mechanisms address the problem directly at the queue, where the problem occurs.

As another alternative to our aggressive method to keep a remote queue under control, congestion control mechanisms installed in end systems can try to react earlier than TCP normally does: not only upon packet loss, but when a growing queue is detected. This is most commonly done by tracking delay and assuming that it stems from this queue.
There is a large amount of work on such delay-based congestion control; a well-known early example is TCP Vegas [11], and more recent ones are the NADA [12], SCREAM [13] and GCC [14] congestion control algorithms that are currently being proposed in the IETF as delay-limiting congestion control solutions for WebRTC.

Such mechanisms require special functions that keep them from being too cautious: a new mechanism that reacts upon queue growth cannot stop a competing TCP flow from increasing the queue, thereby producing unwanted delay (in IETF discussions, the phrase “don’t try to be friendly to TCP, as it will not return the favor” was coined). A number of mechanisms have turned this disadvantage into an advantage, making an early reaction to queuing delay a favorable approach for low-priority transfers that should “step out of the way” of other traffic when such traffic arrives. A prominent recent example is LEDBAT [7], a variant of which is known to be used in the official BitTorrent client, and a survey of prior related work is provided in [15] and [16].

Explicit Congestion Notification (ECN) [17] is a standard that defines how to use a set of bits to allow transport end systems to communicate with an AQM mechanism. By avoiding unnecessary packet loss, this can in principle allow an AQM mechanism to mark packets from ECN-capable senders before they would normally be dropped [18]. Such early marking however requires special mechanisms in the end systems to avoid being starved by non-ECN-capable senders. One recent proposal can produce very good performance but requires a uniform update of the ECN standard across the network [19], whereas another proposal [20] seeks a compromise by first updating only the sender side without changing the router behavior, potentially enabling routers to mark early only when enough senders have been updated.

V. Conclusion

We have presented a simple temporary fix to the bufferbloat problem, addressing the case where the “bloated” queue is remote and not under control of the person who intends to fix the problem. Our results show that our somewhat drastic approach of “flushing” the queue by sending a burst of UDP packets can produce surprisingly good results. There is however no doubt that sending such an amount of traffic in an uncontrolled fashion can have very negative side-effects. For example, it can induce delay growth and packet loss for flows in other queues than the bloated one, and the potentially excessive packet loss that the Queue Flusher can cause is by itself harmful for loss-sensitive applications that might communicate across the bloated queue.

The most critical factor for latency-sensitive applications such as VoIP is often the peak, not the average latency. This means that more frequently reaching the peak, as in Figure 3, might actually degrade the user-perceived quality, even if the large majority of the samples depicted in Figure 3 have a smaller RTT than the ones in Figure 2. It could be better for the user to experience the “flush” as a brief interruption or distortion rather than seeing a sustained delay increase because of delayed buffering effects in end systems. This could be facilitated by the flusher explicitly informing end systems of a flush, e.g. by sending a signal to hosts inside the corporate network in the scenario shown in Figure 1.

It is obvious that it is better to prevent bufferbloat by installing a solution directly at the bloated queue, if at all possible; our Queue Flusher is therefore at best a temporary fix. Given how good it performed in our tests, we however think that it is worthwhile to investigate it further and optimize it for intermediate practical use, mainly by ensuring that it will only turn itself on as an absolute last resort – somewhat similar to a “circuit breaker” [21].

References