SCTE CABLE-TEC EXPO '13
OCTOBER 21-24 / ATLANTA, GA

SCTE Society of Cable Telecommunications Engineers

Minimizing Bufferbloat and Optimizing Packet Stream Performance in DOCSIS 3.0 CMs and CMTSs

A Technical Paper prepared for the Society of Cable Telecommunications Engineers
By

Tom Cloonan
Chief Technology Officer- Network Solutions
ARRIS
2400 Ogden Ave.- Suite 180, Lisle, IL 60532
630-281-3050
tom.cloonan@arrisi.com

Jim Allen
Staff Software Engineer
ARRIS
2400 Ogden Ave.- Suite 180, Lisle, IL 60532
630-281-3020
jim.allen@arrisi.com

Tony Cotter
Staff Software Engineer
ARRIS
Building 4300, Cork Airport Business Park Kinsale Road, Cork Co., Ireland
tony.cotter@arrisi.com

Ben Widrevitz
Sr. System Architect
ARRIS
2400 Ogden Ave.- Suite 180, Lisle, IL 60532
630-281-3273
ben.widrevitz@arrisi.com

Jeff Howe
Sr. Director- Broadband Architecture
ARRIS
2400 Ogden Ave.- Suite 180, Lisle, IL 60532
630-281-3124
jeff.howe@arrisi.com

Overview

You are trying to catch a plane…it departs in a short 15 minutes! You arrive at airport security and encounter a line with 120 people ahead of you. They are all people destined for a different plane that won't depart for one and a half hours… and they are not in a hurry. You ***are*** in a hurry, but you must now wait. And your wait will be as long as it takes to service all of the other 120 people ahead of you in the line. If it takes one minute to perform a security check on each person, then the wait will be 120 minutes… 2 whole hours! Performing these mental calculations in your head, you realize that it is futile- you ***will*** miss your plane.

If you ever had an experience like that (stuck in line behind a large group of people when you are in a hurry), then you are familiar with the frustrating phenomenon known as Bufferbloat. It affects people in lines, and it affects packets in the queues of network elements (like routers, switches, CMTSs, and CMs).

Bufferbloat wthin the Internet is formally defined to be the undesirably long latency that can be experienced by a latency-sensitive packet that arrives at a shared queue when another set of packets from a different flow (that may or may not be latency-sensitive) are already filling many of the buffers within the queue. Many papers have hypothesized that the existence of Bufferbloat within upstream CM buffers and downstream CMTS buffers could be one of the primary sources of frustration for many Cable Data users.

Problems related to Bufferbloat have most likely always been present within the Internet, but they have become more apparent and more problematic in recent years due to several trends:

1) Higher bandwidth service tiers (ex: 50 and 100 Mbps tiers) are now being offered by service providers (such as Multiple System Operators or MSOs)
2) Higher bandwidth applications have become commonplace (ex: Peer-to-Peer File Transfers, Streaming Video, etc.) that can easily capitalize on the higher bandwidth service tiers
3) Those higher-bandwidth applications are mixed with many other applications that may not require high-bandwidth, but that do require low-latency transport (ex: VoIP, Web-Browsing, Gaming, etc.)
4) Network elements (routers, switches, CMs, CMTSs) now have the ability to build in large buffers using inexpensive, high-speed buffer memories, and some

network elements have taken advantage of this fact by placing large buffers in their ingress ports to help absorb (without packet drops) any transient bursts of bandwidth that may occur on the Internet links

5) Many modern network elements have high-bandwidth ingress links and low-bandwidth egress links. The Bufferbloat problem is exacerbated by the fact that the bursts on the high-bandwidth ingress links can easily fill up the buffer memories without giving them a chance to be drained by the low-bandwidth egress links. (Note: Upstream paths through CMs are particularly susceptible to this problem, and downstream paths through CMTSs are (to a lesser extent) also susceptible to this problem).

At a 10,000-foot level, managing Bufferbloat is relatively simple- it simply requires techniques that help to limit the depths of the buffers in the network elements through which the latency-sensitive packets are propagating. However, this procedure must be implemented with care, because many attempts at minimizing buffer depths (to mitigate Bufferbloat) will often lead to undesirable increases in packet loss rates and can also place undesirable limits on the maximum throughputs for TCP sessions with long Round-Trip Times (RTTs). As an example, one can approximate the peak TCP bandwidth for each RTT value (assuming that the buffer size must equal the TCP congestion window), and those approximations are tabulated in **Table 1**.

Buffer Size (Kbytes)	RTT = 20 msec	RTT = 100 msec	RTT = 200 msec
8	3.2 Mbps	640 kbps	320 kbps
16	6.4 Mbps	1.28 Mbps	640 kbps
32	12.8 Mbps	2.56 Mbps	1.28 Mbps
64	25.6 Mbps	5.12 Mbps	2.56 Mbps
128	51.2 Mbps	10.24 Mbps	5.12 Mbps
256	102.4 Mbps	20.48 Mbps	10.24 Mbps

Table 1- Peak TCP Throughputs as a function of RTT values and Buffer Size Values

Thus, the challenge for most Bufferbloat mitigation techniques is to find the “sweet spot” that mitigates Bufferbloat, minimizes packet loss, and permits TCP sessions to operate with high throughput.

This paper will focus on the pros and cons of various techniques for reducing the effects of Bufferbloat within packet-based DOCSIS networks. The authors recognize and applaud the stellar work already carried out by Greg White and Dan Rice at CableLabs. The CableLabs paper [Whi2] describes the results of simulations comparing the performance levels of various techniques for managing Bufferbloat. The simulated techniques within that paper included:

1) Saturated Tail-Dropping Queues with large buffer depths
2) Saturated Tail-Dropping Queues with short buffer depths (optimized using the new DOCSIS 3.0 Buffer Control ECN, feature [Whi1] to set depths equal to the expected Bandwidth-Delay Product, or BDP)
3) The Controlled Delay (CoDel) active queue management technique [Nich]
4) The Proportional Integral Enhanced (PIE) active queue management technique [Pan]
5) The Statistical Flow Queue with CoDel (SFQ-CoDel) active queue management technique

As described in that CableLabs paper, all of the above approaches can help to reduce the impact of Bufferbloat, but each approach uses a slightly different technique and has very different performance results under different loads. The two Saturated Tail-Dropping approaches perform simple dropping of packets whenever queues reach their maximum size, but they tend to suffer from the fact that they do not respond to queue build-ups quickly and can then be forced to drop many packets (including latency-sensitive packets) once queue saturation is reached. As a result, some latency-sensitive packets still experience Bufferbloat in Saturated Tail-Dropping systems. The CoDel and PIE approaches define various techniques for dropping packets and throttling high-bandwidth TCP flows sooner than Saturated Tail-Dropping does- before the buffers reach saturation. CoDel triggers these drops using measured packet delays (which can be used to infer buffer depths) and PIE triggers these drops based on estimated buffer depths, and both approaches provide improvements over the Saturated Tail-Dropping approaches. The SFQ-CoDel approach is more complex, because it actually establishes different queues for each of the packet flows within a service group, and it uses hash codes to ideally steer different packet flows into different queues, and it then services the queues in a round-robin fashion. It has very good performance, but the performance levels suffer if there are hash collisions that steer two or more flows into a single queue, so the approach included a CoDel algorithm which attempted to use queue drops to deal with the performance degradation that results from a hash collision.

In this paper, we will not attempt to repeat the good work performed at CableLabs. Instead, we will attempt to build on and extend their work by studying some unexplored areas. We define a taxonomy of three generic but different Bufferbloat mitigation approaches (Type A, Type B, and Type C) and then explore all three approaches. As a result, three particular extensions to the CableLabs work will be explored within this paper:

Type C Extension: One extension to the CableLabs work will use simulations to explore (in more depth) the efficacy of the simple Type C Saturated Tail-Dropping proposals, monitoring packet latency as well as average bit-rate, bandwidth stability, and packet loss as the buffer depths are ranged well above and well below the desired BDP value. This work will help MSOs understand what happens when the predicted BDP values (used to select buffer depths for the new DOCSIS 3.0 Buffer Depth ECN)

do not exactly match the actual BDP values experienced by the TCP sessions flowing through those buffers.

Type A Extension: Another extension to the CableLabs work will take a fresh look at resolving the hash collision problems that were identified by CableLabs within the Statistical Flow Queing (SFQ)-based solutions. It will be shown that the resulting performance of the SFQ-based solutions may be optimal and may circumvent many of the hash collision problems identified by the CableLabs work. A technique for minimizing hash collisions is presented and leads to a new Type A Bufferbloat management technique that we call SFQ with Hashing/Serial Searching. This approach to Bufferbloat management may be viewed as an optimal solution if the supporting hardware can provide the resources required for this slightly more complex solution.

Type B Extension: This paper will also discuss and characterize several variants of a new Type B active queue management proposal that was not explored in the CableLabs paper. This low-complexity approach will use simple Latency-based Random Early Detection (LRED) dropping (instead of Saturated Tail-Dropping) as an alternative mechanism to trigger TCP throttling prior to filling the buffer. LRED is a Bufferbloat mitigation technique that is similar to (but subtly different from) a normal RED (or WRED) algorithm. Details and benefits of the LRED scheme will be described below.

Background on Bufferbloat

The term Bufferbloat was popularized by Jim Gettys in his seminal 2011 paper [Gett]. Within that paper, Gettys identified the existence of the problem and argued that large buffers in network elements are the principal cause, because they can lead to excessive network delays with negative impacts on many latency-sensitive applications traversing the Internet. He theorized that the availability of lower-cost and higher-density DRAMs (Dynamic Random Access Memories) has caused some manufacturers of network elements to place larger and larger buffers into their products without carefully examining the implications of those larger buffer sizes.

In his paper, Gettys pointed out some repercussions of those larger buffer sizes. Gettys argued that behavior of TCP flows will oftentimes saturate and fill up the large available buffers contained within some of the Internet's network elements. Large, saturated buffers can lead to long delay times for all packets passing through those buffers. While this condition may not be detrimental to many bandwidth-intensive TCP sessions (like FTP downloads), Gettys argued that the increased delays caused by the existence of these heavily-filled buffers could be detrimental to many other latency-sensitive types of network traffic (such Gaming sessions, Over-The-Top VoIP sessions, Web-browsing sessions, etc.).

To show an example with typical numbers, assume that a network element (like aCM) is driving its Upstream packets through a 2 Mbps DOCSIS Upstream service flow (SF). If

the CM uses a buffer for the SF that is 500 Kbyte in length (333 packets with typical 1500 byte lengths), then a full buffer will take (500 Kbytes)*(8 bits/byte)/(2 Mbps) = 2 seconds to empty. Thus, any packet that is injected into the CM when the buffer is full will incur a 2 second delay just to pass through the CM. This leads to high latencies on the packet streams, and these latencies can be entirely unacceptable for latency-sensitive applications like VoIP sessions or Gaming or Web-browsing sessions that might be injecting packets into the heavily-filled buffers. These lengthy 2-second packet delays can cause other problems as well. As an example, it also implies that TCP bandwidth adjustments for congestion avoidance at other points in the Internet will not be able to occur quickly, because (if ACK acceleration is not enabled) ACKs going back through the CM on the Upstream towards servers in the Internet will be delayed by the same 2 second period. As a result, slow TCP bandwidth adjustments could lead to uncorrected and unacceptable congestion levels developing on the Downstream paths (due to ACKs being delayed in the Upstream paths).

Since many latency-sensitive application types (such as VoIP) do not actually generate enough bandwidth to saturate typical CM and CMTS buffers, two or more traffic streams from different applications must often be multiplexed through a single buffer to create the requisite conditions for the Bufferbloat problem to exist. If multiple applications are sharing a buffer, then the high-bandwidth applications tend to saturate the buffer and the latency-sensitive applications tend to experience the negative repercussions of the resulting delays within the saturated buffer.

Gettys's paper explores an age-old belief and rule-of-thumb stating that the amount of buffering that should be utilized within any network element should be equal to the BDP of the TCP session. The BDP (in this context) is usually defined to be the available bandwidth capacity of the egress link (or the available bandwidth capacity of a logical link like a DOCSIS SF in the case of a CMTS or CM) times the RTT currently being experienced by the TCP connection. This BDP value is usually provided in units of bytes (since bandwidth x latency = [bytes/second] x [seconds] = [bytes]). The rationale behind this age-old belief is that a TCP session should never need to transmit at a rate higher than the bandwidth capacity of the lowest-capacity link in its unidirectional path. Since each network element cannot very easily determine that path-wide bandwidth capacity value, each network element assumes (perhaps incorrectly) that its egress port is the lowest-capacity link in the TCP path, and it assumes that it should (in theory) set its own buffer depth assuming that its own egress link bandwidth capacity (Maximum Sustained Traffic Rate or Tmax for the SF) will be the highest transmission rate that the TCP session will ever have to support. A TCP session with a RTT would climb to a rate of Rmax if it can place BDP = Tmax*RTT bytes onto the network within a window of time equal to RTT. As a result, Gettys argues that BDP = Tmax*RTT bytes would be the maximum number of bytes that any network element buffer would have to absorb without dropping packets for the TCP session, so the argument is that this is the optimal buffer size for that egress port on the network element. Gettys correctly points out that it is difficult to determine a single BDP value that is adequate for all TCP sessions passing through a network element, because the value can actually be quite different for

different TCP sessions due to different RTT latencies that result from different distances that might exist between TCP sources and receivers on different TCP sessions.

Gettys's paper alludes to the fact that there is a sensitive trade-off between setting buffer depths too small (resulting in excessive packet loss and TCP bandwidth reduction- especially for TCP sessions with long RTTs) and setting buffer depths too large (resulting in excessive packet latencies). This leads to interesting issues when two or more TCP sessions are sharing a SF (and a buffer). Ideally, all of the X TCP sessions sharing the SF and buffer would have the same RTT value for their paths and would have the same bandwidth needs- then each of the X TCP sessions would receive ~1/X of the total Tmax value for the SF and would consume ~1/X of the shared buffer. In essence, one can think of each session as having an effective bandwidth of Tmax/X. However, things get more complicated when the X TCP sessions do not have the same RTT values. For example, assume that X=2 and that one session is experiencing a 100 msec RTT while the other session is experiencing a 10 msec RTT. The ideal buffer size for the first session would be ten times the size of the ideal buffer size for the second session. Since only one buffer size can be specified for the single shared buffer, this creates a dilemma that Bufferbloat researchers still need to resolve.

This paper will attempt to answer some of the above questions with extensions to the work done by Gettys and White. We will attempt to include details on buffer memory behavior as we explore the delicate inter-play between packet stream latency, packet stream loss, packet stream bandwidth, and packet stream stability. The goal is to find good guidelines and/or technologies that can be utilized by MSOs as they manage buffer memory sizes and attempt to provide good Quality of Experience (QoE) to all of the application types propagating through their DOCSIS networks.

To assist with the organization of this paper, the authors found it beneficial to divide the variously proposed Bufferbloat management techniques into a taxonomy containing at least three fundamental Bufferbloat management approaches. Within this document, we will call these Bufferbloat management techniques the Type A Management Techniques, the Type B Management Techniques, and the Type C Management Techniques. The taxonomy is illustrated in **Figure 1a**.

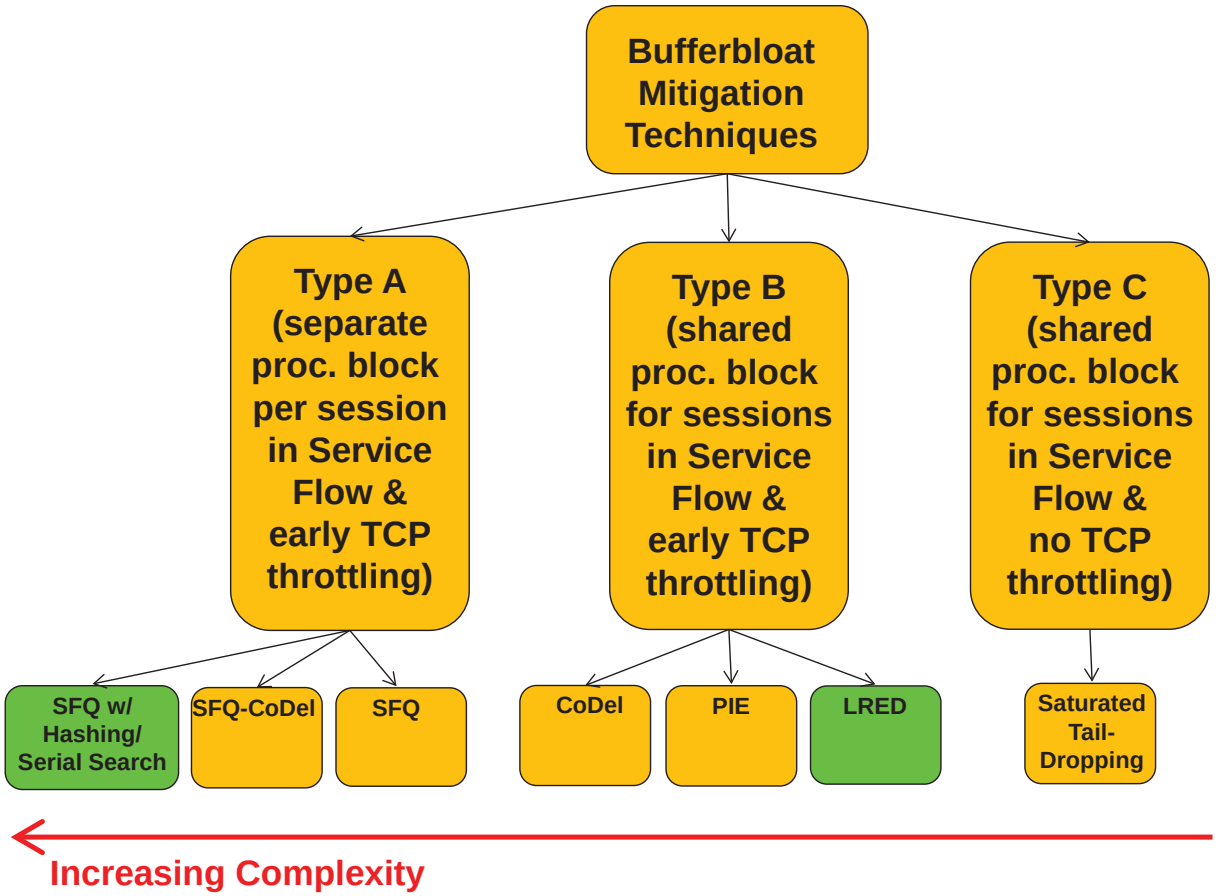


Figure 1a Taxonomy Of Different Bufferbloat Management Techniques

Type A Bufferbloat Management requires that each of the sessions that share a SF must be separated into discrete processing blocks (ex: multiple queues), where each session is processed separately by a different processing block. For approaches discussed in this paper, SFQ-CoDel, Normal SFQ, and SFQ with Hashing/Serial Searching all can be mapped into the Type A Management grouping. These approaches tend to be more complex than their Type B and Type C counterparts, but if designed correctly, they can yield incredibly good results.

Type B Bufferbloat Management requires that the sessions that share a SF must be treated as an aggregate by a single processing block (ex: within a single queue), where all of the sessions experience similar treatment and similar performance levels. However, by definition, we require that Type B Management approaches must include some type of algorithm that makes an attempt at dropping packets from the queue prior to the time when the buffer memory is saturated (filled up). In general, Type B Management approaches try to minimize latency for all sessions to ensure that the latency-sensitive sessions experience low latency. This is often done at the expense of the throughput or loss of packets for bandwidth-intensive sessions. For approaches discussed in this paper, CoDel, and PIE and LRED can all be mapped into the Type B

Management Grouping. These Type B approaches tend to be more complex than their Type C counterparts, and their results are usually much better than the Type C systems. These Type B approaches also tend to be less complex than their Type A counterparts, and their results are not always as good as the Type A systems- for the following reasons:

a. These Type B approaches tend to unnecessarily limit latency on BW-sensitive flows (e.g., File Transfer Protocol or FTP) which tolerate latency very well.
b. Latency limiting always comes at the expense of packet loss and TCP throughput, which can becomes intolerable when attempting to achieve very low latencies.

Type C Bufferbloat Management is the simplest type of Bufferbloat Management system, and it requires that the sessions that share a SF must be treated as an aggregate by a single processing block (ex: within a single queue), where all of the sessions experience similar treatment and similar performance levels. (Note: This is similar to the Type B systems). However, by definition, we require that Type C Management approaches do NOT have any type of algorithm that makes an attempt at dropping packets from the queue prior to the time when the buffer memory is saturated (filled up). As a result, simple Saturated Tail-Dropping is the primary method that would be mapped into the Type C Management Grouping. In general, Type C Management approaches do not perform as well as their Type A and Type B counterparts.

Simulation Baseline

In order to provide a baseline simulation that can be used in comparing potential solutions to the Bufferbloat problem we used the simulated network configuration shown in **Figure 1b**. This model entails a single user who is concurrently uploading large amounts of FTP data to N independent network data servers and simultaneously playing an interactive game that is sensitive to upstream latency. (Each FTP session transmits a continuous stream of 90 MB FTP PUTs with 4 second spacing between blocks. The gaming session is modeled as a TCP session that transmits 10 KB messages at 2 second intervals.) While this configuration is framed to model traffic in the upstream direction, the same analysis applies to the downstream direction (where the buffer would be incorporated into the CMTS).

We have intentionally kept the mixture of traffic types (e.g., data rates, RTTs, service types, protocols) as small as possible. This makes it possible to create a simulation series that smoothly varies just one of these parameters and to chart performance as a function of that parameter. A highly complex traffic mix would defeat this goal.

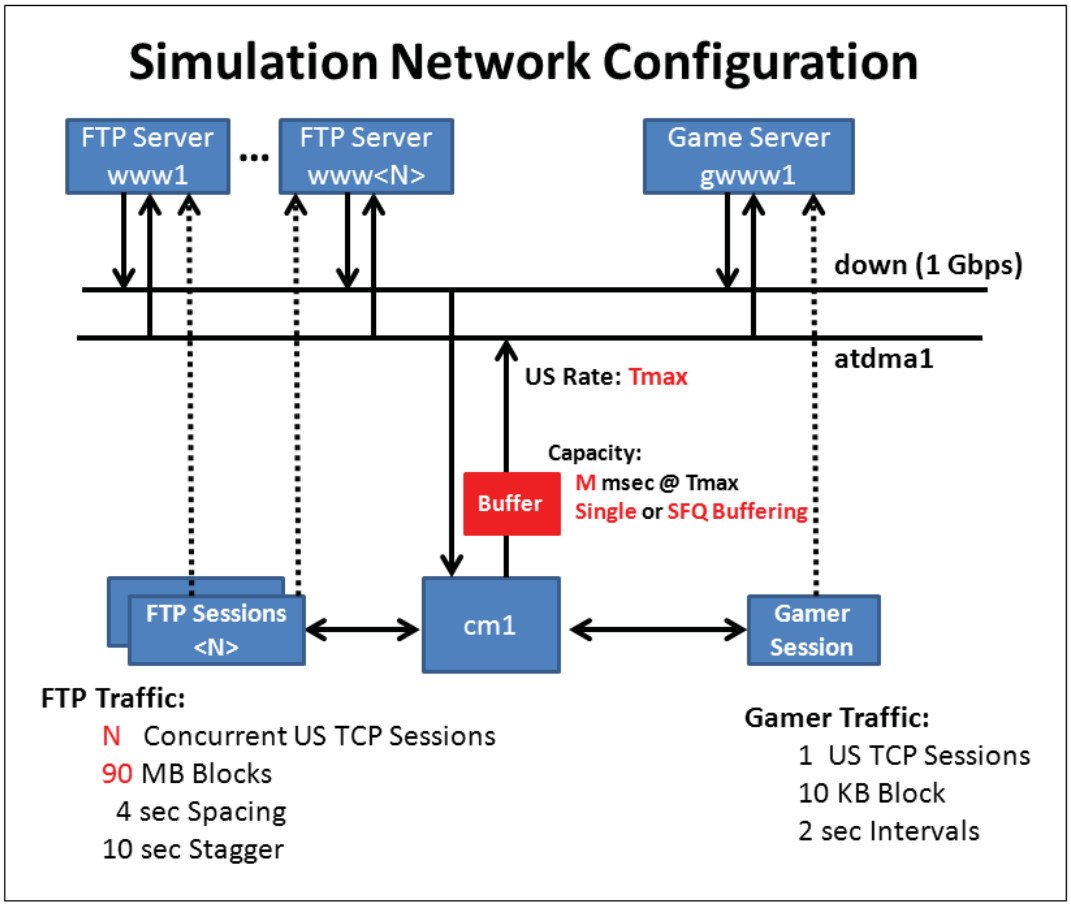


Figure 1b Description of Simulation Network

The upstream buffer used in this baseline model is a simple First-In-First-Out (FIFO) packet buffer with <M> KB of available memory. Buffer overflows are handled by dropping incoming packets when no more buffer storage space is available. This buffer model matches most upstream buffers in use today and should accurately demonstrate the phenomenon of Bufferbloat as it exists in many places in today's network.
Our simulation environment consisted of a packet level traffic simulator with microsecond level accuracy. TCP, FTP protocols and CMTS behavior (when used) were modeled in detail.

Our goal for the baseline simulation was to demonstrate how FTP upload data rate, gamer upstream latency, and overall data loss will vary as a function of the upstream buffer size. This was accomplished by repeating each simulation experiment 20 times while varying only the size of the upstream buffer. By charting the parameters of interest (e.g, data rate, latency, drop rate) versus the amount of physical buffering available we can then show, not just a disconnected set of values, but how they vary smoothly with buffer size. Thus we can show the performance of both large and small buffers (including those in between) in a single understandable chart.

Rather than describing the size of the upstream buffer in bytes, as might seem most logical, we have chosen instead to use the amount of time required to empty a full buffer at the maximum data rate (Tmax). This choice of scale has a normalizing effect on the resulting graphical displays and allows us to more easily compare the results of experiments involving different upstream data rates (Tmax). Converting between the two metrics is fairly straight-forward. To convert from the time metric (in seconds) to a buffer depth metric (in bytes), simply multiply the time metric (in seconds) by Tmax/8 to produce the buffer depth metric (in bytes)

Our expectation is that the upstream packets from the gamer session will become queued behind a long standing queue of FTP session packets resulting in the gamer session experiencing unacceptable amounts of upstream latency. We would expect larger upstream buffers to permit longer FTP standing queues resulting in even larger amounts of added upstream gamer latency. Smaller upstream buffers, on the other hand, will experience many more packet drops and data retransmissions.

Type C Extension: Exploring Mismatches Between Actual BDP Values & Pre-Configured Buffer Depths With Saturated Tail-Drop Queues

At the current moment in history, MSOs will likely be limited (for a while) to using the Buffer Control mechanisms of DOCSIS 3.0 to try to manage upstream Bufferbloat, because other more advanced Bufferbloat management techniques (such as SFQ or CoDel or PIE) have not typically been implemented in many modems to date. As a result, most MSOs will be relying on the Saturated Tail-Drop mechanism to force throttling of TCP sessions when active buffer depths grow to be too large. This mechanism, which simply drops packets entering the tail of the queue whenever the buffer is full, is illustrated in **Figure 1c**.

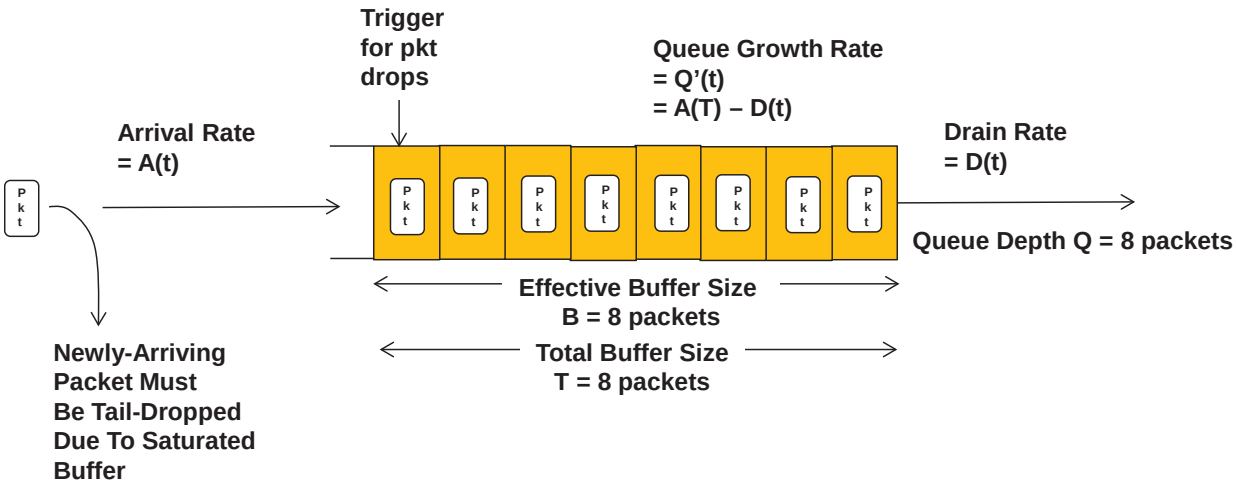


Figure 1c Simple Operation Of Saturated Tail-Dropping Within A Queue

When MSOs pre-configure the buffer depths for SFs using the Buffer Control mechanisms that are now available in DOCSIS 3.0 modems, they can change the Effective Buffer Size for each SF, but they will be forced to make some predictions and guesses about the best buffer depths that will minimize their Bufferbloat problems, minimize packet drops, and ensure high TCP throughputs. Gettys paper proposed that they set the buffer depths of their queues to be equal to the BDP of the packets passing through the network. To perform that BDP calculation, the MSO would have to select an appropriate Bandwidth value and an appropriate Delay value that they expect the packets to experience. The Bandwidth value in the BDP can be set to the Tmax setting for the SF. (Note: As described in an earlier section of the paper, this works even if multiple sessions are sharing the Bandwidth of the SF). But the Delay value in the BDP would probably need to be a guestimate of the "typical" Round-Trip delays that might be expected for the packets passing through the SF. Typical guestimates might choose to use an average of the following rough RTT measurements made on the Internet [Kras]:

- Typical RTT values between sites in the same region: ~20 msec
- Typical RTT values between sites on the same continent: ~100 msec
- Typical RTT values between sites on different continents: ~200 msec.

Unfortunately, there may be multiple sessions (i.e.- packet streams) passing through the SF and through the shared buffer, and each session may be encountering different Round-Trip delays due to propagation through different distances and due to interactions with different congestion levels on the different propagation paths. As a result, the calculated BDP (and buffer depth) cannot be guaranteed to be set to a perfect value that corresponds to ***all*** of the sessions passing through that SF's buffer. We will now explore some of the repercussions of having a buffer depth setting that does not exactly match the actual BDP of a particular session.

Figure 2(a & b) shows the results of a series of simulations in which a gamer session plus 10 FTP sessions were competing for bandwidth over a 5 Mbps upstream service flow with an uncongested RTT value of 50 msec. In **Figure 2a** we can see that the latency-sensitive gamer session experiences virtually the same latency as each of the FTP sessions for which latency is of no consequence. This is a direct result of the fact that the sessions are all sharing the same buffer. We also see that the latency experienced by each of the sessions increases linearly with the size of the upstream buffer. This would tend to imply that MSOs might be wise to select smaller buffers. But how small is too small?

To answer that question, one must also consider packet loss and throughputs, which are captured as a function of the buffer depth in **Figure 2b**. In **Figure 2b,** we can see that the total upstream data rate is nearly constant for all buffer sizes (with the exception of very small upstream buffers). The minimum buffer size widely regarded as necessary to support adequate throughput is known as the BDP and calculated as the product of the SF's maximum upstream Data Rate (Tmax) times the Network RTT. In

this simulation series RTT is equal to 50 msec, and that particular buffer depth that drains in 50 msec (when drained by the Tmax value) is identified by the vertical, red, dashed line on the plot in **Figure 2b**. It can be seen that (as Gettys predicted) buffer depths smaller than this critical size (to the left of the vertical, red, dashed line) start to produce marked reductions in FTP bandwidth and also start to produce marked increases in packet loss. This is primarily due to the fact that TCP sessions will likely attempt to send more packets through the small buffer than can be stored by the small buffer, resulting in buffer overflows and packet drops which ultimately force TCP to throttle its bandwidth.

We have also found that simulation runs with a large number of FTP sessions sharing the buffer produce similar aggregate throughput curves as simulation runs with a small number of FTP sessions sharing the buffer. The throughput is shared by the FTP sessions, and the knee in the curve to the left of the BDP value always occurs at roughly the same buffer size (given by the BDP). Thus, one can conclude that the minimum buffer size suitable for an application does not depend on the number of concurrent sessions sharing a SF.

Also in **Figure 2(b)**, the lower trace shows the data loss (measured in Mbps) resulting from packets dropped due to buffer overflows. While smaller upstream buffers experience more data loss than larger upstream buffers, the amount of data loss experienced in this experiment is not particularly alarming. However, its corresponding impact on TCP throughput performance is quite alarming when the buffers are made too small (i.e..- smaller than the BDP value).

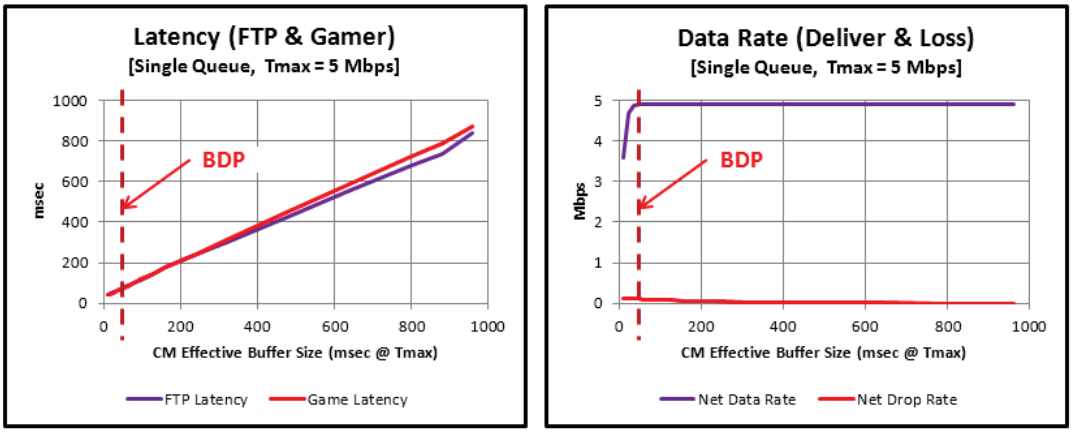


Figure 2(a & b) Latency & Throughput (Tmax = 5 Mbps)

Figure 3(a & b) shows the results of a similar series of simulations for which the upstream SF rate (Tmax) was limited to 1 Mbps. Notice that **Figures 2 and 3** look very similar (as a result of our choice of units for the X axis scaling) even though the corresponding buffer sizes in **Figure 2** are (due to their different Tmax values) five times as large as those in **Figure 3**.

We can also see that the poor throughput for buffers smaller than the BDP value is even more pronounced for the low speed SFs (with low Tmax values) in **Figure 3b**. This indicates that MSOs should be wary of operating with very small buffers when the SF Tmax values are low, because they risk placing limits on the maximum throughputs of TCP sessions that utilize those buffers.

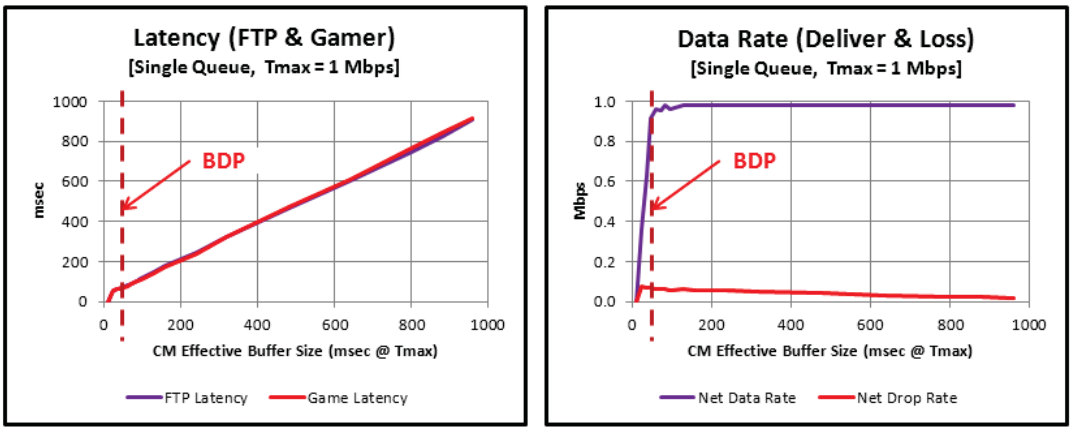


Figure 3(a & b) Latency & Throughput (Tmax = 1 Mbps)

Just as concerning is the significantly high data loss rate predicted by **Figure 3b** for low speed (e.g, 1 Mbps) upstream SFs when the buffer sizes are made small. For upstream buffers able to store only 200 msec of data the data loss rate is nearly 6% of maximum channel rate. This could be especially damaging in the downstream direction where this dropped data has already made a complete transit of the Internet and must now be retransmitted from its source.

It appears that any plan to manage Bufferbloat and latency purely through buffer size ought to pay careful attention to the data loss rate and the maximum TCP throughputs for low speed SFs. Large buffer sizes are helpful in keeping packet losses low and TCP throughputs high, but they can suffer from larger latencies on the latency-sensitive sessions. Conversely, small buffer sizes are helpful in keeping latencies low on latency-sensitive sessions, but they can suffer from high packet losses and low TCP throughputs. Careful adjustments must be made- probably to values to that slightly larger than the BDP value.

Once the buffer size has been selected for a SF based on a configured Tmax value and a predicted RTT, one may wonder what happens if the actual RTTs for a traversing packet stream are different from the predicted RTT.

If the actual RTT is much shorter than the prediction, then the selected buffer depth would be larger than it needs to be, implying that the Effective Buffer Size is larger than the actual BDP and the system would be operating on the curves of **Figure 2** and

Figure 3 to the right of the BDP value. This would result in larger-than-necessary latencies (but would yield low packet drop rates and high TCP throughputs).

If, on the other hand, the actual RTT is much longer than the prediction, then the selected buffer depth would be smaller than it needs to be, implying that the Effective Buffer Size is smaller than the actual BDP and the system would be operating on the curves of **Figure 2** and **Figure 3** to the left of the BDP value. This would result in larger-than-necessary packet drop rates and lower TCP throughputs (but would yield lower latencies).

Type A Extension: Stochastic Flow Queuing with Hashing/Serial Searching To Reduce Hash Collisions

Generic Operation and Analysis

In this section we will examine the effect that the use of a Stochastic Flow Queuing (SFQ) approach might have upon the Bufferbloat issue. We will demonstrate both the strength of this mechanism in the absence of hash collisions and the detrimental effects that hash collisions might have. We will then describe a technique that could greatly reduce the probability of hash collisions within SFQ systems.

By definition, packets from a single "flow" (or sub-flow or session) share a common 5-tuple- i.e., they all have the same Ethertype, IP Source Address, IP Destination Address, TCP/UDP Source Port, and TCP/UDP Destination Port within their headers, and they are assumed to be associated with a single TCP or UDP session. It is important to understand the difference between a session and a SF, because there is often confusion around the two concepts. SFs can have many sessions (and packet streams with many different 5-tuples) mapped into them, but a session has one and only one 5-tuple associated with it.

Since packets from a single session share the same 5-tuple, one can use these five fields in the header of the packets to help separate them into separate and discrete sessions. Hashing is one quick way to map the packets into their designated session in real time. Hashing is a combinatorial combination (ex: Exclusive OR) of the five headers that produces a Y-bit result that points to one of 2^Y unique sessions.

In a network element designed with separate buffers for each session, this hashing function can be used to rapidly steer each packet into its appropriate buffer (as shown in **Figure 4**). A Round-Robin Scheduler can then be used to extract packets from the multiple queues. The resulting SFQ approach may be considered to be more expensive and complex (from a hardware point-of-view) than other Bufferbloat mitigation

techniques, but the ability of this approach to mitigate Bufferbloat issues is stellar (as shown in the CableLabs paper). We will therefore consider this approach in more detail within this paper.

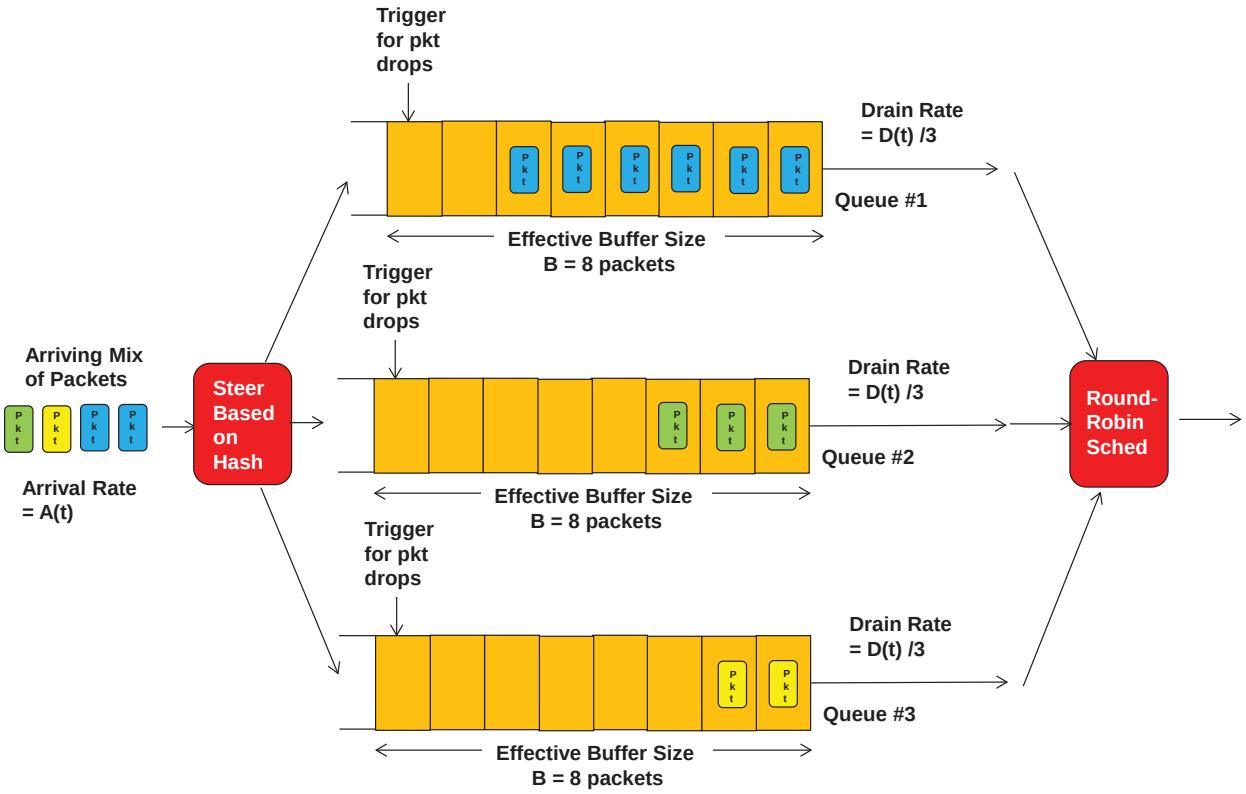


Figure 4 Basic Operation Of SFQ

The packet sequencing challenge that leads to the Bufferbloat problem has much in common with the issues of congestion control among DOCSIS SFs. In both cases several functionally unrelated packet streams compete for a limited amount of bandwidth that is insufficient to handle the full amount of offered traffic.

The issue of congestion control among DOCSIS SFs, however, has been successfully addressed by CMTS manufacturers and no counterpart of the Bufferbloat problem seems to exist between DOCSIS SFs. Since the problem seems to be satisfactorily addressed in the DOCSIS domain, it might be wise to examine the parallels between these two problem domains to see what we might learn from DOCSIS congestion control that might be applied to the Bufferbloat issue.

As mentioned previously, a somewhat complex approach to solving Bufferbloat that has been widely discussed is called SFQ. It turns out that the SFQ mechanism is quite similar to the way that DOCSIS congestion control is implemented in a CMTS. Docsis supports separate queues for SFs in order to allow for Quality of Service functionality.

While this is similar to SFQ in some ways, there are very important differences. This section will attempt to identify similarities and differences between these two mechanisms.

In the existing DOCSIS scheme, packets are first categorized into separate SF's by virtue of a fixed, provisioned rule set (e.g. by a range of IP destinations). Each SF has its own queue, and the set of queues are serviced in a round robin manner (or using other scheduling techniques). The rules by which packets are assigned to queues are not only fixed, but they are mutually exclusive in the sense that each queue will never contain packets from more than one service flow. Because each queue does not interact with the others, a highly loaded queue's contents cannot block or slow the transmission of packets in a different queue. In other words, Bufferbloat effects are absent from a service flow perspective. However within a given service flow, different TCP/IP sessions can interfere with each other if they map to the same packet stream within theSF. These rules are agnostic to the type of application involved with the packets, and therefore in general maintain net-neutrality.

Like the existing DOCSIS SF scheme, SFQ offers separate queues which avoid Bufferbloat between the queues. The mapping of packets into each queue is aimed at avoiding delaying one TCP/IP flow (a session) by another session. In this case, the rule-set by which packets are assigned to queues is dynamic and seeks to put each session into its own queue. If this goal is achieved packets from one flow cannot be impeded by packets from a different flow, making the latencies independent between sessions. This is a highly attractive attribute.

By their nature, Internet sessions dynamically appear and disappear over relatively short periods of time, and in general are not predictable in advance. To implement SFQ, therefore, decisions need to be made to separate sessions by five-tuple and dynamically create (and destroy) queues for each flow.

In practical terms it may not always be possible to have as many queues as there are sessions; if this occurs, then the SFQ system may be forced to temporarily enter an undesirable state where more than one session is assigned to the same queue; in this undesirable case the Bufferbloat phenomenon can exist between sessions. However, by a judicious design choice of the number of allowed queues, this should be a rare case.

The dynamic assignment of packets to different SFQ queues needs to be fast and minimize situations where multiple SFs get assigned to the same queue. DOCSIS service flow classification is less of a challenge since a pre-configured rule-set can be used to create a search index which is not only static, but it also results in never having a queue with more than one SF mapped into it. In the case of SFQ, dynamically created hashing algorithms will be needed which will result in efficient and effective classification. But hashing collision must be minimized. For example, the authors have

considered several alternatives and have identified some combinations of hashes and linear searches which seem appropriate.

In order to be effective in eliminating or minimalizing the Bufferbloat phenomenon, the SFQ implementations must be placed in the CM for upstream packets and in the CMTS for downstream packets. Due to the complexities of these SFQ approaches, it may not always be possible to include these functions in the CMs and CMTSs.

Figure 5(a & b) shows the results of a simulation series that was identical to the situation modeled in **Figure 2** with the single exception that the upstream buffer separated each upstream TCP session onto its own queue. These queues were then drained in a simple packet round-robin fashion.

Figure 5a shows that this has resulted in identical packet latency for all FTP sessions (for which latency is of no concern), but the separation of sessions into different queues within the SFQ arrangement has almost completely eliminated any additional latency on the gaming session (resulting in only the 50 msec of round-trip latency that is inherent in the uncongested network layout). **Figure 5b** shows that both the overall data throughput and packet loss rates are virtually identical to the baseline experiment in **Figure 2b**. We can see here that (in cases where we were able to guarantee a total absence of hashing collisions) the use of SFQ buffering can completely eliminate Bufferbloat, resulting in an overall performance equivalent to that of DOCSIS congestion control.

It should be apparent that the MSO is free to be more cavalier in their selection of buffer sizes when SFQ is used. Since they don't know the exact BDP values for the actual packets, they can lean towards the use of larger BDP values (like 200 msec or so) and not incur the penalty of longer delays for the latency-sensitive sessions (since those sessions will be placed on their own queue and typically experience low latencies in a fashion similar to that of the gaming session in **Figure 4a**.

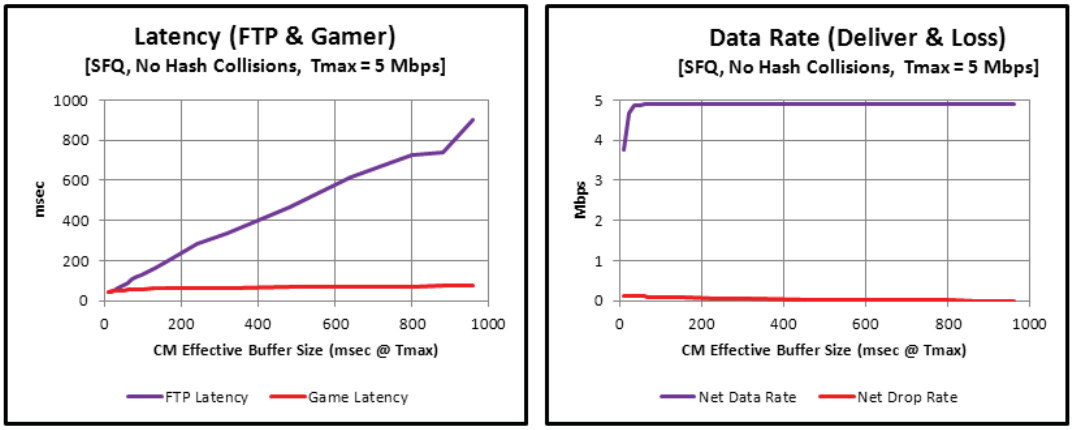


Figure 5(a & b) Latency & Throughput (SFQ w/o Hash Collisions)

If the MSO chooses longer buffer depths in an SFQ environment, then the sessions with heavy bandwidth (such as the FTP sessions above) will experience longer latencies as they fill the larger buffers (as shown in **Figure 5a**), but this additional latency on any FTP session is, in this case, due only to previously queued packets for that *same* FTP session and should not be viewed as a problem. (Note: The FTP packets are being delivered from source to destination as quickly as the network will permit. Whether the packets from a particular FTP session are stored at the source or at the network element buffer is unimportant). For this reason the throughput of each FTP session actually improves slightly (but with quickly diminishing returns) as the total buffer size increases due to a need to drop fewer packets in order to prevent buffer overflow. As shown, the longer buffers do not typically lead to longer latencies for the latency-senstive sessions (such as the gamer session), because most latency-sensitive applications (gaming, VoIP, etc.) do not typically generate enough bursty bandwidth to fill their isolated SFQ buffer. Web-browsing is an interesting latency-sensitive application that can generate enough bursty bandwidth to fill its isolated SFQ buffer, but as with the FTP sessions, the bandwidth will be delivered as fast as the link will permit.

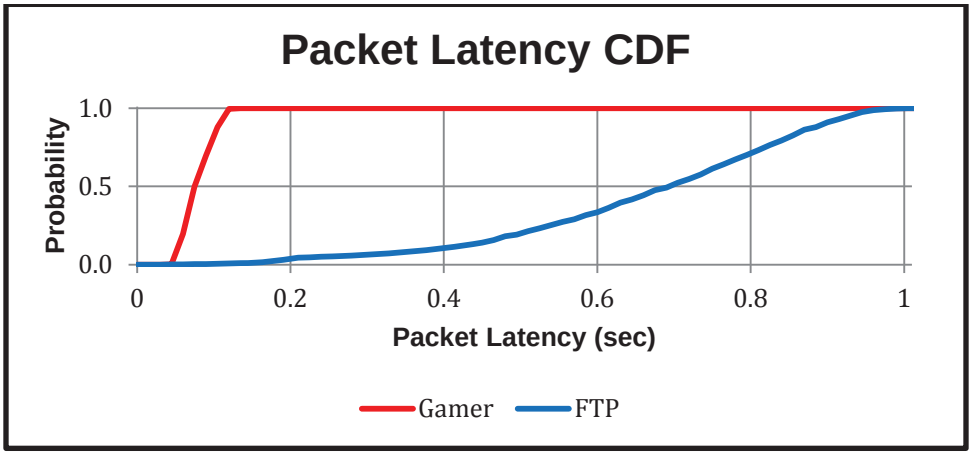


Figure 6 Latency CDF (Gamer in red & FTP in blue) for SFQ

Figure 6 shows the Cumulative Distribution Function (CDF) for all packet latencies for both gamer traffic (in red) and FTP sessions (in green) flowing through an SFQ system with effective buffers sized to be able to store 960 msec of 5 Mbps traffic (i.e.- 600 Kbyte buffers). This chart shows that all gamer packet latencies (in red) are tightly clustered between 40 & 100 msec while most FTP packets experience latencies over a much wider range between 0.5 and 1.0 sec.

This chart shows that all of the individual packet latencies (not just their average value) fall within a desirable range when a well-designed SFQ system is utilized.

Hash Collisions

We now turn our attention to examining how SFQ buffer performance is affected in the event of a hashing collision. Hash collisions cause problems that occur when two (or more) different sessions map to the same identical hash value and therefore have their packet streams steered to the same identical SFQ queue. This condition essentially breaks the perfect isolation that previously existed between sessions passing through the different SFQ queues. **Figure 7(a & b)** shows the results of a simulation series where hash collisions have occurred.

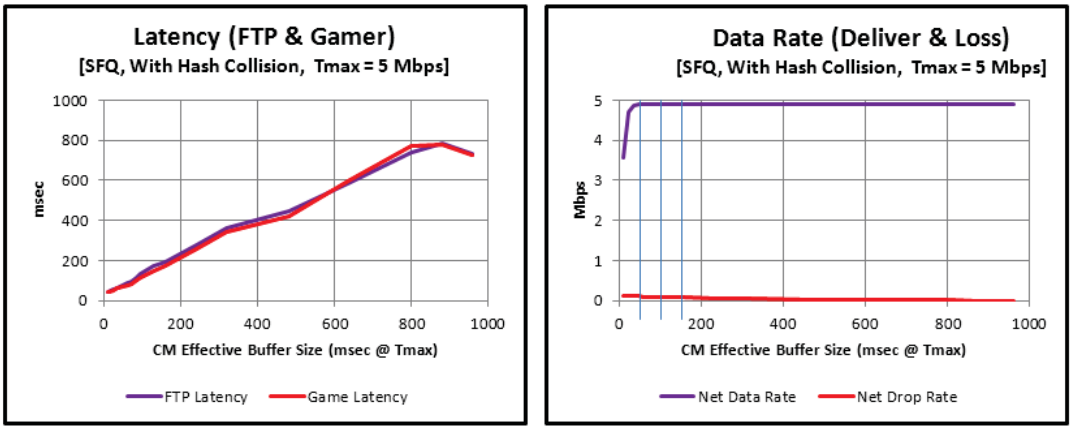


Figure 7(a & b) Latency & Throughput (SFQ w/ Hash Collision)

Thus, **Figure 7** is identical to **Figure 5** with the exception that a single hash collision has been purposely forced between the gamer session and one of the FTP sessions. It is apparent from **Figure 7a** that the gamer session now experiences the same amount of Bufferbloat that was present in the baseline simulation series, because the packets incur the latency of a single FTP stream and packet drops to throttle the FTP stream do not occur until Saturated Tail-Dropping occurs. In effect, the previous benefits of SFQ buffering are completely undone in the event of a hashing collision involving a latency-sensitive session.

This might seem counterintuitive since the gamer is now sharing a hash queue with only one of the ten FTP sessions instead of with all 10 FTP sessions (as is the case in the baseline series). However, the difference here is that the single SFQ that the gamer shares in this case is draining at only 1/10th the rate of the baseline queue – so the net result is the same.

In summary, these two sets of simulations show two important things:

1. In the absence of hash collisions, SFQ buffering is excellent at mitigating the Bufferbloat problem and is fully as good as the current DOCSIS congestion control algorithms (and shows no signs of Bufferbloat).
2. In the presence of hash collisions involving latency-sensitive sessions, SFQ shows no improvement at all over standard Saturated Tail-Dropping queue– but also no further degradation, either.

Both the excellent performance of SFQ buffering and its vulnerability to infrequent hash collisions has been previously observed in the literature, and a combined approach (SFQ-CoDel) has been suggested [Whi2]. This combined approach proposes the use of an additional CoDel mechanism on top of the normal SFQ operation to help partially minimize the large latencies that can result in the event of a hash collision involving latency-sensitive sessions. But the performance results of the combined approach are not as astounding as the results experienced by a standard SFQ system that avoids hash collisions altogether. As a result, we will explore (below) ways to greatly decrease the probability of hash collisions within standard SFQ systems.

Decreasing Hash Collision Probability

In this section we will look specifically at the likelihood that one of these hash collisions might occur. Instead of simply assuming that these collisions are always going to happen, what if we could design a mechanism for which the probability of a hashing collision were so extremely rare that it could reasonably be ignored. (Note: The reader should remember that even in the worst-case scenario with SFQ experiencing a hash collision, the latency would never be greater than what we currently see every day in the field with standard Saturated Tail-Dropping). If such a low-collision hashing mechanism could be found, then SFQ buffering would not require a second mechanism, like CoDel, to cover for the effects of buffer collisions. (Or it could add the second mechanism and rarely rely on it, so that the SFQ system could typically experience the higher-level performance offered by normal SFQ operation).

It turns out that the harder problem here is the challenge of simply calculating the collision probability rather than how to design the hashing mechanism. Many (but certainly not all) commonly used hashing mechanisms actually already have extremely low collision probabilities – but it is not a simple matter to calculate that probability numerically.

In the CableLabs paper, it was assumed that a simple hashing algorithm was used (by itself) for selecting into which queue a session should be mapped. For example, if each service flow was provided with a generous number of (say) Q=256 SFQ queues into which one could separate the sessions propagating through that service flow, a simple hash would blindly steer the packets to the queue to which they originally hashed (assuming the use of an 8-bit hash pointer). This approach falls victim to the well-known "Birthday Paradox," whereby there is a high probability that two sessions will experience a hash collision- even within the 256 SFQ queues and even with a small number of sessions sharing the service flow's 256 SFQ queues. In general, the probability that the first session that grabs a queue experiences no hash collision is P(1)=1. The probability that the second session that grabs a queue experiences no hash collision is P(2)=255/256, and the probability that neither the first or second session experience a hash collision is given by P(1)*P(2) = 1*(255/256) = 99.6%. The probability that the third session that grabs a queue experiences no hash collision is P(3)=254/256, and the

probability that neither the first or second or third session experiences a hash collision is given by P(1)*P(2)*P(3) = 1*(255/256)*(254/256) = 98.8%. This process can be continued to calculate the probability of a hash collision whenever X sessions are sharing a service group, and the results are plotted in **Figure 8**. As can be seen in the figure, the probability of a hash collision rises rapidly as the number of sessions sharing a service flow grows. With as few as 10 sessions, the probability of a hash collision within the 256 queues rises above 16% when normal hashing is used! With as few as 20 sessions, the probability of a hash collision within the 256 queues rises above 50% when normal hashing is used! These results are horrendous, and as indicated in the CableLabs paper, they imply that SFQ alone does not work well as a Bufferbloat mitigation scheme because of the high probability that hash collisions will return its performance back to the lower performance levels of a Saturated Tail-Dropping system (which probably doesn't justify the added complexity of an SFQ solution).

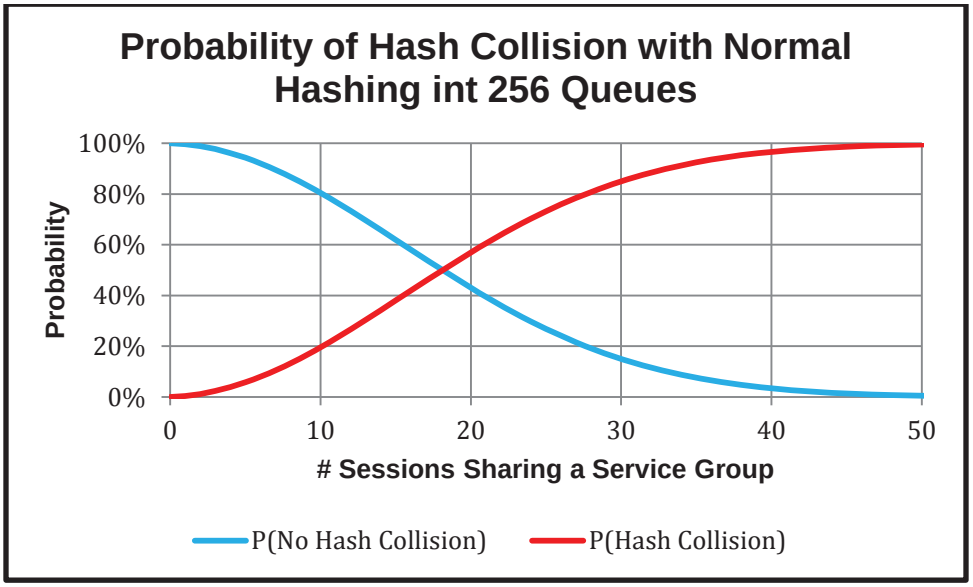


Figure 8 Probability Of Hash Collisions With Normal Hashing Into 256 Queues

In an effort to rectify this inherent problem of SFQ systems that utilize simple hashing for session-to-SFQ queue mapping, we will now propose and analyze the performance of a slightly different system that uses a slightly different algorithm for session-to-SFQ queue mapping.

The new proposal uses a fairly straightforward combination of hashing and serial searches. In the particular system with 256 SFQ queues per Service Flow, the system would hash into one of 64 queue groups, and each queue group would contain four queues. Once a session is mapped into a queue group, a serial search will identify the first unused queue within the four queues of the queue group and then assign that session to that particular unused queue. Since a serial search through four queues is not exceptionally challenging, this was deemed to be somewhat reasonable from a hardware/software complexity point of view. Using this simple modification, up to four

different sessions can randomly hash into the same queue group and still be intelligently steered into different queues, so the probability of a hash collision is greatly reduced through the use of the simple serial search mechanism. Obviously, in the unlikely event that five or more sessions hash into the same queue group, then this modified SFQ system is forced to allow a hash collision to occur and the system must then steer two different sessions into a common queue. But we can perform Monte-Carlo simulation techniques to calculate the probability of experiencing a hash collision as a function of the number of queue groups in the hash table and the number of queues in the queue group.

The occurrence of a collision for the modified SFQ design with 64 queue groups and 4 queues per queue group would require that 5 (=4+1) different sessions hash to the same queue group before a hash collision would occur. Since we were unable to calculate a closed form solution for calculating the probability of such an event, we resort to the use of Monte Carol simulation techniques. **Figure 9** shows the results of this simulation, with the x-axis showing the number of sessions that are sharing the service flow and with the y-axis showing the probability of a hash collision.

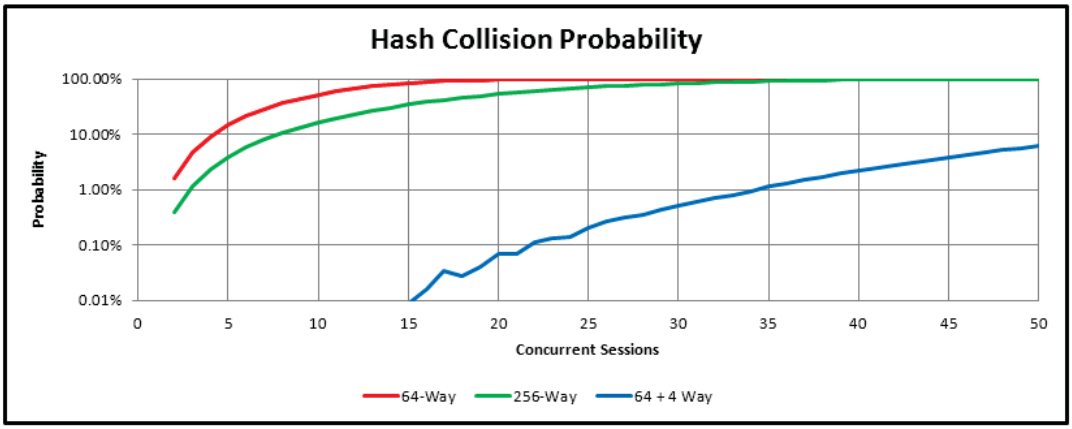


Figure 9 Hash Collision Probability With 64 Queue Groups And 4 Queues Per Queue Group (256 Queues Total)

Within **Figure 9**, the green plot shows the probability of a hash collision if we perform normal hashing into the 256 individual queues. The red plot shows the probability of a hash collision if we perform normal hashing into the 64 queue groups and register a hash collision any time that two or more sessions map to the same queue group (even though this would not result in a hash collision with our added serial search mechanism that is described by the blue plot). The blue plot shows the probability of hash collision if we perform normal hashing into the 64 queue groups and then perform a serial search for an unused queue within the 4 queues inside of the queue group. From the blue plot, we can see that even as many as 20 concurrent sessions would have only a 0.05% chance of experiencing a hash collision! Even as many as 35 concurrent sessions would have only a 1% likelihood of having any hash collision at all.

Thus, the addition of this proposed serial search mechanism to the simple hashing techniques normally used in SFQ systems provides great reductions in the probability of SFQ hash collisions. This is even true when Peer-to-Peer services (such as BitTorrent) are being utilized by users within the home. BitTorrent experiments that were performed by the authors on the Internet indicated that most of our observed BitTorrent sessions had up to 50 peers, but typically had only 10 (or so) concurrent TCP sessions that were actively exchanging data with those peers. While this is not full validation, it does give a good indication that the improved latency performance of SFQ with Hashing/Serial Searching can likely provide excellent performance for most of the use cases that will be found in most homes.

It is worth remembering that most of the potential (but unlikely) hash collisions within an SFQ system with Hashing/Serial Searching would still not degrade service. It is only when a latency-sensitive session actually collides with a very high throughput session that any Bufferbloat issues would result.

While not all hashing mechanism designs will produce the collision-free performance that we have shown in this example, many variations on the above design do exist which would produce excellent results even with very large numbers of concurrent sessions. It may well be the case that a straightforward SFQ buffer, by itself, could completely eradicate Bufferbloat (even without the addition of secondary mechanisms like CoDel). Nevertheless, one could easily add a secondary mechanism (such as CoDel or the LRED algorithms proposed in the next section) on top of SFQ with Hashing/Serial Searching to produce a very powerful Bufferbloat mitigation system.

Type B Extension: Latency-Based Random Early Detection (LRED) For Managing Bufferbloat

Generic Operation and Analysis

The SFQ Bufferbloat mitigation technique that employed both hashing and serial searches (as described in the previous section) uses a Type A approach that definitely provides very good control of Bufferbloat for latency-sensitive sessions while also yielding low packet loss and high TCP throughputs. However, it is possible that some network elements may not have the hardware and software resources to implement the higher complexity levels required by an SFQ solution. As a result, the authors wanted to also explore simpler (medium-level complexity) Bufferbloat mitigation techniques that fell into the Type B camp.

In this section, we will propose and analyze a novel Bufferbloat mitigation technique that would probably be categorized as a Type B approach. It shares a lot of similarities with

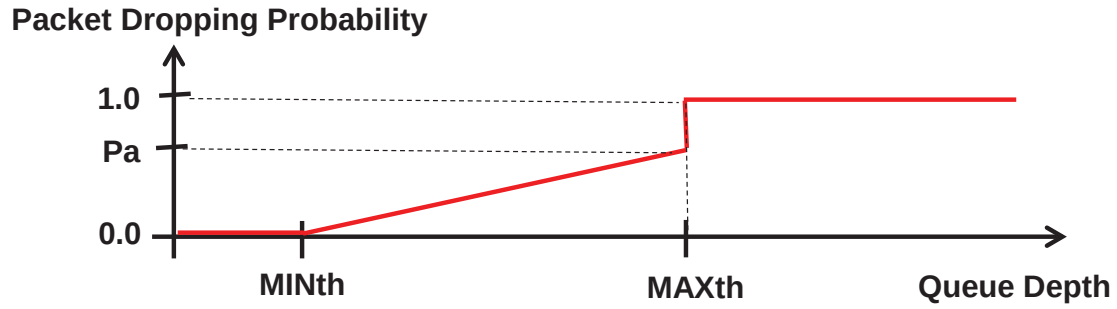
the CoDel and PIE approaches. CoDel and PIE are both very clever techniques that have been shown to provide good results in managing Bufferbloat. But we began to wonder if there wasn't a slightly simpler approach to the packet dropping algorithms that could piggy-back on top of and capitalize on some congestion control algorithms that might already be present within certain network elements.

This led the authors to consider the use of the well-known Random Early Detection (RED) algorithms or Weighted Random Early Detection (WRED) algorithms that are often implemented in one form or another within network elements. [Floy] Normal RED algorithms adjust the dropping probability of packets passing through a network queue as a function of the current queue depth (or a queue depth average). Normal WRED algorithms add packet priority to the mix. In general, packets with lower priority and packets in longer queues will experience higher dropping probabilities in an attempt to throttle TCP flows.

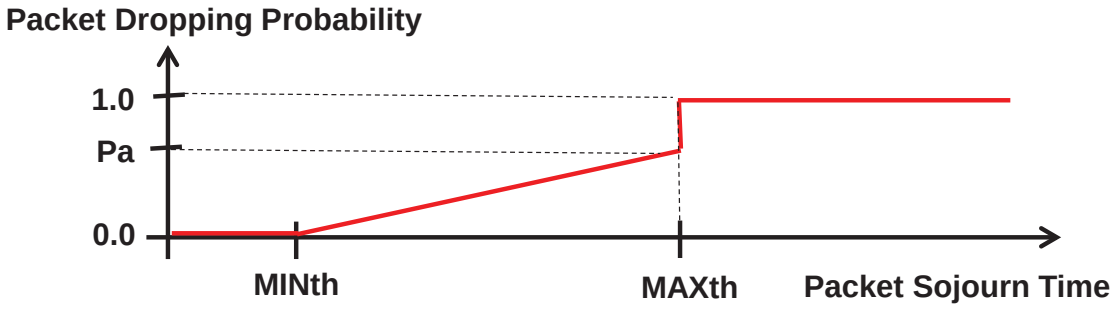
These algorithms are usually included in network elements to manage congestion control within the networks by inserting random packet drops to throttle TCP sessions in order to avoid buffer overflow. These algorithms are typically based on special dropping probability curves that are used to determine the probability with which each packet passing through a queue should be dropped. The standard shape of the RED dropping probability curve is shown in **Figure 10(a and b)**, and it is described by three distinct parameters (the minimum threshold MINth, the maximum threshold MAXth, and the dropping probability Pa at MAXth). Different configured values of MINth, MAXth, and Pa can yield quite different dropping probabilities for the packets passing through the queues as network congestion and queue depths change. (Note: **Figure 10a** uses the traditional queue depths and **Figure 10b** is a non-traditional version that uses packet sojourn times. Either real-time or average queue depths and sojourn times can be and have been utilized in different implementations).

Queue depth (**Figure 10a**) and packet latency (**Figure 10b**) are closely related in Type B mechanisms, where **Depth = Latency * Tmax / 8**. One advantage of using packet latency instead of queue depth for RED calculations is that the "effective Tmax" that is experienced under heavy network congestion may be significantly lower than the configured Tmax. Mechanisms based on latency will automatically compensate for this change in Tmax, where mechanisms based on queue depth will not. For Type A mechanisms this difference is even more important since every sub-queue has a different length and effective Tmax. A latency-based RED mechanism requires an aditional overhead of saving a simple packet timestamp on entry to the queue.

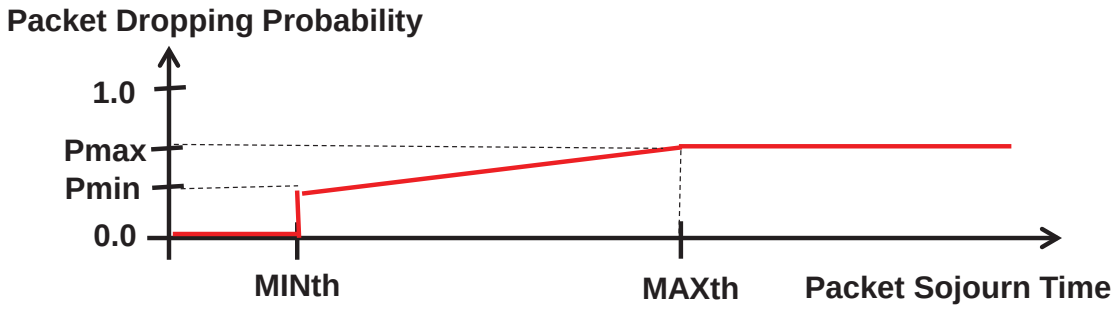
If this RED or WRED mechanism were already implemented within a network element (or could be easily added), we wondered if we could use some of the sub-systems from RED or WRED congestion control algorithms to help create a simple Type B Bufferbloat management system.



a) Dropping Probability Curve As A Function of Queue Depth



b) Dropping Probability Curve As A Function of Sojourn Time



c) Modified Dropping Probability Curve As A Function of Sojourn Time

Figure 10(a & b &c) Typical Drop Probability Curves for RED & WRED

Thus, in this section we introduce a novel Bufferbloat management technique called Latency-Based Random Early Detection (LRED). LRED adds a slight modification to RED in that it adjusts the dropping probability as a function of the latency experienced by packets as they exit the queue.

There are several ways to implement this solution. One of which is to modify a single First-In-First-Out (FIFO) buffer queue by time stamping each packet on arrival. Upon retrieving a packet from the head of the queue we compare the length of time that the

packet has remained in the queue (i.e.- the packet's sojourn time). Depending on the magnitude of the sojourn time, the system calculates a dropping probability value between 0.0 and 1.0 for this packet by accessing the dropping probability curve shown in **Figure 10b.** Using this dropping probability value, a random number generator is then used to determine a uniformly distributed random variable between 0.0 and 1.0. If the random variable is less than the dropping probability value, then the packet is dropped. Otherwise the packet is passed.

A similar implementation based on queue depth (instead of sojourn time) can also be envisioned, but it would use the dropping probability curve shown in **Figure 10a**. (Note: When using queue depths, LRED and RED become virtually identical). Results are similar with either approach on uncongested networks.

Over time, we have also looked at modifications to the RED dropping probability curves and migrated towards the curve shown in **Figure 10c**. (Note: This particular curve was used for the LRED simulations described in this paper). This curve tended to start dropping of packets more aggressively with a dropping probability of Pmin when the sojourn times exceeded the MINth value and then leveled off the drop probabilities at a maximum value of Pmax when the sojourn times exceeded the MAXth value. We are exploring other dropping probability curve shapes as well.

Our thinking in moving to the curve in **Figure 10c** is that, while it may be desirable to move to a 100% drop rate when we are dropping packets to preserve basic network functionality (as is a common motivation for using RED), we were reluctant to go that high simply to provide latency QoS on an otherwise well-functioning service flow, where such a high drop rate would destroy the session throughput. We were pleased to find that surprisingly low drop rates can have significant influence on session latency.

The advantages of the LRED approach are twofold:

1. Tail drops (from conventional buffer overflows) become head drops making TCP much more effective in responding to and recovering from the dropped packet.
2. Drop clusters (which can be quite common for small buffer sizes and for systems that employ simple Saturated Tail-Dropping algorithms) are eliminated. These clusters can be extremely destructive to latency-sensitive sessions.

Figure 11(a & b) shows the results of a simulation series that was identical to the situation modeled in **Figure 2**, but LRED techniques were employed to initiate early dropping of packets. The dropping probability curve of **Figure 10c** was utilized within this simulation series, with the following parameters: MINth = 200 msec, MAXth = 400 msec, Pmin = 1%, and Pmax = 5%.

Figure 11a shows that this has resulted in identical packet latency for both FTP and Gamer sessions, but the use of LRED dropping has caused these latency values to level out at about 200 msec even when the buffer depth size is increased. As a result, LRED dropping is throttling the TCP sessions and ensuring that only a finite amount of

the buffer is utilized. **Figure 11b** shows that both the overall data throughput and packet loss rates are quite reasonable as well.

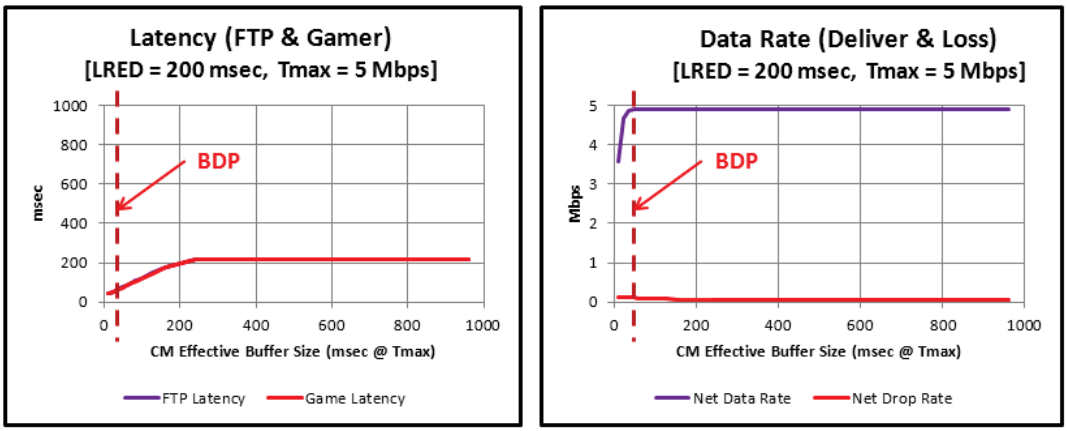


Figure 11(a & b) Latency & Throughput (LRED)

It should be apparent that an MSO is free to be more cavalier in their selection of buffer sizes when LRED is used, because as long as the buffer is large enough, the effect of LRED dropping will limit the total latency experienced by the FTP and Gaming sessions. Since MSOs don't know the exact BDP values for the actual packets, they can lean towards the use of larger buffer depths and not incur the penalty of longer delays for the latency-sensitive sessions.

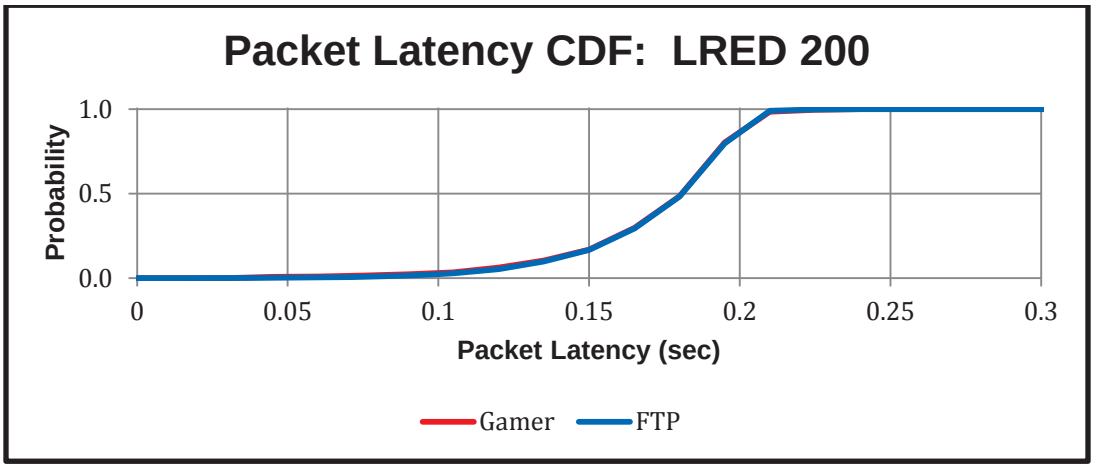


Figure 12 Latency CDF (Gamer & FTP) for LRED

Figure 12 shows the Cumulative Distribution Function (CDF) for all packet latencies for both gamer traffic (in red) and FTP sessions (in blue) flowing through the same LRED system described in **Figure 11** with a physical buffer sized to be able to store 960 msec of 5 Mbps traffic (i.e.- 600 Kbyte buffers). We can see that all packet latencies (not just the average) are distributed between 100 and 210 msec.

In this example we see a close relationship between the value of MINth (200 msec) and the resulting average latency (200+ msec) experienced by packets passing through the buffer. In fairness, we should state clearly at this point that "your mileage may vary". In our studies we have found the resulting latency to be "near" the MINth value (below when few drops are needed and above when significant drops are required). It is also important to remember that the MINth value is compared to the sojourn time that a packet spends on queue – while the packet latency also includes the upstream (in this case) portion of the uncongested network RTT.

Naturally, there are physical limits on how little packet latency can be practically achieved through LRED alone without requiring an unacceptable drop rate. We have introduced the Pmax drop rate in **Figure 10c** as a fail-safe mechanism to cause LRED to limit its drop rate when faced with an unreachable latency goal.

A number of factors affect the ease (efficiency) with which LRED can achieve it target latency. The following factors (and their combination) can reduce the effectiveness of LRED:

1) Large number of concurrent sessions
2) Low Tmax
3) Large uncongested RTT

Recall that for N concurrent sessions a single packet drop can affect only 1/Nth of the active TCP streams. A greater drop rate will be required to have the same effect on all TCP streams for a larger value of N. In addition, as the value of Tmax is reduced the interval of time between drops increases resulting in less efficient control of the TCP stream.

Fortunately, LRED seems to have a very large sweet spot and may be applicable to a major portion of our network problem domain (especially as Tmax values are trending up). Future work might map this multi-dimensional sweet spot.

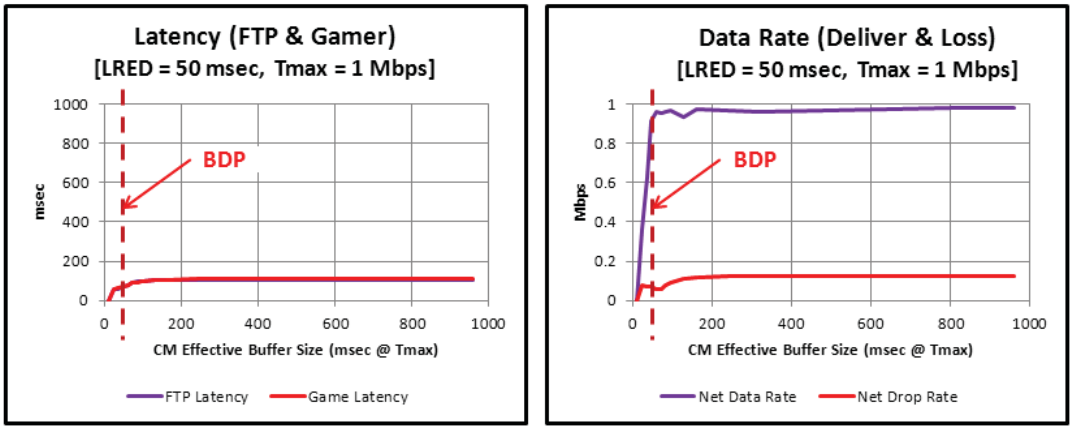


Figure 13(a & b) Latency & Throughput (LRED w/ Tmax = 1 Mbps)

Figure 13(a & b) illustrates how LRED behaves when it fails to achieve its packet latency goal of 50 msec. This example uses 10 concurrent FTP sessions plus a single gamer session over a slow 1 Mbps Tmax. It achieves average latencies near 100 msec. We can see that the packet drop rate was allowed to grow to over 10% before LRED gave up on trimming another 50 msec off of the average latency.

As a result, in its current state of development, LRED appears to be quite suitable for application as a stand-alone Type B Bufferbloat Mitigation technique if the target latency is on the order of 100 msec or greater. LRED can also serve well as a very simple, secondary Bufferbloat Mitigation mechanism that can be added on top of an SFQ system to limit the maximum latency incurred within any of the queues of the SFQ system (or to take control if/when hash collisions occur). However, the LRED solution (as it currently exists) requires high dropping probabilities to force the sessions to have target latencies of much less than 100 msec. This problem is exacerbated if many sessions are sharing the queue that is being managed. Modifications to the LRED solution are currently under study in an attempt to extend its operating point to target latencies that are much less than 100 msec.

Combining SFQ with Hashing/Serial Searching & LRED

In the CableLabs paper, the authors described a combination of SFQ and CoDel that offered the benefits of SFQ performance (when no hash collisions occurred) and relied on the CoDel scheme to improve on the reduced performance levels that occur in the event of a hash collision. In this paper the Type B (CoDel) mechanism does not distinguish between low and high latency sessions and is required to introduce large levels of latency control in order to manage latency-sensitive sessions in the event of a hash collision.

In this section we will describe a similar approach using SFQ w/ Hashing/Serial Search in combination with LRED that has very attractive properties. The advantages of this approach derive from the ability to avoid ever dropping packets from latency-sensitive sessions (due to their short sojourn times) and the ability to use very low levels of LRED drops (sufficient only to avoid overflowing a large physical buffer). Since hash collisions are extremely rare LRED need not keep the latency of non-latency-sensitive session extremely low.

The motivation for adding LRED to the SFQ w/ Hashing/Serial Search mechanism is focused on a fairly minor defect in the unmodified SFQ approach. In the approach as presented in **Figure 5** a very small drop rate results from infrequent overflows of the total physical buffer. These drops occur very infrequently but do not discriminate between FTP and gamer sessions. TCP recovery mechanisms, however, are much less effective for short packet burst (such as those typical of latency-sensitive sessions) – and these are the very latencies we are trying to shorten. The addition of LRED to this approach allows us to introduce just enough LRED drops to insure that drops never

occur due to buffer overflow – and, therefore, latency-sensitive packets are never dropped because their small latency is well below MINth.

Figure 14(a & b) shows the results of a simulation series identical to the one shown in **Figure 2** with the exception that the buffer uses both SFQ with Hashing/Serial Searching (described in the previous section) and LRED with the following parameters: MINth = 700 msec, MAXth = 1400 msec, Pmin = 1%, and Pmax = 5%. Notice that resulting FTP packet latency peaks at a value that is marginally larger than 500 msec while the gamer session latency is around 50 msec (very near the uncongested RTT minimum).

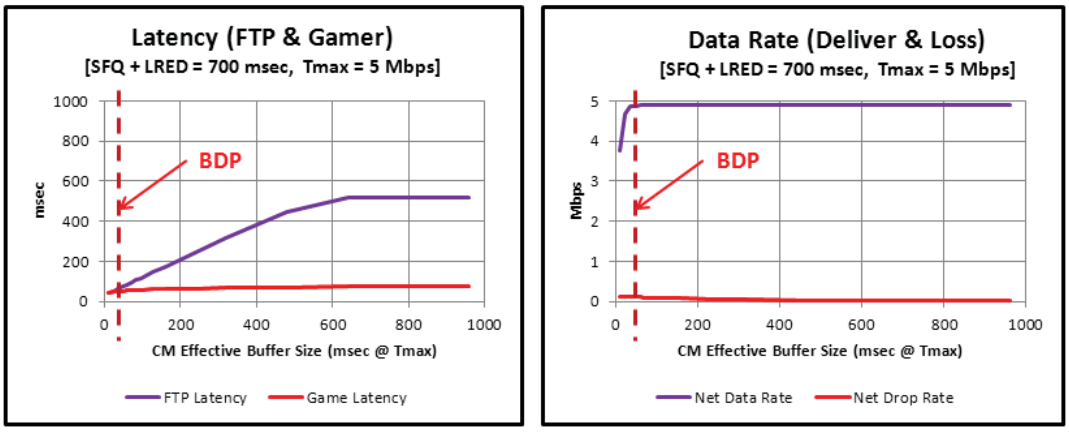


Figure 14(a & b) Latency & Throughput (SFQ with Hashing/Serial Searching & LRED)

Figure 15 and **Figure 16** show that all latency-sensitive packets (not just their average value) fall in a very narrow distribution between 50 and 100 msec. Both of these examples use a Tmax of 5 Mbps and a physical buffer size able to hold 960 msec of data at that rate. In both cases the Baseline (in green) curve shows the latency common to both traffic types in the baseline (Type A) example shown in **Figure 2**.

We can see from these two figures that the LRED value of MINth can be varied smoothly between 700 and 200 msec without affecting the gamer latency – only the FTP latency is affected.

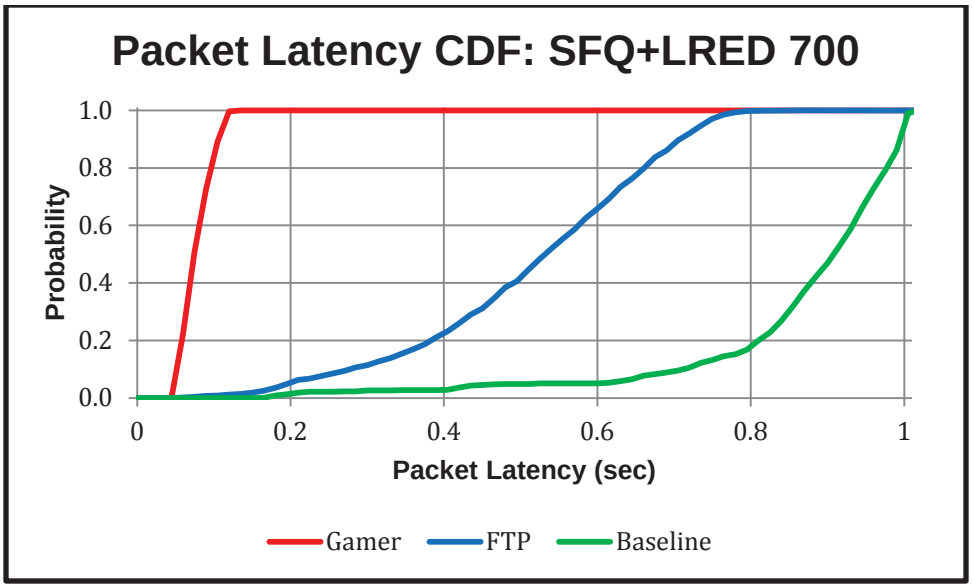


Figure 15 Latency CDF (SFQ with Hashing/Serial Searching + LRED 700)

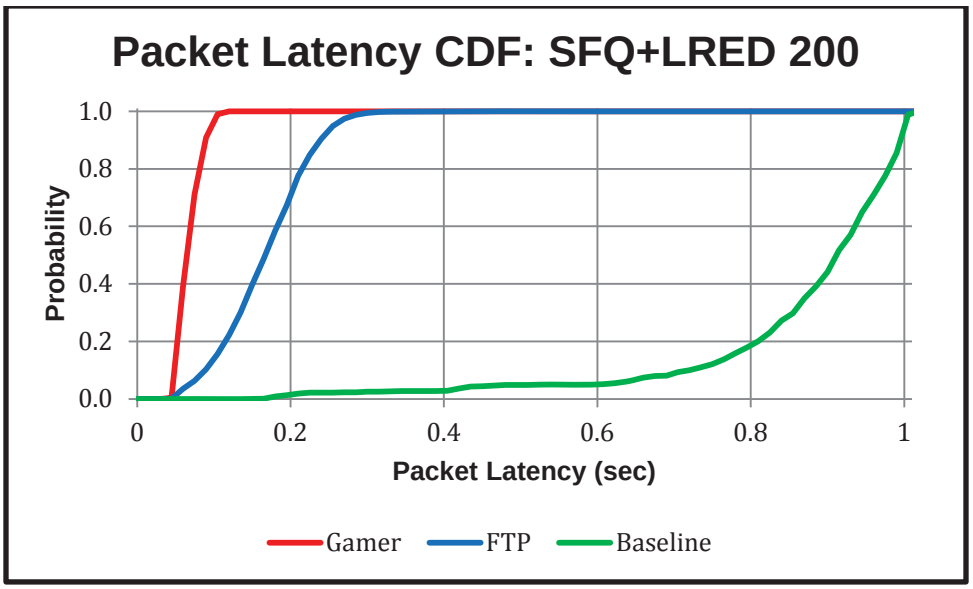


Figure 16 Latency CDF (SFQ with Hashing/Serial Searching + LRED 200)

Considerations For All Bufferbloat Mitigation Schemes

While performing the aforementioned analyses, there were several potential issues that were identified by the authors. It seemed prudent to outline some of those issues within this paper.

In general, the flow of Internet traffic through a network involves interaction between multiple devices and multiple internet protocols. A minor change in the behavior of any one device may cause an unintended change elsewhere. This change can be a technical change (whereby instabilities can occur) or can be a social change (whereby certain types of service implementation are favored over others).

The following is a list of some of the identified issues.

LEDBAT

The LEDBAT algorithm, which is widely used to throttle bitTorrent traffic, is designed to reduce the data transmission rate whenever latency is detected. A Bufferbloat avoidance algorithm will significantly reduce the latency, As a result, Bufferbloat mitigation can result in congestion not being detected by LEDBAT algorithms, and this may lead to the undesirable condition where this class of traffic may take a larger share of the available service flow capacity.

Session Bloat

We have found the effectiveness of all types of Bufferbloat mitigation to be very sensitive to the number of concurrent sessions flowing through a buffer. This property might well be expected since the only fundamental type of flow control that is available is packet dropping which relies on a response from TCP. Any single dropped packet can affect only one of a set a concurrent flows – all other flows are not aware of this drop. We might expect a service flow with 50 concurrent sessions to require ~50 times as many dropped packets to achieve a same degree of latency control.

The performance of any Bufferbloat mitigation mechanism should be evaluated against a large number of concurrent sessions since this condition cannot be avoided in the field and is intentionally increased by services such as BitTorrent and web browsing.

If a Bufferbloat mitigation algorithm eliminates or significantly reduces the latency on a session at the expense of its overall session throughput then it might encourage services to create even more parallel sessions like BitTorrent and web browsers do today. This could further aggravate the condition for all buffers in a network.

Low Bandwidth Service Flows

We have found Bufferbloat mitigation on low bandwidth flows (e.g., 1 Mbps and below) to be more difficult (especially with a large number of concurrent sessions). It might be advisable to closely study the behavior of an avoidance mechanism on SFs with low Tmax values before it is adopted and deployed.

Large bursts

A large data burst arriving at a buffer already straining to limit latency can disturb its equilibrium and sometimes trigger undesirable transient behavior. Acceptance testing for a potential Bufferbloat solution should include data burst stress testing.

Most Bufferbloat avoidance schemes detect latency due to a buildup of packets in a queue but a large burst can then arrive on top of that. Although latency is reduced to a minimum acceptable level for the application which is generating the burst, it may not be an acceptable level for other applications sharing the same service-flow. For example, the resultant bursty latency may still be excessive for a gaming application which shares the same service-flow. Therefore, some Bufferbloat improvement schemes may work well for sessions where latency does not matter but may be insufficient for applications like gaming where latency really matters.

Short-lived flows

The majority of schemes which aim to control the SF bitrate by dropping packets work well for a small number of long-lived sessions. However, real Internet traffic is composed of a lot of short random bursts for web page downloads and short periodic bursts for adaptive bit rate (ABR) video. Careful attention should be paid to any operational overhead imposed by the frequent creation and deletion of these short-lived sessions.

A mechanism, like SFQ, that attempts to identify and separate these flows must be able to create and delete these session queues with very little overhead.

Sudden rate changes

A sudden drop in bitrate (for example, from the DOCSIS peak-rate to the max-sustained-rate) may frequently occur in the normal course of CMTS traffic policing. This will cause a buildup in latency in the SF queue as packets continue to arrive at a high rate but leave the queue at a much lower rate. A Bufferbloat avoidance scheme will attempt to counteract this increase in latency but may take several seconds to reach a new equilibrium point. This could lead to instability in all TCP flows which would not have occurred in the absence of Bufferbloat mitigation. As a result, we should be alert to possible incompatibilities between standard DOCSIS QoS algorithms and Bufferbloat algorithms,

All of the above areas may require new research to be carried out in the future.

Conclusions

This paper has attempted to extend the analysis work on DOCSIS Bufferbloat management techniques that was originally started by CableLabs. In particular, the paper studied four different types of Bufferbloat management techniques. One of the techniques was a low-complexity technique known as Saturated Tail-Dropping. A second technique was a medium-complexity technique known as Latency-based RED. A third technique was a high-complexity technique known as SFQ with Hashing/Serial Searching. Finally, a combined technique using both SFQ with Hashing/Serial Searching and Latency-based RED was also studied.

In general, the authors would recommend that an MSO or vendor who can support the added complexity should first consider SFQ with Hashing/Serial Searching coupled with Latency-based RED. This solution probably provides the best performance of all solutions studied within the paper, as it separates each of the sessions within a service flow into different queues and produces near optimal performance on latency-sensitive sessions.

If a slightly simpler approach is required, then the MSO or vendor should consider SFQ with Hashing/Serial Searching. This approach still separates each of the sessions within a SF into different queues and produces excellent performance.

If an even simpler approach is required, then the MSO or vendor should consider Latency-based WRED. This solution uses packet sojourn times to determine when random packet drops should be triggered to throttle TCP bandwidths and reduce queue depths. In general, this solution does a fairly good job at limiting latency for all of the sessions sharing a service flow buffer, but it does experience more packet loss and more TCP throughput reduction than the previous solutions.

If the simplest approach is required, then the MSO or vendor should consider a Saturated Tail-Dropping solution. These solutions only drop packets to throttle TCP bandwidths whenever the shared buffer is entirely filled. The challenging part of this solution is to choose an appropriate buffer depth. The authors propose using a buffer depth that is slightly larger than the BDP values for the traffic with the longest expected RTT. The analysis in the paper showed that a session whose RTT is much longer than the anticipated RTT may excessively overflow the buffer size and experience large packet loss and much lower TCP throughputs.

The end goal of Bufferbloat Mitigation is to achieve low-latency for gaming sessions even in the presence of bursty, high-bandwidth TCP traffic that is sharing the same SF. So far, the only available schemes which achieves this goal consistently are the SFQ-based (Type A) schemes. However, the CoDel and PIE and LRED schemes can also achieve it for a high percentage of the time. Unfortunately, implementation practicalities

(ex: hardware or processing limitations in some CMs or CMTSs) may limit the application of SFQ within some devices, so continued research must continue on alternative schemes in order to achieve the end goal of a simple Bufferbloat Mitigation scheme which imposes negligible latency, with acceptable packet loss and close-to-the-target TCP throughput level.

Bibliography

[Floy] Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance," http://www.icir.org/floyd/papers/early.pdf, Retrieved electronically.

[Gett] Jim Gettys, "Bufferbloat: Dark Buffers in the Internet," http://queue.acm.org/detail.cfm?id=2071893, Retrieved electronically.

[Kuro] Jim Kurose and Keith Ross, "Computer Networking: A Top-down Approach Featuring the Internet, Sixth Edition," Addison-Wesley, 2013, ISBN 978-0-13-285620-1.

[Nich] K. Nichols and V. Jacobson, "Controlled Delay Active Queue Management draft-nichols-twvwg-code-01," http://tools.ietf.org/html/draft-nichols-tsvwg-codel-01, Retrieved electronically.

[Pan] Rong Pan et. al., "QoE: Easy as PIE," 2013 NCTA Cable Show Spring Technical Forum, NCTA.

[Whi1] Greg White, "DOCSIS Best Practices and Guidelines for Cable Modem Buffer Control, CM-GL-Buffer-V01-110915," CableLabs, 2011.

[Whi2] Greg White and Dan Rice, "Active Queue Management Algorithms for DOCSIS 3.0," CableLabs, 2013.

Abbreviations and Acronyms

ABR	Available Bit Rate
BDP	Bandwidth Delay Product
CDF	Cumulative Distribution Function
CM	Cable Modem
CMTS	Cable Modem Termination System
CoDel	Controlled Delay
DOCSIS	Data over Cable Service Interface Specification
DRAM	Dynamic Random Access Memory
ECN	Explicit Congestion Notification
FIFO	First in First Out
FTP	File Transfer Protocol
IP	Internet Protocol
LEDBAT	Low Extra Delay Background Transport
LRED	Latency-Based Random Early Detection
Mbps	Megabits per second
MSO	Multiple System Operator
PIE	Proportional Integral Enhanced
QoS	Quality of Service
RED	Random Early Detection
RTT	Round-trip Time
SF	Service Flow
SFQ	Stochastic Flow Queuing
TCP	Transmission Control Protocol
Tmax	Maximum Sustained Traffic Rate
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
WRED	Weighted Random Early Detection