Transport Area Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2020

B. Briscoe, Ed.
G. White
CableLabs

July 8, 2019

Queue Protection to Preserve Low Latency
draft-briscoe-docsis-q-protection-00

Abstract

This informational document defines and explains the specification of the queue protection algorithm used in DOCSIS 3.1. It is believed this algorithm will be useful in scenarios other than DOCSIS. A shared low latency queue relies on the non-queue-building behaviour of every traffic flow using it. However, some flows might not take such care, either accidentally or maliciously. If a queue is about to exceed a threshold level of delay, the queue protection algorithm can rapidly detect the flow(s) most likely to be responsible. It can then prevent selected packets of these flows (or whole flows) from harming the queuing delay of other traffic in the low latency queue.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2020.

Table of Contents

1. Introduction

This informational document defines and explains the specification of the queue protection (QProt) algorithm used in DOCSIS 3.1 [DOCSIS3.1]. It is believed this algorithm will be useful in scenarios other than DOCSIS.

Low queuing delay depends on hosts sending their data smoothly either at low rate or responding to explicit congestion notifications (ECN). So low latency is something hosts create themselves, not something the network gives them. Therefore, there is no incentive for flows to mis-mark their packets for the low latency queue, However, traffic from an application that does not behave in a non-queue-building way might erroneously be classified into a low latency queue, whether accidentally or maliciously. QProt prevents such erroneous behavior

from harming the queuing delay of other traffic in the low latency queue.

In normal scenarios without misclassified traffic, QProt does not intervene at all in the classification or forwarding of packets.

An overview of how low latency support has been added to DOCSIS is given in [I-D.white-tsvwg-lld]. In each direction of a DOCSIS link (upstream and downstream), there are two queues: one for Low Latency and one for Classic traffic, in an arrangement similar to the IETF's Coupled DualQ AQM [I-D.ietf-tsvwg-aqm-dualq-coupled]. The Classic queue is intended for traffic such as traditional (Reno/Cubic) TCP that needs about a round trip of buffering to fully utilize the link, and therefore has no incentive to mismark itself as low latency. The QProt function is located at the ingress to the Low Latency queue. Therefore, in the upstream QProt is located on the cable modem (CM), and in the downstream it is located on the cable CMTS (CM Termination System). If an arriving packet triggers queue protection, the DOCSIS algorithm reclassifies the packet from the Low Latency queue into the Classic queue.

If QProt is used in settings other than DOCSIS, it would be a simple matter to detect queue-building flows by using slightly different conditions, and/or trigger a different action as a consequence, as appropriate for the scenario, e.g. dropping instead of reclassifying packets or perhaps accumulating a second per-flow score to decide whether to redirect a whole flow rather than just certain packets.

The algorithm is based on a principled approach to quantifying how much each user contributes to congestion, which is used in economics to allocate responsibility for the cost of one party's behaviour on others (the economic externality). Another important feature of the approach is that the metric used for the queuing score is based on the same variable that determines the level of ECN signalling seen by the sender [RFC8311], [I-D.ietf-tsvwg-ecn-l4s-id]. This transparency is necessary to be able to objectively state (in Section 3) how a flow can keep on the 'good' side of the algorithm.

1.1. Document Roadmap

The core of the document is the walk-through of the DOCSIS QProt algorithm's pseudocode in Section 4.

Prior to that, two brief sections provide a "bluffer's guide to QProt" which should suffice for those who do not need the details or the insights. Section 2 summarizes the approach used in the algorithm. Then Section 3 considers QProt from the perspective of the end-system, by defining the behaviour that a flow needs to comply

with to avoid the QProt algorithm ejecting its packets from the low latency queue.

Section 5 gives deeper insight into the principles and rationale behind the algorithms. Then Section 6 explains the limitations of the approach, followed by the usual closing sections.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

The normative language for the DOCSIS QProt algorithm is in the DOCSIS 3.1 specifications [DOCSIS3.1], [DOCSIS3.1-CM-OSS], [DOCSIS3.1-CCAP-OSS] not in this informational guide.

The following terms and abbreviations are used:

CM: Cable Modem

CMTS: CM Termination System

Congestion-rate: The rate at which a flow induces ECN-marked (or dropped) bytes, where an ECN-mark on a packet is defined as marking all the packet's bytes. Congestion-bit-rate and congestion-volume were introduced in [RFC7713] and [RFC6789].

Non-queue-building: A flow that tends not to build a queue

Queue-building: A flow that builds a queue, and therefore is a candidate for the queue protection algorithm to detect and sanction

ECN: Explicit Congestion Notification

QProt: The Queue Protection function

2. Approach - In Brief

The algorithm is divided into mechanism and policy.

- The mechanism aspects identify flows, maintain flow-state and accumulate per-flow queuing scores;

- The policy aspects tend to be brief, but more likely to be modified in future. They can be divided into conditions and actions:
  - The conditions are the logic that determines when action should be taken to avert the risk of queuing delay becoming excessive;
  - The actions determine how this risk is averted, e.g. by redirecting packets from a flow into another queue, or to reclassify a whole flow that seems to be misclassified.

2.1. Mechanism

The algorithm maintains per-flow-state, where 'flow' usually means an end-to-end (layer-4) 5-tuple. The flow-state consists of a queuing score normalized to also represent the flow-state's own expiry time (explained in Section 5.3). A higher queuing score pushes out the expiry time further.

Non-queue-building flows tend to release their flow-state rapidly --- it usually expires reasonably early in the gap between the packets of a normal flow. Then the memory can be recycled for packets from other flows that arrive in between. So only queue-building flows hold flow state persistently.

The simplicity and effectiveness of the algorithm is due to the definition of the queuing score. It uses the internal variable from the AQM that determines the ECN marking probability, probNative, of the low latency queue. In floating point arithmetic, (0 <= probNative <= 1). The algorithm scales the size of each arriving packet of a flow by the value of probNative.

The algorithm so far would accumulate a number that would rise at the so-called congestion-rate of the flow, i.e. the rate at which the flow is contributing to congestion, or the rate at which the AQM is forwarding bytes of the flow that are ECN marked. However, rather than growing continually, the queuing score is also aged at a constant rate.

In practice, the queuing score is normalized into time units (to represent the expiry time of the flow state, as above). Then it does not need to be explicitly aged, because the natural passage of time implicitly 'ages' an expiry time.

2.2. Policy

The algorithm uses the queuing score to determine whether to eject each packet as it arrives, rather than allow it to further increase queuing delay. This limits the policies available. For instance, when queueing delay exceeds a threshold, it is not possible to eject a packet from the flow with the highest queuing scoring, because that would involve searching the queue for such a packet (if indeed there was still one in the queue). Nonetheless, it is still possible to develop a policy that protects the low latency of the queue.

Currently in DOCSIS, when the policy conditions are met, the action taken to protect the low latency queue is to reclassify a packet into the Classic queue.

3. Necessary Flow Behaviour

The QProt algorithm described here can be used for responsive and/or unresponsive flows.

- It is possible to objectively describe the least responsive way that a flow will need to respond to congestion signals in order to avoid triggering queue protection, no matter the link capacity and no matter how much other traffic there is.

- It is not possible to describe how fast or smooth an unresponsive flow should be to avoid queue protection, because this depends on how much other traffic there is and the capacity of the link, which an application is unable to know. However, the smoother an unresponsive flow paces its packets and the lower its rate relative to typical broadband link capacities, the less likelihood that it will risk causing enough queueing to trigger queue protection.

In DOCSIS, unresponsive flows are classified into the low latency queue by a Non-Queue-Building (NQB) Diffserv codepoint [I-D.white-tsvwg-nqb], or an operator can use various other additional local classifiers.

Responsive low latency flows have to use L4S ECN [I-D.ietf-tsvwg-ecn-l4s-id] to get classified into the low latency queue.

The QProt algorithm is driven from the same variable that drives the ECN marking probability in the low latency queue (Annex N of [DOCSIS3.1]). The algorithm that calculates this internal variable is run on the arrival of every packet, whether it is ECN-capable or

not, so that it can be used by the QProt algorithm. But the variable is only used to ECN-mark packets that are ECN-capable.

Not only does this dual use of the variable improve processing efficiency, but it also makes the basis of the QProt algorithm visible and transparent, at least for responsive ECN-capable flows. Then it is possible to state objectively that a flow can avoid triggering queue protection by keeping the bit rate of ECN marked packets (the congestion-rate) below AGING, which is a configured constant of the algorithm (default 2^19 B/s ~= 4.2 Mb/s). Note that a congestion controller is advised to keep the average congestion-rate well below this level (e.g. ~1 Mb/s), to ensure that queue protection is not triggered during transient dynamics.

If the QProt algorithm is used in other settings, it would need to be based on the visible level of congestion signalling, in a similar way to DOCSIS. Without transparency of the basis of the algorithm's decisions, end-systems would not be able to avoid triggering queue protection on an objective basis.

4. Pseudocode Walk-Through

The algorithm categorizes packets into flows, usually defined by a common 5-tuple (or 4-tuple) of:

- source and destination IP addresses of the innermost IP header found;
- protocol of the layer above this IP header
- either of:
  - source and destination port numbers, for TCP, UDP, UDP-Lite, SCTP, DCCP, etc.
  - Security Parameters Index (SPI) for IPSec Encapsulating Security Payload (ESP) [RFC4303].

Annex P.3 of DOCSIS 3.1 [DOCSIS3.1] defines various strategies to find these headers by skipping extension headers or encapsulations. If they cannot be found the spec defines various less-specific 3-tuples that would be used. DOCSIS 3.1 should be referred to for all these strategies, which will not be repeated here.

4.1. Input Parameters, Constants and Variables

The operator input parameters that set the parameters in the first two blocks of pseudocode below are defined for cable modems (CMs) in [DOCSIS3.1-CM-OSS] and for CMTSs in [DOCSIS3.1-CCAP-OSS]. The constants below that are derived from them or hard-coded.

```
// Input Parameters
QPROTECT_ON         // Queue Protection is enabled if TRUE
CRITICALqL_us       // Threshold delay of L queue [us]
CRITICALqLSCORE_us  // The threshold queuing score [us]
LG_AGING            // The aging rate of the q'ing score, as
                    //  log base 2 of the congestion-rate [lg(B/s)]

// Input Parameters for the calcProbNative() algorithm:
MAXTH_us            // Max marking threshold [us] for IAQM
LG_RANGE            // Log base 2 of the range of ramp [lg(ns)]
                    //  Default: 2^19 = 524288 ns (roughly 525 us)


// Constants, either derived from input parameters or hard-coded
AGING = pow(2, (LG_AGING-30) );       // Convert lg([B/s]) to [B/ns]
CRITICALqL = CRITICALqL_us * 1000             // Convert [us] to [ns]
CRITICALqLSCORE = CRITICALqLSCORE_us * 1000   // Convert [us] to [ns]
// Threshold for the q'ing score condition
CRITICALqLPRODUCT = CRITICALqL * CRITICALqLSCORE

ATTEMPTS = 2; // Max attempts to pick a bucket (vendor-specific)
BI_SIZE = 5;  // Bit-width of index number for non-default buckets
NBUCKETS = pow(2, BI_SIZE);  // No. of non-default buckets
MASK = NBUCKETS-1;           // convenient constant, filled with ones

                       // Queue Protection exit states
EXIT_SUCCESS  = 0;     // Forward the packet
EXIT_SANCTION = 1;     // Redirect the packet

MAX_PROB      = 1; // For integer arithmetic, would use a large int
                   //  e.g., 2^31, to allow space for overflow
MAXTH = MAXTH_us * 1000;   // Max marking threshold [ns]
// Minimum marking threshold of 2 MTU for slow links [ns]
FLOOR =  2 * 8 * MAX FRAME SIZE * 10^9 / MAX RATE;
RANGE = (1 << LG_RANGE);      // Range of ramp [ns]
MINTH = max ( MAXTH - RANGE, FLOOR);
MAXTH = MINTH + RANGE;           // Max marking threshold [ns]
```

The following definitions explain the purpose of important variables and functions.

```
// Public variables:
qdelay         // The current queuing delay of the LL queue [ns]
probNative     // The native probability of the LL queue within [0,1]

// External variables
packet             // The structure holding packet header fields
packet.size        // The size of the current packet [B]
packet.uflow       // The flow identifier of the current packet
                   //  (e.g. 5-tuple or 4-tuple if IPSec)

// Irrelevant details of DOCSIS function to return qdelay are removed
qdelayL(...)      // Returns current delay of the low latency Q [ns]
```

The array of bucket structures defined below is used by all the Queue Protection functions:

```
struct bucket { // The leaky bucket structure to hold per-flow state
   id;          // identifier (e.g. 5-tuple) of the flow using bucket
   t_exp;       // expiry time;
                // (t_exp - now) = flow's normalized q'ing score [ns]
};
struct bucket buckets[NBUCKETS+1];
```

4.2. Queue Protection Data Path

All the functions of Queue Protection operate on the data path, driven by packet arrivals.

The following functions that maintain per-flow queuing scores and manage per-flow state are considered primarily as mechanism:

```
pick_bucket(uflow_id); // Returns bucket identifier

fill_bucket(bucket_id, pkt_size, probNative); // Returns queuing
score

calcProbNative(qdelay) // Returns probability of ECN-marking
```

The following function is primarily concerned with policy:

```
qprotect(packet, ...); // Returns exit status to either forward or
redirect the packet
```

It is more likely that there might be future modifications to policy aspects. Therefore, policy aspects would be less appropriate candidates for any hardware acceleration.

The entry point to these functions is qprotect(), which would be called from packet classification as follows:

```
classifier(packet) {
   // ...
   // Classify packet
      // if packet classified to Low Latency Service Flow
      if (QPROTECT_ON) {
         if (qprotect(packet, qL.byte_length) == EXIT_SANCTION) {
            // redirect packet to    Classic Service Flow
         }
      }
      // Forward packet to Low Latency Service Flow
   // Continue...
}
```

On each packet arrival, qprotect() measures the current queue delay and derives the native probability from it. Then it uses pick_bucket to find the bucket already holding the flow's state, or to allocate a new bucket if the flow is new or its state has expired (the most likely case). Then the queuing score is updated by the fill_bucket() function. That completes the mechanism aspects.

The comments against the subsequent policy conditions and actions should be self-explanatory at a superficial level. The deeper rationale for these conditions is given in Section 5.4.

```
   // Per packet queue protection
   qprotect(packet, ...) {

      bckt_id;   // bucket index
      qLscore;   // queuing score of pkt's flow [ns]

      qdelay = qL.qdelay(...);
      probNative = calcProbNative(qdelay);

      bckt_id = pick_bucket(packet.uflow);
      // Not shown: if (bckt_id->t_exp risks overflow) EXIT_SANCTION
      qLscore = fill_bucket(buckets[bckt_id], packet.size, probNative);

      // Determine whether to sanction packet
      if (  ( qdelay > CRITICALqL )  // Test if qdelay over threshold...
         // ...and if flow's q'ing score scaled by qdelay/CRITICALqL
         // ...exceeds CRITICALqLSCORE
         && ( qdelay * qLscore > CRITICALqLPRODUCT ) )

         return EXIT_SANCTION;

      else
         return EXIT_SUCCESS;
   }
```

The pick_bucket() function is optimized for flow-state that will normally have expired from packet to packet of the same flow. it is just one way of finding the bucket associated with the flow ID of each packet - it might be possible to develop more efficient alternatives.

The algorithm is arranged so that the bucket holding any live (non-expired) flow-state associated with a packet will always be found before a new bucket is allocated. The constant ATTEMPTS, defined earlier, determines how many hashes are used to find a bucket for each flow (actually, only one hash is generated; then, by default, 5 bits of it at a time are used as the hash value, because by default there are 2^5 = 32 buckets).

The algorithm stores the flow's own ID in its flow-state. So, when the next packet of a flow arrives, if it finds its own ID, but the flow-state has expired, the algorithm just adds the packet's queuing score to 'now', as a new flow would, If it does not find the flow's ID, and the expiry time is still current, the algorithm can tell that another flow is using that bucket, and it continues to look for a bucket for the flow. Even if it finds a bucket where the expiry time has passed, it doesn't immediately use it. It merely remembers it as the potential bucket to use. But first it runs through all the

ATTEMPTS hashes to check for another bucket assigned to the flow, in case it is still live.

If a live bucket is not already associated with the packet's flow, the algorithm should then have already set aside an existing bucket with a score that has aged out. Given this bucket is no longer necessary to hold state for its previous flow, it can be recycled for use by the present packet's flow.

If all else fails, there is one additional bucket (called the dregs) that can be used. If the dregs is still in live use by another flow, subsequent flows that cannot find a bucket of their own all share it, adding their score to the one in the dregs. A flow might get away with using the dregs on its own, but when there are many mis-marked flows, multiple flows are more likely to collide in the dregs, including innocent flows. The choice of number of buckets and number of hash attempts determines how likely it will be that this undesirable scenario will occur.

```
   // Pick the bucket associated with flow uflw
   pick_bucket(uflw) {

      now;                    // current time [ns]
      j;                      // loop counter
      h32;                    // holds hash of the packet's flow IDs
      h;                      // bucket index being checked
      hsav;                   // interim chosen bucket index

      h32   = hash32(uflw);   // 32-bit hash of flow ID
      hsav  = NBUCKETS;       // Default bucket
      now   = get_time_now();

      // The for loop checks ATTEMPTS buckets for ownership by flow-ID
      // It also records the 1st bucket, if any, that could be recycled
      // because it's expired.
      // Must not recycle a bucket until all ownership checks completed
      for (j=0; j<ATTEMPTS; j++) {
         // Use least signif. BI_SIZE bits of hash for each attempt
         h = h32 & MASK;
         if (buckets[h].id == uflw) {    // Once uflw's bucket found...
            if (buckets[h].t_exp <= now) // ...if bucket has expired...
               buckets[h].t_exp = now;   // ...reset it
            return h;                    // ...use it
         }
         else if ( (hsav == NBUCKETS)  // If not seen expired bucket yet
                                       //  and this bucket has expired
             && (buckets[h].t_exp <= now) ) {
            hsav = h;                  // set it as the interim bucket
         }
         h32 >>= BI_SIZE;          // Bit-shift hash for next attempt
      }
      // If reached here, no tested bucket was owned by the flow-ID
      if (hsav != NBUCKETS) {
         // If here, found an expired bucket within the above for loop
         buckets[hsav].t_exp = now;              // Reset expired bucket
      } else {
         // If here, we're having to use the default bucket (the dregs)
         if (buckets[hsav].t_exp <= now) {   // If dregs has expired...
            buckets[hsav].t_exp = now;       // ...reset it
         }
      }
      buckets[hsav].id = uflw; // In either case, claim for recycling
      return hsav;
   }
```

The fill_bucket() function both accumulates and ages the queuing score over time, as outlined in Section 2.1. To make aging the score

efficient, the increment of the queuing score is normalized to units of time by dividing by AGING, so that the result represents the new expiry time of the flow.

It might be thought that, instead of multiplying the packet size (pkt_sz) by probNative, packets could be selected by the AQM with probability probNative, as they are for ECN-marking. Then the full size of those selected packets would be used to increment the queuing score. However, experience with other congestion policers has found that the score then increments far too jumpily, particularly when probNative is low.

A deeper explanation of the queuing score is given in Section 5.

```
fill_bucket(bckt_id, pkt_sz, probNative) {
   // Add packet's queuing score
   // For integer arithmetic, a bit-shift can replace the division
   buckets[bckt_id].t_exp += probNative * pkt_sz / AGING;
   return  (buckets[bckt_id].t_exp - now);
}
```

To derive this queuing score, the QProt algorithm uses the linear ramp function calcProbNative() to normalize instantaneous queuing delay into a probability in the range [0,1], which it assigns to probNative.

```
calcProbNative(qdelay){
      if ( qdelay >= MAXTH ) {
         probNative = MAX_PROB;
      } else if ( qdelay > MINTH ) {
         probNative = MAX_PROB * (qdelay - MINTH)/RANGE;
         // In practice, the * and the / would use a bit-shift
      } else {
         probNative = 0;
      }
      return probNative;
}
```

5. Rationale

5.1. Rationale: Blame for Queuing, not for Rate in Itself

Figure 1 poses the question of which flow is more to blame for queuing delay; the unresponsive constant bit rate flow (c) that is consuming about 80% of the capacity, or the flow sending regular short unresponsive bursts (b)? The smoothness of c seems better for avoiding queuing, but its high rate does not. However, if flow c was

not there, or ran slightly more slowly, b would not cause any queuing.

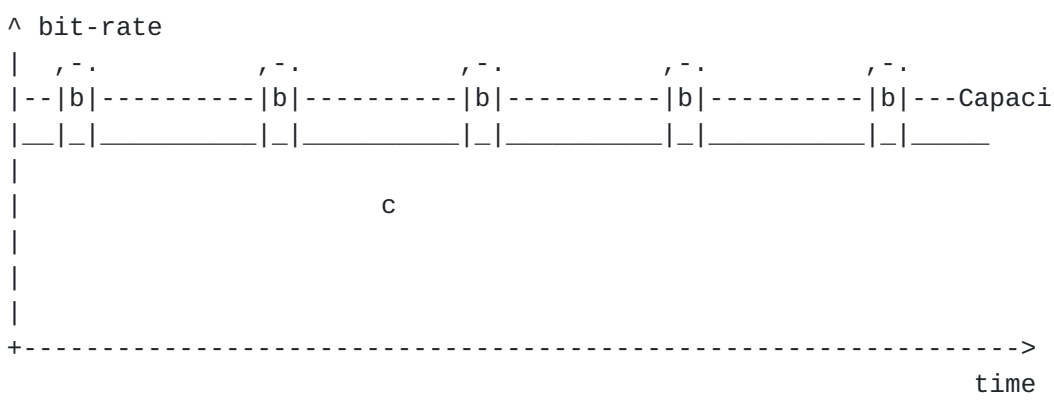

Figure 1: Which is More to Blame for Queuing Delay?

To explain queuing scores, in the following it will initially be assumed that the QProt algorithm is accumulating queuing scores, but not taking any action as a result.

To quantify the responsibility that each flow bears for queuing delay, the QProt algorithm accumulates the product of the level of congestion and the rate of each flow, both measured at the instant each packet arrives. The level of congestion is normalized to a dimensionless number between 0 and 1 (nativeProb). The instantaneous flow rate is represented at each discrete event when a packet arrives by the packet's size, which accumulates faster the more packets arrive within each unit of time. The unit of the resulting queue score is "congested-bytes" per second, which distinguishes it from just bytes per second.

Then, during the periods between bursts (b), neither flow accumulates any queuing score - the high rate of c is benign. But, during each burst, if we say the rate of c and b are 80% and 45% of capacity, thus causing 125% overload, they each bear (80/125)% and (45/125)% of the responsibility for the queuing delay (64% and 36%). The algorithm does not explicit calculate these percentages. They are just the outcome of the number of packets arriving from each flow during the burst.

To summarize, the queuing score never sanctions rate solely on its own account. It only sanctions rate inasmuch as it causes queuing.

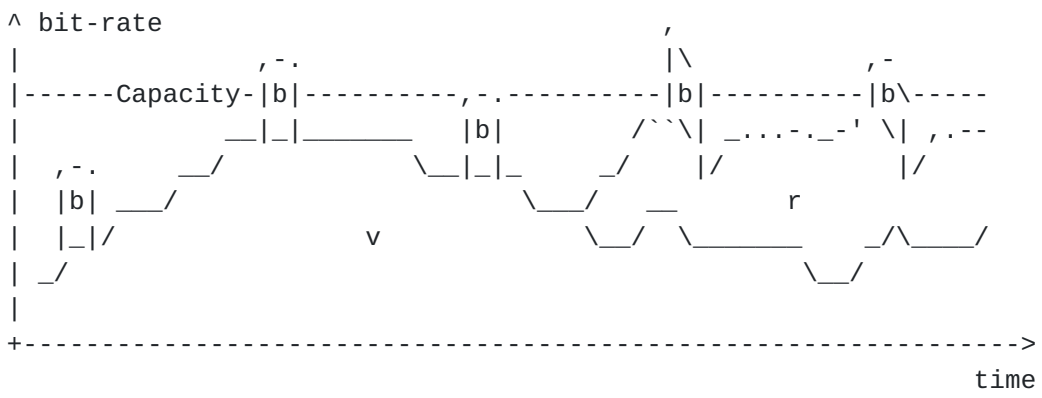

Figure 2: Responsibility for Queuing: More Complex Scenario

Figure 1 gives a more complex illustration of the way the queuing score assigns responsibility for queuing (limited to the precision that ASCII art can illustrate). The unresponsive bursts (b) are the same as in the previous example, but a variable rate video (v) replaces flow c. It's rate varies as the complexity of the video scene varies. Also on a slower timescale, in response to the level of congestion, the video adapts its quality. However, on a short time-scale it appears to be unresponsive to small amounts of queuing. Also, part-way through, a low latency responsive flow (r) joins in, aiming to fill the balance of capacity left by the other two.

The combination of the first burst and the low application-limited rate of the video causes neither flow to accumulate queuing score. In contrast, the second burst causes similar excessive overload (125%) to the example in Figure 1. Then, the video happens to reduce its rate (probably due to a less complex scene) so the third burst causes only a little congestion. Let us assume the resulting queue causes probNative to rise to just 1%, then the queuing score will only accumulate 1% of the size of each packet of flows v and b during this burst.

The fourth burst happens to arrive just as the new responsive flow (r) has filled the available capacity, so it leads to very rapid growth of the queue. After a round trip the responsive flow rapidly backs off, and the adaptive video also backs off more rapidly than it would normally, because of the very high congestion level. The rapid response to congestion of flow r reduces the queuing score that all three flows accumulate, but they each still bear the cost in proportion to the product of the rates at which their packets arrive at the queue and the value of probNative when they do so. Thus, during the fifth burst, they all accumulate less score than the fourth, because the queuing delay is not as excessive.

5.2. Rationale for Aging the Queuing Score

Even well-behaved flows will not always be able to respond fast enough to dynamic events. Also well-behaved flow(s), e.g. DCTCP [RFC8257], TCP Prague [PragueLinux] or the L4S variant of SCReAM for real-time media [RFC8298], can maintain a very shallow queue by continual careful probing for more while also continually subtracting a little from their rate (or congestion window) in response to low levels of ECN signalling. Therefore, the QProt algorithm needs to continually offer a degree of forgiveness to age out the queuing score as it accumulates.

Scalable congestion controllers such as those above maintain their congestion window in inverse proportion to the congestion level, probNative, That leads to the important property that on average a scalable flow holds the product of its congestion window and the congestion level constant, no matter the capacity of the link or how many other flows it competes with. For instance, if the link capacity doubles, a scalable flow induces half the congestion probability. Or if three scalable flows compete for the capacity, each flow will reduce to one third of the capacity and increase the congestion level by 3x.

This suggests that the QProt algorithm will not sanction a well-behaved scalable flow if it ages out the queuing score at a sufficient constant rate. The constant will need to be somewhat about the average of a well-behaved scalable flow to allow for normal dynamics.

Relating QProt's aging constant to a scalable flow does not mean that a flow has to behave like a scalable flow. It can be less aggressive, but not more. For instance, a longer RTT flow can run at a lower congestion-rate than the aging rate, but it can also increase its aggressiveness to equal the rate of short RTT scalable flows [ScalingCC]. The constant aging of QProt also means that a long-running unresponsive flow will be prone to trigger QProt if it runs faster than a competing responsive scalable flow would. And, of course, if a flow causes excessive queuing in the short-term, its queuing score will still rise faster than the constant aging process will decrease it. Then QProt will still eject the flow's packets before they harm the low latency of the shared queue.

5.3. Rationale for Normalized Queuing Score

The QProt algorithm holds a flow's queuing score state in a structure called a bucket, because of its similarity to a classic leaky bucket (except the contents of the bucket does not represent bytes).

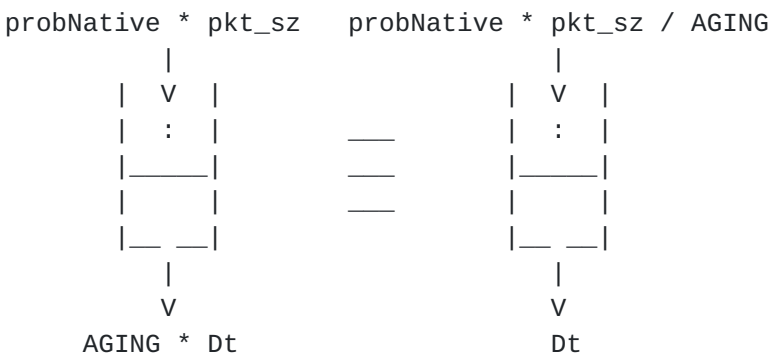

Figure 3: Normalization of Queuing Score

The accumulation and aging of the queuing score is shown on the left of Figure 3 in token bucket form. Dt is the difference between the times when the scores of the current and previous packets were processed.

A normalized equivalent of this token bucket is shown on the right of Figure 3, dividing both the input and output by the constant AGING rate. The result is a bucket-depth that represents time and it drains at the rate that time passes.

As a further optimization, the time the bucket was last updated is not stored with the flow-state. Instead, when the bucket is initialized the queuing score is added to the system time 'now' and the resulting expiry time is written into the bucket. Subsequently, if the bucket has not expired, the incremental queuing score is added to the time already held in the bucket. Then the queuing score always represents the expiry time of the flow-state itself. This means that the queuing score does not need to be aged explicitly because it ages itself implicitly.

5.4. Rationale for Policy Conditions

Pseudocode for the QProt policy conditions is given in Section 4.1 within the second half of the qprotect() function. When each packet arrives, after finding its flow state and updating the queuing score of the packet's flow, the algorithm checks whether the shared queue delay exceeds a constant threshold CRITICALqL (e.g. 2 ms), as repeated below for convenience:

```
if (  ( qdelay > CRITICALqL )  // Test if qdelay over threshold...
   // ...and if flow's q'ing score scaled by qdelay/CRITICALqL
   // ...exceeds CRITICALqLSCORE
   && ( qdelay * qLscore > CRITICALqLPRODUCT ) )
   // Recall that CRITICALqLPRODUCT = CRITICALqL * CRITICALqLSCORE
```

If the queue delay threshold is exceeded, the flow's queuing score is temporarily scaled up by the current queue delay normalized as a ratio of the threshold queuing delay CRITICALqL. If this scaled up score exceeds another constant threshold CRITICALqLSCORE, the packet is ejected. The actual last line of code above multiplies both sides of the second condition by CRITICALqLSCORE to avoid a costly division.

This approach allows each packet to be assessed once, as it arrives. Once queue delay exceeds the threshold, it has two implications:

- The current packet might be ejected even though there are packets already in the queue from flows with higher queuing scores. However, any flow that continues to contribute to the queue will have to send further packets, giving an opportunity to eject them as well, as they subsequently arrive.

- The next packets to arrive might not be ejected, because they might belong to flows with low queuing scores. In this case, queue delay could continue to rise with no opportunity to eject a packet. This is why the queuing score is scaled up by the current queue delay. Then, the more the queue has grown without ejecting a packet, the more the algorithm 'raises the bar' to further packets.

The above approach is preferred over searching for the flow with the highest queuing score and searching for one of its packets to eject from the queue (if one is still there).

Figure 4 explains this approach graphically. On the horizontal axis it shows actual harm, meaning the queuing delay in the shared queue. On the vertical axis it shows the behaviour record of the flow associated with the currently arriving packet, represented in the algorithm by the flow's queuing score. The shaded region represents the combination of actual harm and behaviour record that will lead to the packet being ejected.

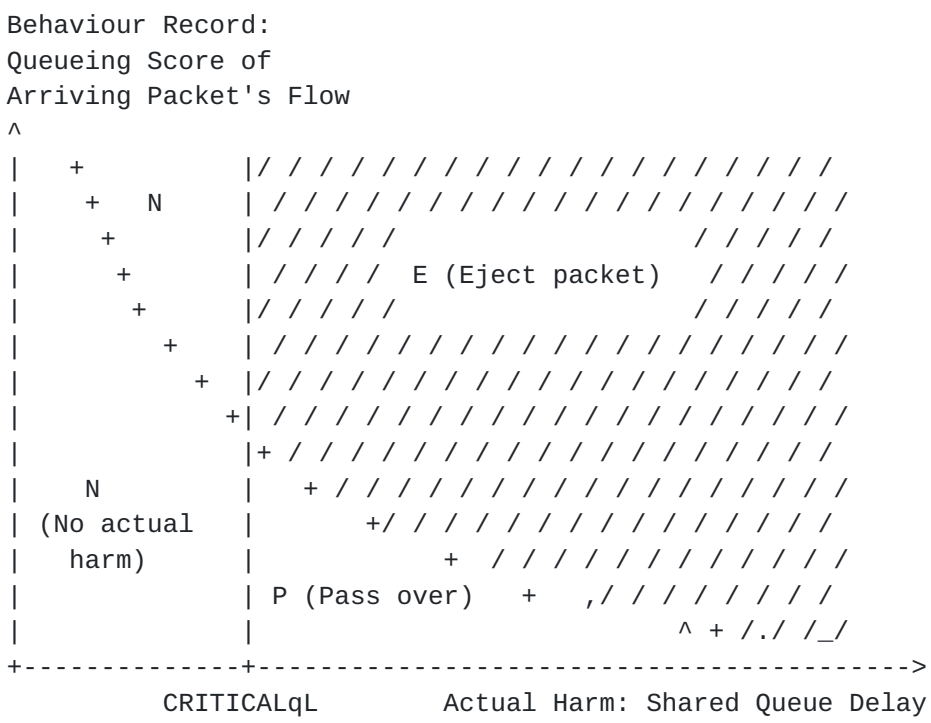

Figure 4: Graphical Explanation of the Policy Conditions

The region labelled 'N' represents cases where the first condition is not met - No actual harm - queue delay is below the critical threshold, CRITICALqL.

The region labelled 'E' represents cases where there is actual harm (queue delay exceeds CRITICALqL) and the queuing score associated with the arriving packet is high enough to be able to eject it with certainty.

The region labelled 'P' represents cases where there is actual harm, but the queuing score of the arriving packet is insufficient to eject it, so it has to be Passed over. This adds to queuing delay, but the alternative would be to sanction an innocent flow. It can be seen that, as actual harm increases, the judgement of innocence becomes increasingly stringent; the behaviour record of the next packet's flow does not have to be as bad to eject it.

6. Limitations

The QProt algorithm groups packets with common layer-4 flow identifiers. It then uses this grouping to accumulate queuing scores and to sanction packets.

Some applications might initiate multiple flows between the same end-points, e.g. for media, control, data, etc. Others might use common

flow identifiers for all these streams. Also, a user might group multiple application flows within the same encrypted VPN between the same layer-4 tunnel end-points.

The use of a queuing score that excludes those aspects of flow rate that do not contribute to queuing Section 5.1 goes some way to mitigating this limitation. However, ultimately this choice of flow identifiers is pragmatic and not particularly principled.

7. IANA Considerations

This specification contains no IANA considerations.

8. Security Considerations

The whole of this document considers the security concern of how to identify traffic that does not comply with the non-queue-building behaviour required to use a shared low latency queue, whether accidentally or maliciously.

The algorithm has been designed to be fail gracefully in the face of traffic crafted to overrun the resources of the algorithm. This means that non-queue-building flows will always be less likely to be sanctioned than queue-building flows. But an attack could be contrived to deplete resources in such as way that the proportion of innocent (non-queue-building) flows that are incorrectly sanctioned could increase.

9. Comments Solicited

Comments and questions are encouraged and very welcome. They can be addressed to the IETF Transport Area mailing list <tsv-area@ietf.org>, and/or to the authors.

10. Acknowledgements

Thanks to Tom Henderson for his review of this document. The design of the QProt algorithm and the settings of the parameters benefited from discussion and critique from the participants of the cable industry working group on Low Latency DOCSIS.

11. References

11.1. Normative References

[DOCSIS3.1]
CableLabs, "MAC and Upper Layer Protocols Interface (MULPI) Specification, CM-SP-MULPIv3.1", Data-Over-Cable Service Interface Specifications DOCSIS(R) 3.1 Version i17 or later, January 2019, <https://specification-search.cablelabs.com/CM-SP-MULPIv3.1>.

[DOCSIS3.1-CCAP-OSS]
CableLabs, "CCAP Operations Support System Interface Spec", Data-Over-Cable Service Interface Specifications DOCSIS(R) 3.1 Version i14 or later, January 2019, <https://specification-search.cablelabs.com/CM-SP-CM-OSSIv3.1>.

[DOCSIS3.1-CM-OSS]
CableLabs, "Cable Modem Operations Support System Interface Spec", Data-Over-Cable Service Interface Specifications DOCSIS(R) 3.1 Version i14 or later, January 2019, <https://specification-search.cablelabs.com/CM-SP-CM-OSSIv3.1>.

[I-D.ietf-tsvwg-ecn-l4s-id]
Schepper, K. and B. Briscoe, "Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay (L4S)", draft-ietf-tsvwg-ecn-l4s-id-06 (work in progress), March 2019.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

11.2. Informative References

[I-D.ietf-tsvwg-aqm-dualq-coupled]
Schepper, K., Briscoe, B., and G. White, "DualQ Coupled AQMs for Low Latency, Low Loss and Scalable Throughput (L4S)", draft-ietf-tsvwg-aqm-dualq-coupled-09 (work in progress), July 2019.

[I-D.white-tsvwg-lld]
White, G., Sundaresan, K., and B. Briscoe, "Low Latency DOCSIS - Technology Overview", draft-white-tsvwg-lld-00 (work in progress), March 2019.

[I-D.white-tsvwg-nqb]
White, G. and T. Fossati, "Identifying and Handling Non Queue Building Flows in a Bottleneck Link", draft-white-tsvwg-nqb-02 (work in progress), June 2019.

[PragueLinux]
Briscoe, B., De Schepper, K., Albisser, O., Misund, J., Tilmans, O., Kuehlewind, M., and A. Ahmed, "Implementing the `TCP Prague' Requirements for Low Latency Low Loss Scalable Throughput (L4S)", Proc. Linux Netdev 0x13 , March 2019, <https://www.netdevconf.org/0x13/session.html?talk-tcp-prague-l4s>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

[RFC6789] Briscoe, B., Ed., Woundy, R., Ed., and A. Cooper, Ed., "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, DOI 10.17487/RFC6789, December 2012, <https://www.rfc-editor.org/info/rfc6789>.

[RFC7713] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements", RFC 7713, DOI 10.17487/RFC7713, December 2015, <https://www.rfc-editor.org/info/rfc7713>.

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

[RFC8298] Johansson, I. and Z. Sarker, "Self-Clocked Rate Adaptation for Multimedia", RFC 8298, DOI 10.17487/RFC8298, December 2017, <https://www.rfc-editor.org/info/rfc8298>.

[ScalingCC]
Briscoe, B. and K. De Schepper, "Resolving Tensions between Congestion Control Scaling Requirements", Simula Technical Report TR-CS-2016-001 arXiv:1904.07605, July 2017, <https://arxiv.org/abs/1904.07605>.

Authors' Addresses

Bob Briscoe (editor)
CableLabs
UK

Email: ietf@bobbriscoe.net
URI: http://bobbriscoe.net/

Greg White
CableLabs
US

Email: G.White@CableLabs.com