Stream:	Independent Submission	
RFC:	9957	
Category:	Informational	
Published:	April 2026	
ISSN:	2070-1721	
Authors:	B. Briscoe, Ed.	G. White
	Independent	*CableLabs*

RFC 9957
The DOCSIS® Queue Protection Algorithm to Preserve Low Latency

Abstract

This Informational RFC explains the specification of the queue protection algorithm used in Data-Over-Cable Service Interface Specification (DOCSIS) technology since version 3.1. 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 flows most likely to be responsible. It can then prevent harm to other traffic in the low-latency queue by ejecting selected packets (or all packets) of these flows. This document is designed for four audiences: a) congestion control designers who need to understand how to keep on the "good" side of the algorithm; b) implementers of the algorithm who want to understand it in more depth; c) designers of algorithms with similar goals, perhaps for non-DOCSIS scenarios; and d) researchers interested in evaluating the algorithm.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9957.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction

This Informational RFC explains the specification of the queue protection (QProt) algorithm used in DOCSIS technology since version 3.1 [DOCSIS].

Although the algorithm is defined in Annex P of [DOCSIS], it relies on cross references to other parts of the set of specifications. This document pulls all the strands together into one self-contained document. The core of the document is a similar pseudocode walk-through to that in the DOCSIS specification, but it also includes additional material:

i. a brief overview,
ii. a definition of how a data sender needs to behave to avoid triggering Queue Protection, and
iii. a section giving the rationale for the design choices.

Low queuing delay depends on hosts sending their data smoothly, either at a low rate or responding to Explicit Congestion Notification (ECN) (see [RFC8311] and [RFC9331]). Therefore, low-latency queuing is something hosts create themselves, not something the network gives them. This tends to ensure that self-interest alone does not drive flows to mismark 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 protects other traffic in the low-latency queue from the harm due to excess queuing that would otherwise be caused by such anomalous behaviour.

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

An overview of how low-latency support has been added to DOCSIS technology is given in [LLD]. In each direction of a DOCSIS link (upstream and downstream), there are two queues: one for Low-Latency (LL) and one for Classic traffic, in an arrangement similar to the IETF's Dual-Queue Coupled Active Queue Management (AQM) [RFC9332]. The two queues enable a transition from "Classic" to "Scalable" congestion control so that low latency can become the norm for any application, including ones seeking both full throughput and low latency, not just low-rate applications that have been more traditionally associated with a low-latency service. The Classic queue is only necessary for traffic such as traditional (Reno [RFC5681] / Cubic [RFC9438]) TCP that needs about a round trip of buffering to fully utilize the link; therefore, this traffic 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); in the downstream, it is located on the CM Termination System (CMTS). If an arriving packet triggers queue protection, the QProt algorithm ejects the packet from the Low-Latency queue and reclassifies it into the Classic queue.

If QProt is used in settings other than DOCSIS links, it would be a simple matter to detect queue-building flows by using slightly different conditions and/or to 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. Such work is for future study and out of scope of the present document.

The QProt algorithm is based on a rigorous approach to quantifying how much each flow 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 (see [RFC8311] and [RFC9331]). This makes the internal queuing score visible externally as ECN markings. 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, 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 algorithm. Then, Section 6 explains the limitations of the approach. The usual closing sections follow.

1.2. Terminology

The normative language for the DOCSIS QProt algorithm is in the DOCSIS specifications [DOCSIS], [DOCSIS-CM-OSS], and [DOCSIS-CCAP-OSS]: not in this Informational RFC. If there is any inconsistency, the DOCSIS specifications take precedence.

The following terms and abbreviations are used:

CM: Cable Modem

CMTS: CM Termination System

Congestion-rate: The transmission rate counting only bits or bytes contained within packets of a flow that have the Congestion Experienced (CE) codepoint set in the IP-ECN field [RFC3168] (including IP headers unless specified otherwise). Congestion-bit-rate and congestion-volume were introduced in [RFC7713] and [RFC6789].

DOCSIS: Data-Over-Cable System Interface Specification. "DOCSIS" is a registered trademark of Cable Television Laboratories, Inc. ("CableLabs").

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

Note the use of lowercase "non-queue-building", which describes the behaviour of a flow, in contrast to uppercase Non-Queue-Building (NQB), which refers to a Diffserv marking [RFC9956]. A flow identifying itself as uppercase Non-Queue-Building may not behave in a non-queue-building way, which is what the QProt algorithm is intended to detect.

queue-building: A flow that builds a queue. If it is classified into the Low-Latency queue, it is therefore a candidate for the Queue Protection algorithm to detect and sanction.

ECN: Explicit Congestion Notification [RFC3168]

ECN marking or ECN signalling: Setting the IP-ECN field of an increasing proportion of packets to the Congestion Experienced (CE) codepoint [RFC3168] as queuing worsens.

QProt: The Queue Protection function

2. Approach (In Brief)

The QProt algorithm is divided into mechanism and policy. There is only a tiny amount of policy code, but policy might need to be changed in the future. So, where hardware implementation is being considered, it would be advisable to implement the policy aspects in firmware or software:

- The mechanism aspects identify flows, maintain flow state, and accumulate per-flow queuing scores;
- The policy aspects 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 by reclassifying 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 that decays over time. Indeed, it is transformed into time units so that it represents 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. Thus, only queue-building flows hold flow state persistently.

The simplicity and effectiveness of the algorithm is due to the definition of the queuing score. The queueing score represents the share of blame for queuing that each flow bears. The scoring algorithm uses the same internal variable, probNative, that the AQM for the low-latency queue uses to ECN-mark packets. (The other two forms of marking, Classic and coupled, are driven by Classic traffic; therefore, they are not relevant to protection of the LL queue). In this way, the queuing score accumulates the size of each arriving packet of a flow but scaled by the value of probNative (in the range 0 to 1) at the instant the packet arrives. Therefore, a flow's score accumulates faster:

- the higher the degree of queuing and
- the faster the flow's packets arrive when there is queuing.

Section 5.1 explains further why this score represents blame for queuing.

The algorithm, as described so far, would accumulate a number that would rise at the so-called Congestion-rate of the flow (see Section 1.2), 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 reduced (or "aged") at a constant rate. This is because it is unavoidable for capacity-seeking flows to induce a continuous low level of congestion as they track available capacity. Section 5.2 explains why this allowance can be set to the same constant for any scalable flow, whatever its bit rate.

For implementation efficiency, the queuing score is transformed into time units. It then represents the expiry time of the flow state (as already discussed above). Then, it does not need to be explicitly aged because the natural passage of time implicitly "ages" an expiry time. The transformation into time units simply involves dividing the queuing score of each packet by the constant aging rate (this is explained further in Section 5.3).

2.2. Policy

2.2.1. Policy Conditions

The algorithm uses the queuing score to determine whether to eject each packet only at the time it first arrives. This limits the policies available. For instance, when queueing delay exceeds a threshold, it is not feasible 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, one were still in the queue). Nonetheless, it is still possible to develop a policy that protects the low latency of the queue by making the queuing score threshold stricter the greater the excess of queuing delay relative to the threshold (this is explained in Section 5.4).

2.2.2. Policy Action

At the time of writing, the DOCSIS QProt specification states that, when the policy conditions are met, the action taken to protect the low-latency queue is to reclassify a packet into the Classic queue (this is justified in Section 5.5).

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 more smoothly 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.

Responsive low-latency flows can use a Low Latency, Low Loss, and Scalable throughput (L4S) ECN codepoint [RFC9331] to get classified into the low-latency queue.

A sender can arrange for flows that are smooth but do not respond to ECN marking to be classified into the low-latency queue by using the Non-Queue-Building (NQB) Diffserv codepoint [RFC9956], which the DOCSIS specifications support, or an operator can use various other local classifiers.

As already explained in Section 2.1, the QProt algorithm is driven from the same variable that drives the ECN-marking probability in the Low-Latency or "LL" queue (the "Native" AQM of the LL queue is defined in the Immediate Active Queue Management Annex of [DOCSIS]). The algorithm that calculates this internal variable is run on the arrival of every LL packet, whether or not it is ECN-capable, 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 Mb/s). Note that it is in a congestion controller's own interest to keep its average Congestion-rate well below this level (e.g., ~1 Mb/s) to ensure that it does not trigger queue protection during transient dynamics.

If the QProt algorithm is used in other settings, it would still need to be based on the visible level of congestion signalling, in a similar way to the DOCSIS approach. 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

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 [DOCSIS-CM-OSS] and for CMTSs in [DOCSIS-CCAP-OSS]. Then, further constants are either derived from the input parameters or hard-coded.

Defaults and units are shown in square brackets. Defaults (or indeed any aspect of the QProt algorithm) are subject to change, so the latest DOCSIS specifications are the definitive references. Also, any operator might set certain parameters to non-default values.

```
<CODE BEGINS>
// Input Parameters
MAX_RATE;          // Configured maximum sustained rate [b/s]
QPROTECT_ON;       // Queue Protection is enabled [Default: TRUE]
CRITICALqL_us;     // LL queue threshold delay [µs] Default: MAXTH_us
CRITICALqLSCORE_us;// The threshold queuing score [Default: 4000 µs]
LG_AGING;          // The aging rate of the q'ing score [Default: 19]
                   //  as log base 2 of the Congestion-rate [lg(B/s)]

// Input Parameters for the calcProbNative() algorithm:
MAXTH_us;          // Max LL AQM marking threshold [Default: 1000 µs]
LG_RANGE;          // Log base 2 of the range of ramp [lg(ns)]
                   //  Default: 2^19 = 524288 ns (roughly 525 µs)
<CODE ENDS>
```

```
<CODE BEGINS>
// Constants, either derived from input parameters or hard-coded
T_RES;                                  // Resolution of t_exp [ns]
                                        // Convert units (approx)
AGING = pow(2, (LG_AGING-30) ) * T_RES;   // lg([B/s]) to [B/T_RES]
CRITICALqL = CRITICALqL_us * 1000;        // [µs] to [ns]
CRITICALqLSCORE = CRITICALqLSCORE_us * 1000/T_RES; // [µs] to [T_RES]
// Threshold for the q'ing score condition
CRITICALqLPRODUCT = CRITICALqL * CRITICALqLSCORE;
qLSCORE_MAX = 5E9 / T_RES;           // Max queuing score = 5 s

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, with BI_SIZE LSBs set

                       // 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]
MAX_FRAME_SIZE = 2000;  // DOCSIS-wide constant [B]
// 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]
<CODE ENDS>
```

Throughout the pseudocode, most variables are integers. The only exceptions are floating-point variables representing probabilities (MAX_PROB and probNative) and the AGING parameter. The actual DOCSIS QProt algorithm is defined using integer arithmetic, but in the floating-point arithmetic used in this document, 0 <= probNative <= 1. Also, the pseudocode omits overflow checking and it would need to be made robust to non-default input parameters.

The resolution for expressing time, T_RES, needs to be chosen to ensure that expiry times for buckets can represent times that are a fraction (e.g., 1/10) of the expected packet interarrival time for the system.

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

```
<CODE BEGINS>
// Public variables:
qdelay;        // The current queuing delay of the LL queue [ns]
probNative;    // Native marking probability of 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]
<CODE ENDS>
```

Pseudocode for how the algorithm categorizes packets by flow ID to populate the variable packet.uflow is not given in detail here. The application's flow ID is usually defined by a common 5-tuple (or 4-tuple) of:

- source and destination IP addresses of the innermost IP header found;
- the protocol (IPv4) or next header (IPv6) field in this IP header
- either of:
  - source and destination port numbers, for TCP, UDP, UDP-Lite, Stream Control Transmission Protocol (SCTP), Datagram Congestion Control Protocol (DCCP), etc.
  - Security Parameter Index (SPI) for IPsec Encapsulating Security Payload (ESP) [RFC4303].

The Microflow Classification section of the Queue Protection Annex of the DOCSIS specification [DOCSIS] defines various strategies to find these headers by skipping extension headers or encapsulations. If they cannot be found, the specification defines various less-specific 3-tuples that would be used. The DOCSIS specification should be referred to for all these strategies, which will not be repeated here.

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

```
<CODE BEGINS>
struct bucket { // The leaky bucket structure to hold per-flow state
   id;          // identifier (e.g., 5-tuple) of flow using bucket
   t_exp;       // expiry time in units of T_RES
                // (t_exp - now) = flow's transformed q'ing score
};
struct bucket buckets[NBUCKETS+1];
<CODE ENDS>
```

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 ECN-marking probability of the
                             * Native LL AQM */
```

The following function is primarily concerned with policy:

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

('...' suppresses distracting detail.)

Future modifications to policy aspects are more likely than modifications to mechanisms. Therefore, policy aspects would be less appropriate candidates for any hardware acceleration.

The entry point to these functions is qprotect(), which is called from packet classification before each packet is enqueued into the appropriate queue, queue_id, as follows:

```
<CODE BEGINS>
classifier(packet) {
   // Determine which queue using ECN, DSCP, and any local-use fields
   queue_id = classify(packet);
   //  LQ & CQ are macros for valid queue IDs returned by classify()
   if (queue_id == LQ) {
      // if packet classified to Low-Latency Service Flow
      if (QPROTECT_ON) {
         if (qprotect(packet, ...) == EXIT_SANCTION) {
            // redirect packet to Classic Service Flow
            queue_id = CQ;
         }
      }
   return queue_id;
}
<CODE ENDS>
```

4.2.1. The qprotect() Function

On each packet arrival at the Low-Latency (LL) queue, qprotect() measures the current delay of the LL queue and derives the Native LL marking 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.

```
<CODE BEGINS>
// Per-packet Queue Protection
qprotect(packet, ...) {

   bckt_id;   // bucket index
   qLscore;   // queuing score of pkt's flow in units of T_RES

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

   bckt_id = pick_bucket(packet.uflow);
   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 ) )
      // or qLSCORE_MAX reached
      || ( qLscore >= qLSCORE_MAX ) )

      return EXIT_SANCTION;

   else
      return EXIT_SUCCESS;
}
<CODE ENDS>
```

4.2.2. The pick_bucket() Function

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 a packet of a flow arrives, the algorithm tries up to ATTEMPTS times to hash to a bucket, looking for the flow's own ID. If found, it uses that bucket, first resetting the expiry time to "now" if it has expired.

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 another flow's 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 look for a bucket assigned to the flow ID. Then, if a live bucket is not already

associated with the packet's flow, the algorithm should 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 mismarked 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.

```
<CODE BEGINS>
// Pick the bucket associated with flow uflw
pick_bucket(uflw) {

   now;                      // current time
   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();   // in units of T_RES

   // 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;                    // Either way, 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;
}
<CODE ENDS>
```

4.2.3. The fill_bucket() Function

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 transformed into units of time by dividing by AGING so that the result represents the new expiry time of the flow.

Given that probNative is already used to select which packets to ECN-mark, it might be thought that the queuing score could just be incremented by the full size of each selected packet, instead of incrementing it by the product of every packet's size (pkt_sz) and probNative. However, the unpublished experience of one of the authors 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.

```
<CODE BEGINS>
fill_bucket(bckt_id, pkt_sz, probNative) {
   now;                                          // current time
   now = get_time_now();                         // in units of T_RES
   // Add packet's queuing score
   // For integer arithmetic, a bit-shift can replace the division
   qLscore = min(buckets[bckt_id].t_exp - now
                 + probNative * pkt_sz / AGING, qLSCORE_MAX);
   buckets[bckt_id].t_exp = now + qLscore;
   return qLscore;
}
<CODE ENDS>
```

4.2.4. The calcProbNative() Function

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

```
<CODE BEGINS>
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;
}
<CODE ENDS>
```

5. Rationale

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

Figure 1 shows the bit rates of two flows as stacked areas. It 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 were not there, or ran slightly more slowly, b would not cause any queuing.

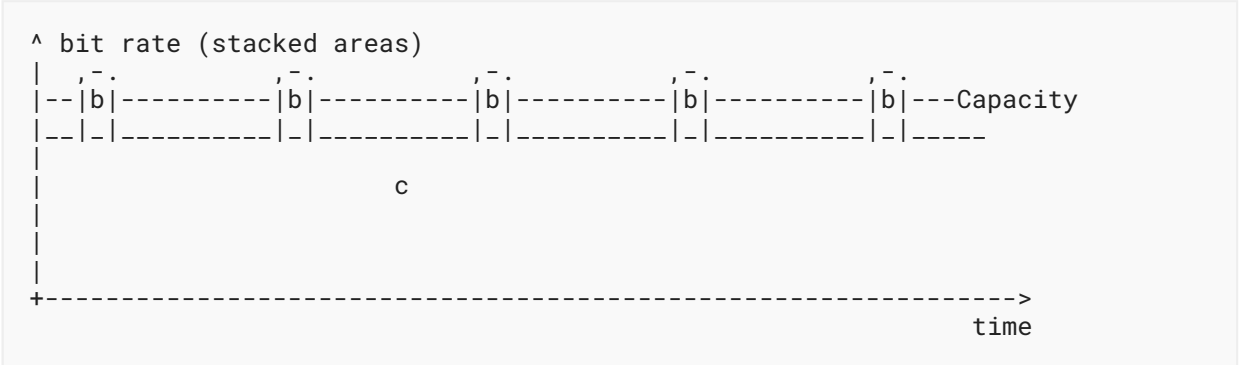

Figure 1: Which is more to blame for queuing delay?

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 rate of each flow and the level of congestion, both measured at the instant each packet arrives. 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 level of congestion is normalized to a dimensionless number between 0 and 1 (probNative). This fractional congestion level is used in preference to a direct dependence on queuing delay for two reasons:

- to be able to ignore very low levels of queuing that contribute insignificantly to delay
- to be able to erect a steep barrier against excessive queuing delay

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 25% overload, they each bear (80/125)% and (45/125)% of the

responsibility for the queuing delay (64% and 36%). The algorithm does not explicitly 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.

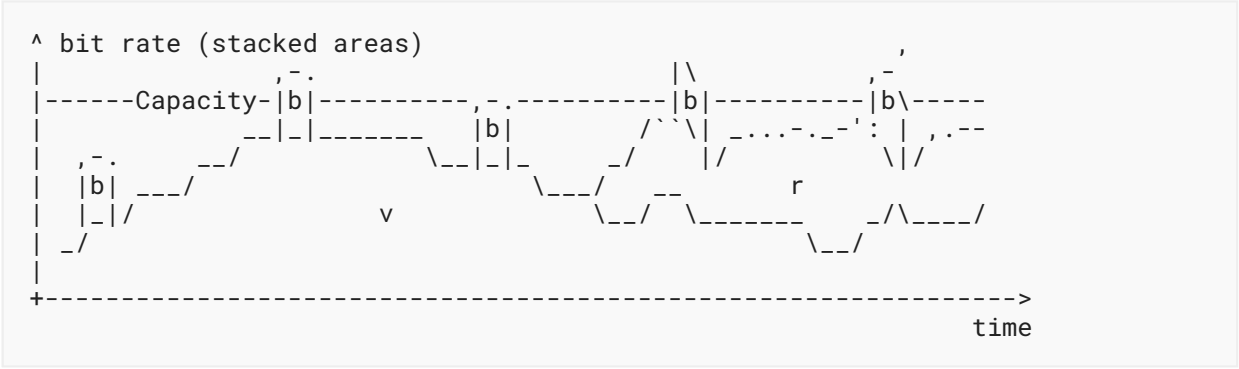

Figure 2: Responsibility for Queuing: A More Complex Scenario

Figure 2 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 figure shows the bit rates of three flows represented as stacked areas labelled b, v, and r. The unresponsive bursts (b) are the same as in the previous example, but a variable-rate video (v) replaces flow c. Its 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 timescale it appears to be unresponsive to small amounts of queuing. Also, partway 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 a lower score than the fourth because the queuing delay is not as excessive.

5.2. Rationale: Constant Aging of the Queuing Score

Even well-behaved flows will not always be able to respond fast enough to dynamic events. Also, well-behaved flows, e.g., Data Center TCP (DCTCP) [RFC8257], TCP Prague [PRAGUE-CC], Bottleneck Bandwidth and Round-trip propagation time version 3 (BBRv3) [BBRv3], or the L4S variant of SCReAM [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 they would use on their own and increase the congestion level by 3x. Therefore, in steady state, a scalable flow will induce the same constant amount of "congested-bytes" per round trip, whatever the link capacity and no matter how many flows are sharing the capacity.

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 above 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 aggressive. 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: Transformed Queuing Score

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

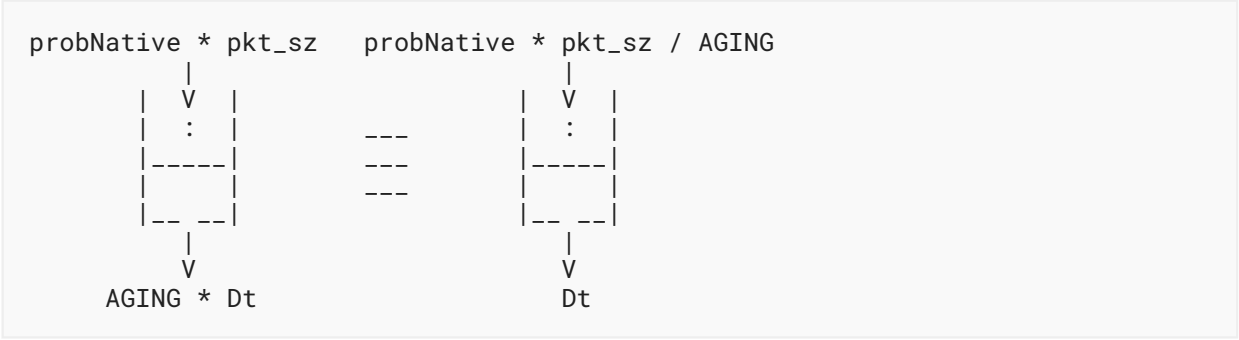

Figure 3: Transformation of Queuing Score

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 transformed 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: 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:

```
<CODE BEGINS>
   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
<CODE ENDS>
```

If the queue delay threshold is exceeded, the flow's queuing score is temporarily scaled up by the ratio of the current queue delay to the threshold queuing delay, CRITICALqL (the reason for the scaling is given next). 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 CRITICALqL 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 the extra per-packet processing cost of searching the buckets for the flow with the highest queuing score and searching the queue for one of its packets to eject (if one is still in the queue).

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.

> Note that, by default, CRITICALqL_us is set to the maximum threshold of the ramp marking algorithm MAXTH_us. However, there is some debate as to whether setting it to the minimum threshold instead would improve QProt performance. This would roughly double the ratio of qdelay to CRITICALqL, which is compared against the CRITICALqLSCORE threshold. Therefore, the threshold would have to be roughly doubled accordingly.

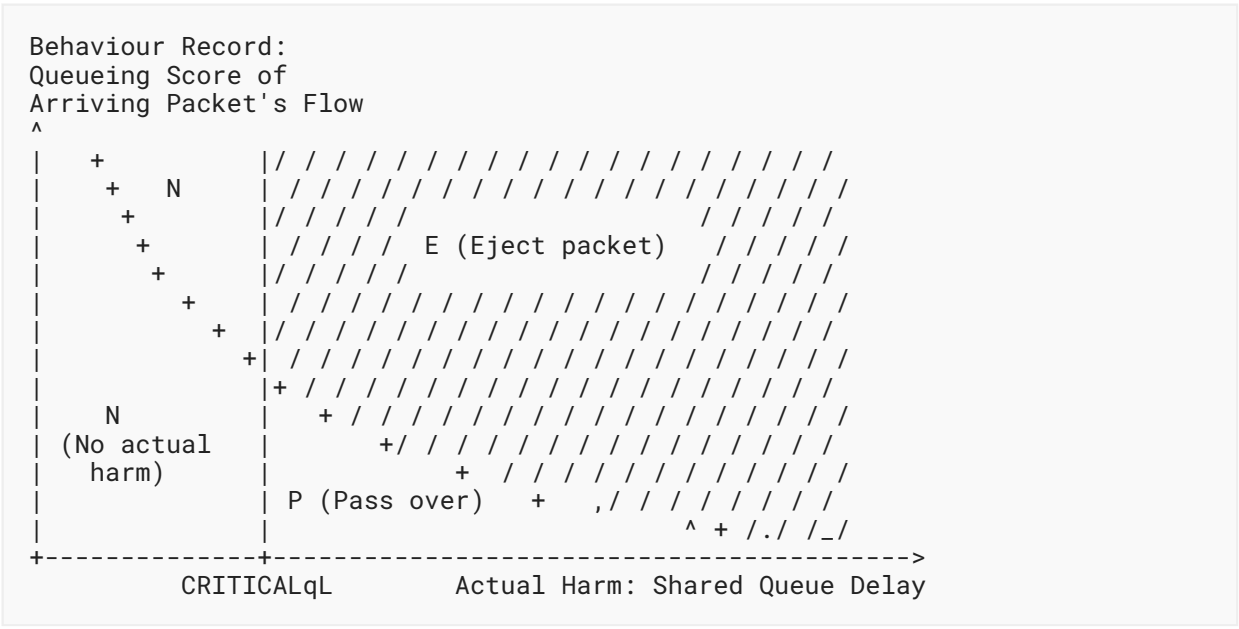

Figure 4: Graphical Explanation of the Policy Conditions

The regions labelled "N" represent 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 risk sanctioning 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.

Conditioning ejection on actual harm helps prevent VPN packets being ejected unnecessarily. VPNs consisting of multiple flows can tend to accumulate queuing score faster than it is aged out because the aging rate is intended for a single flow. However, whether or not some traffic is in a VPN, the queue delay threshold (CRITICALqL) will be no more likely to be exceeded. Therefore, conditioning ejection on actual harm helps reduce the chance that VPN traffic will be ejected by the QProt function.

5.5. Rationale: Reclassification as the Policy Action

When the DOCSIS QProt algorithm deems that it is necessary to eject a packet to protect the Low-Latency queue, it redirects the packet to the Classic queue. In the Low-Latency DOCSIS architecture (as in Dual-Queue Coupled AQMs generally), the Classic queue is expected to

frequently have a larger backlog of packets, which is caused by classic congestion controllers interacting with a classic AQM (which has a latency target of 10 ms) as well as other bursty traffic.

Therefore, typically, an ejected packet will experience higher queuing delay than it would otherwise, and it could be re-ordered within its flow (assuming QProt does not eject all packets of an anomalous flow). The mild harm caused to the performance of the ejected packet's flow is deliberate. It gives senders a slight incentive to identify their packets correctly.

If there were no such harm, there would be nothing to prevent all flows from identifying themselves as suitable for classification into the low-latency queue and just letting QProt sort the resulting aggregate into queue-building and non-queue-building flows. This might seem like a useful alternative to requiring senders to correctly identify their flows. However, handling of misclassified flows is not without a cost. The more packets that have to be reclassified, the more often the delay of the low-latency queue would exceed the threshold. Also, more memory would be required to hold the extra flow state.

When a packet is redirected into the Classic queue, an operator might want to alter the identifier(s) that originally caused it to be classified into the Low-Latency queue so that the packet will not be classified into another low-latency queue further downstream. However, redirection of occasional packets can be due to unusually high transient load just at the specific bottleneck, not necessarily at any other bottleneck and not necessarily due to bad flow behaviour. Therefore, Section 5.4.1.2 of [RFC9331] precludes a network node from altering the end-to-end ECN field to exclude traffic from L4S treatment. Instead a local-use identifier ought to be used (e.g., Diffserv codepoint or VLAN tag) so that each operator can apply its own policy, without prejudging what other operators ought to do.

Although not supported in the DOCSIS specifications, QProt could be extended to recognize that large numbers of redirected packets belong to the same flow. This might be detected when the bucket expiry time t_exp exceeds a threshold. Depending on policy and implementation capabilities, QProt could then install a classifier to redirect a whole flow into the Classic queue, with an idle timeout to remove stale classifiers. In these "persistent offender" cases, QProt might also overwrite each redirected packet's DSCP or clear its ECN field to Not-ECT, in order to protect other potential L4S queues downstream. The DOCSIS specifications do not discuss sanctioning whole flows; further discussion is beyond the scope of the present document.

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.

This choice of identifier for grouping is pragmatic with no scientific basis. All the packets of a flow certainly pass between the same two endpoints. However, some applications might initiate multiple flows between the same endpoints, 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 endpoints. And, even if

there were a one-to-one mapping between flows and applications, there is no reason to believe that the rate at which congestion can be caused ought to be allocated on a per-application-flow basis.

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 because the algorithm does not judge responsibility for queuing delay primarily on the combined rate of a set of flows grouped under one flow ID.

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

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

Section 8.2 of [RFC9330] of the L4S architecture [RFC9330] introduces the problem of maintaining low latency by either self-restraint or enforcement and places DOCSIS Queue Protection (QProt) in context within a wider set of approaches to the problem.

8.1. Resource Exhaustion Attacks

The QProt algorithm has been designed to fail gracefully in the face of traffic crafted to overrun the resources used for the algorithm's own processing and flow state. 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 a way that the proportion of innocent (non-queue-building) flows that are incorrectly sanctioned could increase.

Incorrect sanctioning is intended not to be catastrophic; it results in more packets from well-behaved flows being redirected into the Classic queue, which introduces more reordering into innocent flows.

8.1.1. Exhausting Flow State Storage

To exhaust the number of buckets, the most efficient attack is to send enough long-running attack flows to increase the chance that an arriving flow will not find an available bucket and will, therefore, have to share the "dregs" bucket. For instance, if ATTEMPTS=2 and NBUCKETS=32, it requires about 94 attack flows, each using different port numbers, to increase the probability to 99% that an arriving flow will have to share the dregs, where it will share a high degree of redirection into the Classic queue with the remainder of the attack flows.

For an attacker to keep buckets busy, it is more efficient to hold onto them by cycling regularly through a set of port numbers (94 in the above example) rather than to keep occupying and releasing buckets with single packet flows across a much larger number of ports.

During such an attack, the coupled marking probability will have saturated at 100%. Therefore, to hold a bucket, the rate of an attack flow needs to be no less than the AGING rate of each bucket: 4 Mb/s by default. However, for an attack flow to be sure to hold on to its bucket, it would need to send somewhat faster. Thus, an attack with 100 flows would need a total force of more than 100 * 4 Mb/s = 400 Mb/s.

This attack can be mitigated (but not prevented) by increasing the number of buckets. The required attack force scales linearly with the number of buckets, NBUCKETS. Therefore, if NBUCKETS were doubled to 64, twice as many 4 Mb/s flows would be needed to maintain the same impact on innocent flows.

Probably the most effective mitigation would be to implement redirection of whole-flows once enough of the individual packets of a certain offending flow had been redirected. This would free up the buckets used to maintain the per-packet queuing score of persistent offenders. Whole-flow redirection is outside the scope of the current version of the QProt algorithm specified here, but it is briefly discussed at the end of Section 5.5.

It might be considered that all the packets of persistently offending flows ought to be discarded rather than redirected. However, this is not recommended because attack flows might be able to pervert whole-flow discard, turning it onto at least some innocent flows, thus amplifying an attack that causes reordering into total deletion of some innocent flows.

8.1.2. Exhausting Processing Resources

The processing time needed to apply the QProt algorithm to each Low-Latency (LL) packet is small and intended not to take all the time available between each of a run of fairly small packets. However, an attack could use minimum sized packets launched from multiple input interfaces into a lower capacity output interface. Whether the QProt algorithm is vulnerable to processor exhaustion will depend on the specific implementation.

Addition of a capability to redirect persistently offending flows from LL to Classic would be the most effective way to reduce the per-packet processing cost of the QProt algorithm when under attack. As already mentioned in Section 8.1.1, this would also be an effective way to mitigate flow-state exhaustion attacks. Further discussion of whole-flow redirection is outside the scope of the present document but is briefly discussed at the end of Section 5.5.

9. Comments Solicited

Evaluation and assessment of the algorithm by researchers is solicited. Comments and questions are also encouraged and welcome. They can be addressed to the authors.

10. References

10.1. Normative References

[DOCSIS]

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, 21 January 2019, <https://www.cablelabs.com/specifications/CM-SP-MULPIv3.1>.

[DOCSIS-CCAP-OSS] CableLabs, "CCAP Operations Support System Interface Specification", Data-Over-Cable Service Interface Specifications DOCSIS(r) 3.1 Version I14 or later, 7 February 2019, <https://www.cablelabs.com/specifications/CM-SP-CCAP-OSSIv3.1>.

[DOCSIS-CM-OSS] CableLabs, "Cable Modem Operations Support System Interface Specification", Data-Over-Cable Service Interface Specifications DOCSIS(r) 3.1 Version I14 or later, 21 January 2019, <https://www.cablelabs.com/specifications/CM-SP-CM-OSSIv3.1>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[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>.

[RFC9331] De Schepper, K. and B. Briscoe, Ed., "The Explicit Congestion Notification (ECN) Protocol for Low Latency, Low Loss, and Scalable Throughput (L4S)", RFC 9331, DOI 10.17487/RFC9331, January 2023, <https://www.rfc-editor.org/info/rfc9331>.

[RFC9956] White, G., Fossati, T., and R. Geib, "A Non-Queue-Building Per-Hop Behavior (NQB PHB) for Differentiated Services", RFC 9956, DOI 10.17487/RFC9956, April 2026, <https://www.rfc-editor.org/info/rfc9956>.

10.2. Informative References

[BBRv3] "TCP BBR v3 Release", commit 90210de, 18 March 2025, <https://github.com/google/bbr/blob/v3/README.md>.

[LLD] White, G., Sundaresan, K., and B. Briscoe, "Low Latency DOCSIS: Technology Overview", CableLabs White Paper, February 2019, <https://cablela.bs/low-latency-docsis-technology-overview-february-2019>.

[PRAGUE-CC] De Schepper, K., Tilmans, O., Briscoe, B., and V. Goel, "Prague Congestion Control", Work in Progress, Internet-Draft, draft-briscoe-iccrg-prague-congestion-control-04, 24 July 2024, <https://datatracker.ietf.org/doc/html/draft-briscoe-iccrg-prague-congestion-control-04>.

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

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[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>.

[RFC9330] Briscoe, B., Ed., De Schepper, K., Bagnulo, M., and G. White, "Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture", RFC 9330, DOI 10.17487/RFC9330, January 2023, <https://www.rfc-editor.org/info/rfc9330>.

[RFC9332] De Schepper, K., Briscoe, B., Ed., and G. White, "Dual-Queue Coupled Active Queue Management (AQM) for Low Latency, Low Loss, and Scalable Throughput (L4S)", RFC 9332, DOI 10.17487/RFC9332, January 2023, <https://www.rfc-editor.org/info/rfc9332>.

[RFC9438] Xu, L., Ha, S., Rhee, I., Goel, V., and L. Eggert, Ed., "CUBIC for Fast and Long-Distance Networks", RFC 9438, DOI 10.17487/RFC9438, August 2023, <https://www.rfc-editor.org/info/rfc9438>.

[ScalingCC] Briscoe, B. and K. De Schepper, "Resolving Tensions between Congestion Control Scaling Requirements", arXiv:1904.07605, Simula Technical Report TR-CS-2016-001, DOI 10.48550/arXiv.1904.07605, July 2017, <https://arxiv.org/abs/1904.07605>.

[SCReAM] "SCReAM", commit 0208f59, 10 November 2025, <https://github.com/EricssonResearch/scream/blob/master/README.md>.

Acknowledgements

Thanks to Tom Henderson, Magnus Westerlund, David Black, Adrian Farrel, and Gorry Fairhurst for their reviews 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. CableLabs funded Bob Briscoe's initial work on this document.

Authors' Addresses

Bob Briscoe (EDITOR)
Independent
United Kingdom
Email: ietf@bobbriscoe.net
URI: https://bobbriscoe.net/

Greg White
CableLabs
United States of America
Email: G.White@CableLabs.com