DualQ Coupled AQMs for Low Latency, Low Loss and Scalable Throughput (L4S)
draft-ietf-tsvwg-aqm-dualq-coupled-10

Abstract

   The Low Latency Low Loss Scalable Throughput (L4S) architecture
   allows data flows over the public Internet to achieve consistent
   ultra-low queuing latency, generally zero congestion loss and scaling
   of per-flow throughput without the scaling problems of traditional
   TCP. To achieve this, L4S data flows have to use one of the family
   of 'Scalable' congestion controls (Data Centre TCP and TCP Prague are
   examples) and a form of Explicit Congestion Notification (ECN) with
   modified behaviour. However, until now, Scalable congestion controls
   did not co-exist with existing TCP Reno/Cubic traffic --- Scalable
   controls are so aggressive that 'Classic' TCP algorithms drive
   themselves to a small capacity share. Therefore, until now, L4S
   controls could only be deployed where a clean-slate environment could
   be arranged, such as in private data centres (hence the name DCTCP).
   This specification defines `DualQ Coupled Active Queue Management
   (AQM)', which enables these Scalable congestion controls to safely
   co-exist with Classic Internet traffic.

   Analytical study and implementation testing of the Coupled AQM have
   shown that Scalable and Classic flows competing under similar
   conditions run at roughly the same rate. It achieves this
   indirectly, without having to inspect transport layer flow
   identifiers. When tested in a residential broadband setting, DCTCP
   also achieves sub-millisecond average queuing delay and zero
   congestion loss under a wide range of mixes of DCTCP and `Classic'
   broadband Internet traffic, without compromising the performance of
   the Classic traffic. The solution also reduces network complexity
   and requires no configuration for the public Internet.

Status of This Memo

Table of Contents

1. Introduction

This document specifies a framework for DualQ Coupled AQMs, which is
the network part of the L4S architecture [I-D.ietf-tsvwg-l4s-arch].
L4S enables both ultra-low queuing latency and high throughput at the
same time, for ad hoc numbers of capacity-seeking applications all
sharing the same capacity.

1.1. Outline of the Problem

Latency is becoming the critical performance factor for many (most?)
applications on the public Internet, e.g. interactive Web, Web
services, voice, conversational video, interactive video, interactive
remote presence, instant messaging, online gaming, remote desktop,
cloud-based applications, and video-assisted remote control of
machinery and industrial processes. In the developed world, further
increases in access network bit-rate offer diminishing returns,
whereas latency is still a multi-faceted problem. In the last decade
or so, much has been done to reduce propagation time by placing
caches or servers closer to users. However, queuing remains a major
intermittent component of latency.

Traditionally ultra-low latency has only been available for a few
selected low rate applications, that confine their sending rate
within a specially carved-off portion of capacity, which is
prioritized over other traffic, e.g. Diffserv EF [RFC3246]. Up to
now it has not been possible to allow any number of low latency, high

throughput applications to seek to fully utilize available capacity, because the capacity-seeking process itself causes too much queuing delay.

To reduce this queuing delay caused by the capacity seeking process, changes either to the network alone or to end-systems alone are in progress. L4S involves a recognition that both approaches are yielding diminishing returns:

o Recent state-of-the-art active queue management (AQM) in the network, e.g. fq_CoDel [RFC8290], PIE [RFC8033], Adaptive RED [ARED01]) has reduced queuing delay for all traffic, not just a select few applications. However, no matter how good the AQM, the capacity-seeking (sawtoothing) rate of TCP-like congestion controls represents a lower limit that will either cause queuing delay to vary or cause the link to be under-utilized. These AQMs are tuned to allow a typical capacity-seeking TCP-Friendly flow to induce an average queue that roughly doubles the base RTT, adding 5-15 ms of queuing on average (cf. 500 microseconds with L4S for the same mix of long-running and web traffic). However, for many applications low delay is not useful unless it is consistently low. With these AQMs, 99th percentile queuing delay is 20-30 ms (cf. 2 ms with the same traffic over L4S).

o Similarly, recent research into using e2e congestion control without needing an AQM in the network (e.g.BBRv1 [BBRv1]) seems to have hit a similar lower limit to queuing delay of about 20ms on average (and any additional BBRv1 flow adds another 20ms of queuing) but there are also regular 25ms delay spikes due to bandwidth probes and 60ms spikes due to flow-starts.

L4S learns from the experience of Data Center TCP [RFC8257], which shows the power of complementary changes both in the network and on end-systems. DCTCP teaches us that two small but radical changes to congestion control are needed to cut the two major outstanding causes of queuing delay variability:

1. Far smaller rate variations (sawteeth) than TCP-Friendly congestion controls;

2. A shift of smoothing and hence smoothing delay from network to sender.

Without the former, a 'Classic' flow's round trip time (RTT) varies between roughly 1 and 2 times the base RTT between the machines in question. Without the latter a 'Classic' flow's response to changing events is delayed by a worst-case (transcontinental) RTT, which could

be hundreds of times the actual smoothing delay needed for the RTT of typical traffic from localized CDNs.

These changes are the two main features of the family of so-called 'Scalable' congestion controls (which includes DCTCP). Both these changes only reduce delay in combination with a complementary change in the network and they are both only feasible with ECN, not drop, for the signalling:

1. The smaller sawteeth need an extremely shallow ECN packet-marking threshold in the queue.

2. And no smoothing in the network means that every fluctuation of the queue is signalled immediately.

Without ECN, either of these would lead to very high loss levels. But, with ECN, the resulting high marking levels are fine.

However, until now, Scalable congestion controls (like DCTCP) did not co-exist with existing ECN-capable TCP Reno [RFC5681] or Cubic [RFC8312] traffic --- Scalable controls are so aggressive that these 'Classic' TCP algorithms drive themselves to a small capacity share. Therefore, until now, L4S controls could only be deployed where a clean-slate environment could be arranged, such as in private data centres (hence the name DCTCP).

This document specifies a `DualQ Coupled AQM' extension that solves the problem of coexistence between Scalable and Classic flows, without having to inspect flow identifiers. It is not like flow-queuing approaches [RFC8290] that classify packets by flow identifier into separate queues in order to isolate sparse flows from the higher latency in the queues assigned to heavier flows. If a flow needs both low delay and high throughput, having a queue to itself does not isolate it from the harm it causes to itself. In contrast, L4S addresses the root cause of the latency problem --- it is an enabler for the smooth low latency scalable behaviour of Scalable congestion controls, so that every packet in every flow can enjoy very low latency, then there is no need to isolate each flow into a separate queue.

1.2. Scope

L4S involves complementary changes in the network and on end-systems:

Network: A DualQ Coupled AQM (defined in the present document);

End-system: A Scalable congestion control (defined in Section 2.1.

Packet identifier: The network and end-system parts of L4S can be
   deployed incrementally, because they both identify L4S packets
   using the experimentally assigned explicit congestion notification
   (ECN) codepoints in the IP header: ECT(1) and CE [RFC8311]
   [I-D.ietf-tsvwg-ecn-l4s-id].

Data Center TCP (DCTCP [RFC8257]) is an example of a Scalable
congestion control that has been deployed for some time in Linux,
Windows and FreeBSD operating systems and Relentless TCP [Mathis09]
is another example. During the progress of this document through the
IETF a number of other Scalable congestion controls were implemented,
e.g. TCP Prague [PragueLinux], QUIC Prague and the L4S variant of
SCREAM for real-time media [RFC8298]. (Note: after the v3.19 Linux
kernel, bugs were introduced into DCTCP's scalable behaviour and not
all the patches applied for L4S evaluation had been applied to the
mainline Linux kernel, which was at v5.2 at the time of writing).

The focus of this specification is to get the network part of the L4S
service in place. Then, without any management intervention,
applications can exploit this new network capability as their
operating systems migrate to Scalable congestion controls, which can
then evolve _while_ their benefits are being enjoyed by everyone on
the Internet.

The DualQ Coupled AQM framework can incorporate any AQM designed for
a single queue that generates a statistical or deterministic mark/
drop probability driven by the queue dynamics. Pseudocode examples
of two different DualQ Coupled AQMs are given the appendices. In
many cases the framework simplifies the basic control algorithm, and
requires little extra processing. Therefore it is believed the
Coupled AQM would be applicable and easy to deploy in all types of
buffers; buffers in cost-reduced mass-market residential equipment;
buffers in end-system stacks; buffers in carrier-scale equipment
including remote access servers, routers, firewalls and Ethernet
switches; buffers in network interface cards, buffers in virtualized
network appliances, hypervisors, and so on.

For the public Internet, nearly all the benefit will typically be
achieved by deploying the Coupled AQM into either end of the access
link between a 'site' and the Internet, which is invariably the
bottleneck. Here, the term 'site' is used loosely to mean a home, an
office, a campus or mobile user equipment.

Latency is not the only concern of L4S:

o The 'Low Loss" part of the name denotes that L4S generally
   achieves zero congestion loss (which would otherwise cause
   retransmission delays), due to its use of ECN.

o   The "Scalable throughput" part of the name denotes that the per-
    flow throughput of Scalable congestion controls should scale
    indefinitely, avoiding the imminent scaling problems with TCP-
    Friendly congestion control algorithms [RFC3649].

The former is clearly in scope of this AQM document. However, the
latter is an outcome of the end-system behaviour, and therefore
outside the scope of this AQM document, even though the AQM is an
enabler.

The overall L4S architecture [I-D.ietf-tsvwg-l4s-arch] gives more
detail, including on wider deployment aspects such as backwards
compatibility of Scalable congestion controls in bottlenecks where a
DualQ Coupled AQM has not been deployed. The supporting papers [PI2]
and [DCttH15] give the full rationale for the AQM's design, both
discursively and in more precise mathematical form.

1.3. 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] when, and
only when, they appear in all capitals, as shown here.

The DualQ Coupled AQM uses two queues for two services. Each of the
following terms identifies both the service and the queue that
provides the service:

Classic (denoted by subscript C): The `Classic' service is intended
    for all the behaviours that currently co-exist with TCP Reno (TCP
    Cubic, Compound, SCTP, etc).

Low-Latency, Low-Loss and Scalable (L4S, denoted by subscript L):
    The `L4S' service is intended for a set of congestion controls
    with scalable properties, such as TCP Prague and DCTCP. For the
    public Internet an L4S transport has to comply with the
    requirements in Section 4 of [I-D.ietf-tsvwg-ecn-l4s-id] (aka.
    the 'Prague L4S requirements').

Either service can cope with a proportion of unresponsive or less-
responsive traffic as well, as long (e.g. DNS, VoIP, game sync
datagrams, etc), just as a single queue AQM can if this traffic makes
minimal contribution to queuing. The DualQ Coupled AQM behaviour
below is defined to be similar to a single FIFO queue with respect to
unresponsive and overload traffic.

1.4. Features

   The AQM couples marking and/or dropping from the Classic queue to the
   L4S queue in such a way that a flow will get roughly the same
   throughput whichever it uses. Therefore both queues can feed into
   the full capacity of a link and no rates need to be configured for
   the queues. The L4S queue enables Scalable congestion controls like
   DCTCP or TCP Prague to give stunningly low and predictably low
   latency, without compromising the performance of competing 'Classic'
   Internet traffic.

   Thousands of tests have been conducted in a typical fixed residential
   broadband setting. Experiments used a range of base round trip
   delays up to 100ms and link rates up to 200 Mb/s between the data
   centre and home network, with varying amounts of background traffic
   in both queues. For every L4S packet, the AQM kept the average
   queuing delay below 1ms (or 2 packets where serialization delay
   exceeded 1ms on slower links), with 99th percentile no worse than
   2ms. No losses at all were introduced by the L4S AQM. Details of
   the extensive experiments are available [PI2] [DCttH15].

   Subjective testing was also conducted by multiple people all
   simultaneously using very demanding high bandwidth low latency
   applications over a single shared access link [L4Sdemo16]. In one
   application, each user could use finger gestures to pan or zoom their
   own high definition (HD) sub-window of a larger video scene generated
   on the fly in 'the cloud' from a football match. Another user
   wearing VR goggles was remotely receiving a feed from a 360-degree
   camera in a racing car, again with the sub-window in their field of
   vision generated on the fly in 'the cloud' dependent on their head
   movements. Even though other users were also downloading large
   amounts of L4S and Classic data, playing a gaming benchmark and
   watchings videos over the same 40Mb/s downstream broadband link,
   latency was so low that the football picture appeared to stick to the
   user's finger on the touchpad and the experience fed from the remote
   camera did not noticeably lag head movements. All the L4S data (even
   including the downloads) achieved the same ultra-low latency. With
   an alternative AQM, the video noticeably lagged behind the finger
   gestures and head movements.

   Unlike Diffserv Expedited Forwarding, the L4S queue does not have to
   be limited to a small proportion of the link capacity in order to
   achieve low delay. The L4S queue can be filled with a heavy load of
   capacity-seeking flows (TCP Prague etc.) and still achieve low delay.
   The L4S queue does not rely on the presence of other traffic in the
   Classic queue that can be 'overtaken'. It gives low latency to L4S
   traffic whether or not there is Classic traffic, and the latency of

Classic traffic does not suffer when a proportion of the traffic is
L4S.

The two queues are only necessary because:

o the large variations (sawteeth) of Classic flows need roughly a
   base RTT of queuing delay to ensure full utilization

o while Scalable flows do not need a queue to keep utilization high,
   but they cannot keep latency predictably low if they are mixed
   with legacy TCP flows,

The L4S queue has latency priority, but the coupling from the Classic
to the L4S AQM (explained below) ensures that it does not have
bandwidth priority over the Classic queue.

2. DualQ Coupled AQM

There are two main aspects to the approach:

o the Coupled AQM that addresses throughput equivalence between
   Classic (e.g. Reno, Cubic) flows and L4S flows (that satisfy the
   Prague L4S requirements).

o the Dual Queue structure that provides latency separation for L4S
   flows to isolate them from the typically large Classic queue.

2.1. Coupled AQM

In the 1990s, the `TCP formula' was derived for the relationship
between TCP's congestion window, cwnd, and its drop probability, p.
To a first order approximation, cwnd of TCP Reno is inversely
proportional to the square root of p.

The design focuses on Reno as the worst case, because if it does no
harm to Reno, it will not harm Cubic or any traffic designed to be
friendly to Reno. TCP Cubic implements a Reno-compatibility mode,
which is relevant for typical RTTs under 20ms as long as the
throughput of a single flow is less than about 700Mb/s. In such
cases it can be assumed that Cubic traffic behaves similarly to Reno
(but with a slightly different constant of proportionality). The
term 'Classic' will be used for the collection of Reno-friendly
traffic including Cubic in Reno mode.

The supporting paper [PI2] includes the derivation of the equivalent
rate equation for DCTCP, for which cwnd is inversely proportional to
p (not the square root), where in this case p is the ECN marking
probability. DCTCP is not the only congestion control that behaves

like this, so the term 'Scalable' will be used for all similar
congestion control behaviours (see examples in Section 1.2). The
term 'L4S' is also used for traffic driven by a Scalable congestion
control that also complies with the additional 'Prague L4S'
requirements [I-D.ietf-tsvwg-ecn-l4s-id].

For safe co-existence, under stationary conditions, a Scalable flow
has to run at roughly the same rate as a Reno TCP flow (all other
factors being equal). So the drop or marking probability for Classic
traffic, p_C has to be distinct from the marking probability for L4S
traffic, p_L. [RFC8311] updates the original ECN specification
[RFC3168] to allow these probabilities to be distinct, because RFC
3168 required them to be the same.

Also, to remain stable, Classic sources need the network to smooth
p_C so it changes relatively slowly. It is hard for a network node
to know the RTTs of all the flows, so a Classic AQM adds a _worst-
case_ RTT of smoothing delay (about 100-200 ms). In contrast, L4S
shifts responsibility for smoothing ECN feedback to the sender, which
only delays its response by its _own_ RTT, and allows a more
immediate response if necessary.

The Coupled AQM achieves safe coexistence by making the Classic drop
probability p_C proportional to the square of the coupled L4S
probability p_CL. p_CL is an input to the instantaneous L4S marking
probability p_L but it changes as slowly as p_C. This makes the Reno
flow rate roughly equal the DCTCP flow rate, because the squaring of
p_CL counterbalances the square root of p_C in the Classic 'TCP
formula'.

Stating this as a formula, the relation between Classic drop
probability, p_C, and the coupled L4S probability p_CL needs to take
the form:

$$p\_C = (p\_CL / k)^2 \qquad\qquad (1)$$

where k is the constant of proportionality, which is termed the
coupling factor.

2.2. Dual Queue

Classic traffic needs to build a large queue to prevent under-
utilization. Therefore a separate queue is provided for L4S traffic,
and it is scheduled with priority over the Classic queue. Priority
is conditional to prevent starvation of Classic traffic.

Nonetheless, coupled marking ensures that giving priority to L4S
traffic still leaves the right amount of spare scheduling time for

Classic flows to each get equivalent throughput to DCTCP flows (all
other factors such as RTT being equal).

2.3. Traffic Classification

Both the Coupled AQM and DualQ mechanisms need an identifier to
distinguish L and C packets. Then the coupling algorithm can achieve
coexistence without having to inspect flow identifiers, because it
can apply the appropriate marking or dropping probability to all
flows of each type. A separate
specification [I-D.ietf-tsvwg-ecn-l4s-id] requires the sender to use
the ECT(1) and CE codepoints of the ECN field as this identifier,
having assessed various alternatives. An additional process document
has proved necessary to make the ECT(1) codepoint available for
experimentation [RFC8311].

For policy reasons, an operator might choose to steer certain packets
(e.g. from certain flows or with certain addresses) out of the L
queue, even though they identify themselves as L4S by their ECN
codepoints. In such cases, [I-D.ietf-tsvwg-ecn-l4s-id] says that the
device "MUST NOT alter the end-to-end L4S ECN identifier", so that it
is preserved end-to-end. The aim is that each operator can choose
how it treats L4S traffic locally, but an individual operator does
not alter the identification of L4S packets, which would prevent
other operators downstream from making their own choices on how to
treat L4S traffic.

In addition, an operator could use other identifiers to classify
certain additional packet types into the L queue that it deems will
not risk harm to the L4S service. For instance addresses of specific
applications or hosts (see [I-D.ietf-tsvwg-ecn-l4s-id]), specific
Diffserv codepoints such as EF (Expedited Forwarding) and Voice-Admit
service classes (see [I-D.briscoe-tsvwg-l4s-diffserv]) or certain
protocols (e.g. ARP, DNS). Note that the mechanism only reads these
identifiers. [I-D.ietf-tsvwg-ecn-l4s-id] says it "MUST NOT alter
these non-ECN identifiers".

2.4. Overall DualQ Coupled AQM Structure

Figure 1 shows the overall structure that any DualQ Coupled AQM is
likely to have. This schematic is intended to aid understanding of
the current designs of DualQ Coupled AQMs. However, it is not
intended to preclude other innovative ways of satisfying the
normative requirements in Section 2.5 that minimally define a DualQ
Coupled AQM.

The classifier on the left separates incoming traffic between the two
queues (L and C). Each queue has its own AQM that determines the

likelihood of marking or dropping (p_L and p_C). It has been
proved [PI2] that it is preferable to control load with a linear
controller, then square the output before applying it as a drop
probability to TCP (because TCP decreases its load proportional to
the square-root of the increase in drop). So, the AQM for Classic
traffic needs to be implemented in two stages: i) a base stage that
outputs an internal probability p' (pronounced p-prime); and ii) a
squaring stage that outputs p_C, where

    p_C = (p')^2.   (2)

Substituting for p_C in Eqn (1) gives:

    p' = p_CL / k

So the slow-moving input to ECN marking in the L queue (the coupled
L4S probability) is:

    p_CL = k*p',   (3)

where k is the constant coupling factor (see Appendix C).

It can be seen that these two transformations of p' implement the
required coupling given in equation (1) earlier.

The actual ECN marking probability p_L that is applied to the L queue
needs to track the immediate L queue delay under L-only congestion
conditions, as well as track p_CL under coupled congestion
conditions. So the L queue uses a native AQM that calculates a
probability p'_L as a function of the instantaneous L queue delay.
And, given the L queue has conditional strict priority over the C
queue, whenever the L queue grows, the AQM should apply marking
probability p'_L, but p_L should not fall below p_CL. This suggests:

    p_L = max(p'_L, p_CL),   (4)

which has also been found to work very well in practice.

```
                       _____
                      | |    ,------.
              L4S queue | |===>| ECN  |
             ,'| _____|_|    |marker|\
            <'  |         |     `------'\\
             //`'         v        ^ p_L \\
            //      ,-------.       |      \\
           //       |Native |p'_L |        \\,.
          //        |  L4S  |--->(MAX)    <  |    ___
    ,----------.//        |  AQM  |     ^ p_CL  `\|.'Cond-`.
    |  IP-ECN  |/     `-------'       |          / itional \
==>|Classifier|          ,-------.   (k*p')      [ priority]==>
    |         |\         | Base |     |          \scheduler/
    `---------'\\        |  AQM  |---->:        ,'|`-.___.-'
             \\         |       |p'  |       <'  |
              \\        `-------'   (p'^2)    //`'
               \\          ^          |      //
                \\,.       |          v p_C //
                 <  | _____      .------.//
                  `\|   |      |    | Drop |/
              Classic |queue |===>|/mark |
                  __|_____|     `------'
```
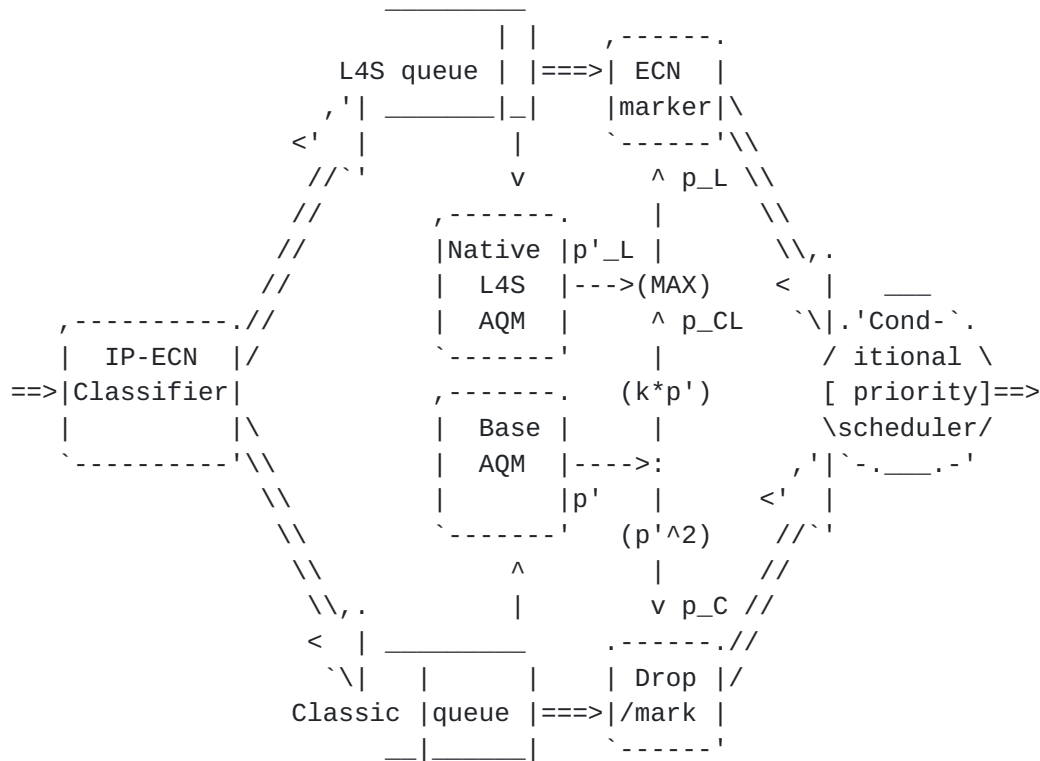
Legend: ===> traffic flow; ---> control dependency.

Figure 1: DualQ Coupled AQM Schematic

After the AQMs have applied their dropping or marking, the scheduler forwards their packets to the link, giving priority to L4S traffic. Priority has to be conditional in some way (see Section 4.1). Simple strict priority is inappropriate otherwise it could lead the L4S queue to starve the Classic queue. For example, consider the case where a continually busy L4S queue blocks a DNS request in the Classic queue, arbitrarily delaying the start of a new Classic flow.

Example DualQ Coupled AQM algorithms called DualPI2 and Curvy RED are given in Appendix A and Appendix B. Either example AQM can be used to couple packet marking and dropping across a dual Q.

DualPI2 uses a Proportional-Integral (PI) controller as the Base AQM. Indeed, this Base AQM with just the squared output and no L4S queue can be used as a drop-in replacement for PIE [RFC8033], in which case it is just called PI2 [PI2]. PI2 is a principled simplification of PIE that is both more responsive and more stable in the face of dynamically varying load.

Curvy RED is derived from RED [RFC2309], but its configuration
parameters are insensitive to link rate and it requires less
operations per packet. However, DualPI2 is more responsive and
stable over a wider range of RTTs than Curvy RED. As a consequence,
DualPI2 has attracted more development and evaluation attention than
Curvy RED, leaving the Curvy RED design incomplete and not so fully
evaluated.

Both AQMs regulate their queue in units of time rather than bytes.
As already explained, this ensures configuration can be invariant for
different drain rates. With AQMs in a dualQ structure this is
particularly important because the drain rate of each queue can vary
rapidly as flows for the two queues arrive and depart, even if the
combined link rate is constant.

It would be possible to control the queues with other alternative
AQMs, as long as the normative requirements (those expressed in
capitals) in Section 2.5 are observed.

2.5. Normative Requirements for a DualQ Coupled AQM

The following requirements are intended to capture only the essential
aspects of a DualQ Coupled AQM. They are intended to be independent
of the particular AQMs used for each queue.

2.5.1. Functional Requirements

A Dual Queue Coupled AQM implementation MUST utilize two queues, each
with an AQM algorithm. The two queues can be part of a larger
queuing hierarchy [I-D.briscoe-tsvwg-l4s-diffserv].

The AQM algorithm for the low latency (L) queue MUST be able to apply
ECN marking to ECN-capable packets.

The scheduler draining the two queues MUST give L4S packets priority
over Classic, although priority MUST be bounded in order not to
starve Classic traffic.

[I-D.ietf-tsvwg-ecn-l4s-id] defines the meaning of an ECN marking on
L4S traffic, relative to drop of Classic traffic. In order to ensure
coexistence of Classic and Scalable L4S traffic, it says, "The
likelihood that an AQM drops a Not-ECT Classic packet (p_C) MUST be
roughly proportional to the square of the likelihood that it would
have marked it if it had been an L4S packet (p_L)." The term
'likelihood' is used to allow for marking and dropping to be either
probabilistic or deterministic.

For the current specification, this translates into the following
requirement. A DualQ Coupled AQM MUST apply ECN marking to traffic
in the L queue that is no lower than that derived from the likelihood
of drop (or ECN marking) in the Classic queue using Eqn. (1).

The constant of proportionality, k, in Eqn (1) determines the
relative flow rates of Classic and L4S flows when the AQM concerned
is the bottleneck (all other factors being equal).
[I-D.ietf-tsvwg-ecn-l4s-id] says, "The constant of proportionality
(k) does not have to be standardised for interoperability, but a
value of 2 is RECOMMENDED."

Assuming Scalable congestion controls for the Internet will be as
aggressive as DCTCP, this will ensure their congestion window will be
roughly the same as that of a standards track TCP congestion control
(Reno) [RFC5681] and other so-called TCP-friendly controls, such as
TCP Cubic in its TCP-friendly mode.

The choice of k is a matter of operator policy, and operators MAY
choose a different value using Table 1 and the guidelines in
Appendix C.

If multiple customers or users share capacity at a bottleneck (e.g.
in the Internet access link of a campus network), the operator's
choice of k will determine capacity sharing between the flows of
different customers. However, on the public Internet, access network
operators typically isolate customers from each other with some form
of layer-2 multiplexing (OFDM(A) in DOCSIS3.1, CDMA in 3G, SC-FDMA in
LTE) or L3 scheduling (WRR in DSL), rather than relying on TCP to
share capacity between customers [RFC0970]. In such cases, the
choice of k will solely affect relative flow rates within each
customer's access capacity, not between customers. Also, k will not
affect relative flow rates at any times when all flows are Classic or
all flows are L4S, and it will not affect the relative throughput of
small flows.

2.5.1.1. Requirements in Unexpected Cases

The flexibility to allow operator-specific classifiers (Section 2.3)
leads to the need to specify what the AQM in each queue ought to do
with packets that do not carry the ECN field expected for that queue.
It is recommended that the AQM in each queue inspects the ECN field
to determine what sort of congestion notification to signal, then
decides whether to apply congestion notification to this particular
packet, as follows:

o If a packet that does not carry an ECT(1) or CE codepoint is
   classified into the L queue:

   * if the packet is ECT(0), the L AQM SHOULD apply CE-marking
     using a probability appropriate to Classic congestion control
     and appropriate to the target delay in the L queue

   * if the packet is Not-ECT, the appropriate action depends on
     whether some other function is protecting the L queue from
     misbehaving flows (e.g. per-flow queue protection or latency
     policing):

     + If separate queue protection is provided, the L AQM SHOULD
       ignore the packet and forward it unchanged, meaning it
       should not calculate whether to apply congestion
       notification and it should neither drop nor CE-mark the
       packet (for instance, the operator might classify EF traffic
       that is unresponsive to drop into the L queue, alongside
       responsive L4S-ECN traffic)

     + if separate queue protection is not provided, the L AQM
       SHOULD apply drop using a drop probability appropriate to
       Classic congestion control and appropriate to the target
       delay in the L queue

   o If a packet that carries an ECT(1) codepoint is classified into
     the C queue:

     * the C AQM SHOULD apply CE-marking using the coupled AQM
       probability p_CL (= k*p').

   The above requirements are worded as "SHOULDs", because operator-
   specific classifiers are for flexibility, by definition. Therefore,
   alternative actions might be appropriate in the operator's specific
   circumstances. An example would be where the operator knows that
   certain legacy traffic marked with one codepoint actually has a
   congestion response associated with another codepoint.

   If the DualQ Coupled AQM has detected overload, it MUST signal
   congestion solely using drop, irrespective of the ECN field.
   Switching to drop if ECN marking is persistently high is required by
   Section 7 of [RFC3168] and Section 4.2.1 of [RFC7567].

2.5.2. Management Requirements

2.5.2.1. Configuration

   By default, a DualQ Coupled AQM SHOULD NOT need any configuration for
   use at a bottleneck on the public Internet [RFC7567]. The following
   parameters MAY be operator-configurable, e.g. to tune for non-
   Internet settings:

   o Optional packet classifier(s) to use in addition to the ECN field
      (see Section 2.3);

   o Expected typical RTT, which can be used to determine the queuing
      delay of the Classic AQM at its operating point, in order to
      prevent typical lone TCP flows from under-utilizing capacity. For
      example:

      * for the PI2 algorithm (Appendix A) the queuing delay target is
         set to the typical RTT;

      * for the Curvy RED algorithm (Appendix B) the queuing delay at
         the desired operating point of the curvy ramp is configured to
         encompass a typical RTT;

      * if another Classic AQM was used, it would be likely to need an
         operating point for the queue based on the typical RTT, and if
         so it SHOULD be expressed in units of time.

      An operating point that is manually calculated might be directly
      configurable instead, e.g. for links with large numbers of flows
      where under-utilization by a single TCP flow would be unlikely.

   o Expected maximum RTT, which can be used to set the stability
      parameter(s) of the Classic AQM. For example:

      * for the PI2 algorithm (Appendix A), the gain parameters of the
         PI algorithm depend on the maximum RTT.

      * for the Curvy RED algorithm (Appendix B) the smoothing
         parameter is chosen to filter out transients in the queue
         within a maximum RTT.

      Stability parameter(s) that are manually calculated assuming a
      maximum RTT might be directly configurable instead.

   o Coupling factor, k;

   o A limit to the conditional priority of L4S. This is scheduler-
      dependent, but it SHOULD be expressed as a relation between the
      max delay of a C packet and an L packet. For example:

      * for a WRR scheduler a weight ratio between L and C of w:1 means
         that the maximum delay to a C packet is w times that of an L
         packet.

      * for a time-shifted FIFO (TS-FIFO) scheduler (see Section 4.1.1)
         a time-shift of tshift means that the maximum delay to a C

packet is tshift greater than that of an L packet. tshift could
be expressed as a multiple of the typical RTT rather than as an
absolute delay.

o The maximum Classic ECN marking probability, p_Cmax, before
   switching over to drop.

2.5.2.2. Monitoring

An experimental DualQ Coupled AQM SHOULD allow the operator to
monitor each of the following operational statistics on demand, per
queue and per configurable sample interval, for performance
monitoring and perhaps also for accounting in some cases:

o Bits forwarded, from which utilization can be calculated;

o Total packets in the three categories: arrived, presented to the
   AQM, and forwarded. The difference between the first two will
   measure any non-AQM tail discard. The difference between the last
   two will measure proactive AQM discard;

o ECN packets marked, non-ECN packets dropped, ECN packets dropped,
   which can be combined with the three total packet counts above to
   calculate marking and dropping probabilities;

o Queue delay (not including serialization delay of the head packet
   or medium acquisition delay) - see further notes below.

   Unlike the other statistics, queue delay cannot be captured in a
   simple accumulating counter. Therefore the type of queue delay
   statistics produced (mean, percentiles, etc.) will depend on
   implementation constraints. To facilitate comparative evaluation
   of different implementations and approaches, an implementation
   SHOULD allow mean and 99th percentile queue delay to be derived
   (per queue per sample interval). A relatively simple way to do
   this would be to store a coarse-grained histogram of queue delay.
   This could be done with a small number of bins with configurable
   edges that represent contiguous ranges of queue delay. Then, over
   a sample interval, each bin would accumulate a count of the number
   of packets that had fallen within each range. The maximum queue
   delay per queue per interval MAY also be recorded.

2.5.2.3. Anomaly Detection

An experimental DualQ Coupled AQM SHOULD asynchronously report the
following data about anomalous conditions:

o Start-time and duration of overload state.

A hysteresis mechanism SHOULD be used to prevent flapping in and
out of overload causing an event storm. For instance, exit from
overload state could trigger one report, but also latch a timer.
Then, during that time, if the AQM enters and exits overload state
any number of times, the duration in overload state is accumulated
but no new report is generated until the first time the AQM is out
of overload once the timer has expired.

2.5.2.4. Deployment, Coexistence and Scaling

[RFC5706] suggests that deployment, coexistence and scaling should
also be covered as management requirements. The raison d'etre of the
DualQ Coupled AQM is to enable deployment and coexistence of Scalable
congestion controls - as incremental replacements for today's TCP-
friendly controls that do not scale with bandwidth-delay product.
Therefore there is no need to repeat these motivating issues here
given they are already explained in the Introduction and detailed in
the L4S architecture [I-D.ietf-tsvwg-l4s-arch].

The descriptions of specific DualQ Coupled AQM algorithms in the
appendices cover scaling of their configuration parameters, e.g. with
respect to RTT and sampling frequency.

3. IANA Considerations

This specification contains no IANA considerations.

4. Security Considerations

4.1. Overload Handling

Where the interests of users or flows might conflict, it could be
necessary to police traffic to isolate any harm to the performance of
individual flows. However it is hard to avoid unintended side-
effects with policing, and in a trusted environment policing is not
necessary. Therefore per-flow policing needs to be separable from a
basic AQM, as an option under policy control.

However, a basic DualQ AQM does at least need to handle overload. A
useful objective would be for the overload behaviour of the DualQ AQM
to be at least no worse than a single queue AQM. However, a trade-
off needs to be made between complexity and the risk of either
traffic class harming the other. In each of the following three
subsections, an overload issue specific to the DualQ is described,
followed by proposed solution(s).

Under overload the higher priority L4S service will have to sacrifice
some aspect of its performance. Alternative solutions are provided

below that each relax a different factor: e.g. throughput, delay, drop. These choices need to be made either by the developer or by operator policy, rather than by the IETF.

4.1.1. Avoiding Classic Starvation: Sacrifice L4S Throughput or Delay?

Priority of L4S is required to be conditional to avoid total starvation of Classic by heavy L4S traffic. This raises the question of whether to sacrifice L4S throughput or L4S delay (or some other policy) to mitigate starvation of Classic:

Sacrifice L4S throughput: By using weighted round robin as the conditional priority scheduler, the L4S service can sacrifice some throughput during overload. This can either be thought of as guaranteeing a minimum throughput service for Classic traffic, or as guaranteeing a maximum delay for a packet at the head of the Classic queue.

The scheduling weight of the Classic queue should be small (e.g. 1/16). Then, in most traffic scenarios the scheduler will not interfere and it will not need to - the coupling mechanism and the end-systems will share out the capacity across both queues as if it were a single pool. However, because the congestion coupling only applies in one direction (from C to L), if L4S traffic is over-aggressive or unresponsive, the scheduler weight for Classic traffic will at least be large enough to ensure it does not starve.

In cases where the ratio of L4S to Classic flows (e.g. 19:1) is greater than the ratio of their scheduler weights (e.g. 15:1), the L4S flows will get less than an equal share of the capacity, but only slightly. For instance, with the example numbers given, each L4S flow will get (15/16)/19 = 4.9% when ideally each would get 1/20=5%. In the rather specific case of an unresponsive flow taking up just less than the capacity set aside for L4S (e.g. 14/16 in the above example), using WRR could significantly reduce the capacity left for any responsive L4S flows.

The scheduling weight of the Classic queue should not be too small, otherwise a C packet at the head of the queue could be excessively delayed by a continually busy L queue. For instance if the Classic weight is 1/16, the maximum that a Classic packet at the head of the queue can be delayed by L traffic is the serialization delay of 15 MTU-sized packets.

Sacrifice L4S Delay: To control milder overload of responsive traffic, particularly when close to the maximum congestion signal, the operator could choose to control overload of the Classic queue

by allowing some delay to 'leak' across to the L4S queue. The
scheduler can be made to behave like a single First-In First-Out
(FIFO) queue with different service times by implementing a very
simple conditional priority scheduler that could be called a
"time-shifted FIFO" (see the Modifier Earliest Deadline First
(MEDF) scheduler of [MEDF]). This scheduler adds tshift to the
queue delay of the next L4S packet, before comparing it with the
queue delay of the next Classic packet, then it selects the packet
with the greater adjusted queue delay. Under regular conditions,
this time-shifted FIFO scheduler behaves just like a strict
priority scheduler. But under moderate or high overload it
prevents starvation of the Classic queue, because the time-shift
(tshift) defines the maximum extra queuing delay of Classic
packets relative to L4S.

The example implementations in Appendix A and Appendix B could both
be implemented with either policy.

4.1.2. Congestion Signal Saturation: Introduce L4S Drop or Delay?

To keep the throughput of both L4S and Classic flows roughly equal
over the full load range, a different control strategy needs to be
defined above the point where one AQM first saturates to a
probability of 100% leaving no room to push back the load any harder.
If k>1, L4S will saturate first, even though saturation could be
caused by unresponsive traffic in either queue.

The term 'unresponsive' includes cases where a flow becomes
temporarily unresponsive, for instance, a real-time flow that takes a
while to adapt its rate in response to congestion, or a TCP-like flow
that is normally responsive, but above a certain congestion level it
will not be able to reduce its congestion window below the minimum of
2 segments [RFC5681], effectively becoming unresponsive. (Note that
L4S traffic ought to remain responsive below a window of 2 segments
(see [I-D.ietf-tsvwg-ecn-l4s-id]).

Saturation raises the question of whether to relieve congestion by
introducing some drop into the L4S queue or by allowing delay to grow
in both queues (which could eventually lead to tail drop too):

Drop on Saturation: Saturation can be avoided by setting a maximum
   threshold for L4S ECN marking (assuming k>1) before saturation
   starts to make the flow rates of the different traffic types
   diverge. Above that the drop probability of Classic traffic is
   applied to all packets of all traffic types. Then experiments
   have shown that queueing delay can be kept at the target in any
   overload situation, including with unresponsive traffic, and no
   further measures are required [DualQ-Test].

Delay on Saturation: When L4S marking saturates, instead of
   switching to drop, the drop and marking probabilities could be
   capped. Beyond that, delay will grow either solely in the queue
   with unresponsive traffic (if WRR is used), or in both queues (if
   time-shifted FIFO is used). In either case, the higher delay
   ought to control temporary high congestion. If the overload is
   more persistent, eventually the combined DualQ will overflow and
   tail drop will control congestion.

The example implementation in Appendix A solely applies the "drop on
saturation" policy.

4.1.3. Protecting against Unresponsive ECN-Capable Traffic

Unresponsive traffic has a greater advantage if it is also ECN-
capable. The advantage is undetectable at normal low levels of drop/
marking, but it becomes significant with the higher levels of drop/
marking typical during overload. This is an issue whether the ECN-
capable traffic is L4S or Classic.

This raises the question of whether and when to switch off ECN
marking and use solely drop instead, as required by both Section 7 of
 [RFC3168] and Section 4.2.1 of [RFC7567].

Experiments with the DualPI2 AQM (Appendix A) have shown that
introducing 'drop on saturation' at 100% L4S marking addresses this
problem with unresponsive ECN as well as addressing the saturation
problem. It leaves only a small range of congestion levels where
unresponsive traffic gains any advantage from using the ECN
capability, and the advantage is hardly detectable [DualQ-Test].

5. Acknowledgements

6. Contributors

The following contributed implementations and evaluations that
validated and helped to improve this specification:

Olga Albisser <olga@albisser.org> of Simula Research Lab, Norway
(Olga Bondarenko during early drafts) implemented the prototype
DualPI2 AQM for Linux with Koen De Schepper and conducted
extensive evaluations as well as implementing the live performance
visualization GUI [L4Sdemo16].

Olivier Tilmans <olivier.tilmans@nokia-bell-labs.com> of Nokia
Bell Labs, Belgium prepared and maintains the Linux implementation
of DualPI2 for upstreaming.

Tom Henderson <tomh@tomh.org> of CableLabs, US implemented various
Coupled DualQ AQMs for ns3, including DualPI2 and DualPIE over
point to point and DOCSIS 3.1 link models and conducted extensive
evaluations.

Ing Jyh (Inton) Tsang of Nokia, Belgium built the End-to-End Data
Centre to the Home broadband testbed on which Coupled DualQ
implementations were tested.

7. References

7.1. Normative References

[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>.

[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>.

[7.2](#). **Informative References**

[Alizadeh-stability]
          Alizadeh, M., Javanmard, A., and B. Prabhakar, "Analysis
          of DCTCP: Stability, Convergence, and Fairness", ACM
          SIGMETRICS 2011 , June 2011,
          <https://dl.acm.org/citation.cfm?id=1993753>.

[AQMmetrics]
          Kwon, M. and S. Fahmy, "A Comparison of Load-based and
          Queue- based Active Queue Management Algorithms", Proc.
          Int'l Soc. for Optical Engineering (SPIE) 4866:35--46 DOI:
          10.1117/12.473021, 2002,
          <https://www.cs.purdue.edu/homes/fahmy/papers/ldc.pdf>.

[ARED01] Floyd, S., Gummadi, R., and S. Shenker, "Adaptive RED: An
          Algorithm for Increasing the Robustness of RED's Active
          Queue Management", ACIRI Technical Report , August 2001,
          <http://www.icir.org/floyd/red.html>.

[BBRv1]   Cardwell, N., Cheng, Y., Hassas Yeganeh, S., and V.
          Jacobson, "BBR Congestion Control", Internet Draft draft-
          cardwell-iccrg-bbr-congestion-control-00, July 2017,
          <https://tools.ietf.org/html/
          draft-cardwell-iccrg-bbr-congestion-control-00>.

[CoDel]   Nichols, K. and V. Jacobson, "Controlling Queue Delay",
          ACM Queue 10(5), May 2012,
          <http://queue.acm.org/issuedetail.cfm?issue=2208917>.

[CRED_Insights]
          Briscoe, B., "Insights from Curvy RED (Random Early
          Detection)", BT Technical Report TR-TUB8-2015-003
          arXiv:1904.07339 [cs.NI], July 2015,
          <https://arxiv.org/abs/1904.07339>.

[DCttH15] De Schepper, K., Bondarenko, O., Briscoe, B., and I.
          Tsang, "`Data Centre to the Home': Ultra-Low Latency for
          All", RITE project Technical Report , 2015,
          <http://riteproject.eu/publications/>.

[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>.

   [DualPI2Linux]
              Albisser, O., De Schepper, K., Briscoe, B., Tilmans, O.,
              and H. Steen, "DUALPI2 - Low Latency, Low Loss and
              Scalable (L4S) AQM", Proc. Linux Netdev 0x13 , March 2019,
              <https://www.netdevconf.org/0x13/
              session.html?talk-DUALPI2-AQM>.

   [DualQ-Test]
              Steen, H., "Destruction Testing: Ultra-Low Delay using
              Dual Queue Coupled Active Queue Management", Masters
              Thesis, Dept of Informatics, Uni Oslo , May 2017.

   [I-D.briscoe-tsvwg-l4s-diffserv]
              Briscoe, B., "Interactions between Low Latency, Low Loss,
              Scalable Throughput (L4S) and Differentiated Services",
              draft-briscoe-tsvwg-l4s-diffserv-02 (work in progress),
              November 2018.

   [I-D.ietf-tsvwg-l4s-arch]
              Briscoe, B., Schepper, K., and M. Bagnulo, "Low Latency,
              Low Loss, Scalable Throughput (L4S) Internet Service:
              Architecture", draft-ietf-tsvwg-l4s-arch-03 (work in
              progress), October 2018.

   [L4Sdemo16]
              Bondarenko, O., De Schepper, K., Tsang, I., and B.
              Briscoe, "Ultra-Low Delay for All: Live Experience, Live
              Analysis", Proc. MMSYS'16 pp33:1--33:4, May 2016,
              <http://dl.acm.org/citation.cfm?doid=2910017.2910633
              (videos of demos: https://riteproject.eu/
              dctth/#1511dispatchwg)>.

   [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>.

   [Mathis09]
              Mathis, M., "Relentless Congestion Control", PFLDNeT'09 ,
              May 2009, <http://www.hpcc.jp/pfldnet2009/
              Program_files/1569198525.pdf>.

   [MEDF]   Menth, M., Schmid, M., Heiss, H., and T. Reim, "MEDF - a
              simple scheduling algorithm for two real-time transport
              service classes with application in the UTRAN", Proc. IEEE
              Conference on Computer Communications (INFOCOM'03) Vol.2
              pp.1116-1122, March 2003.

[PI2]   De Schepper, K., Bondarenko, O., Briscoe, B., and I.
            Tsang, "PI2: A Linearized AQM for both Classic and
            Scalable TCP", ACM CoNEXT'16 , December 2016,
            <https://riteproject.files.wordpress.com/2015/10/
            pi2_conext.pdf>.

[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>.

[RFC0970]   Nagle, J., "On Packet Switches With Infinite Storage",
            RFC 970, DOI 10.17487/RFC0970, December 1985,
            <https://www.rfc-editor.org/info/rfc970>.

[RFC2309]   Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering,
            S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G.,
            Partridge, C., Peterson, L., Ramakrishnan, K., Shenker,
            S., Wroclawski, J., and L. Zhang, "Recommendations on
            Queue Management and Congestion Avoidance in the
            Internet", RFC 2309, DOI 10.17487/RFC2309, April 1998,
            <https://www.rfc-editor.org/info/rfc2309>.

[RFC3246]   Davie, B., Charny, A., Bennet, J., Benson, K., Le Boudec,
            J., Courtney, W., Davari, S., Firoiu, V., and D.
            Stiliadis, "An Expedited Forwarding PHB (Per-Hop
            Behavior)", RFC 3246, DOI 10.17487/RFC3246, March 2002,
            <https://www.rfc-editor.org/info/rfc3246>.

[RFC3649]   Floyd, S., "HighSpeed TCP for Large Congestion Windows",
            RFC 3649, DOI 10.17487/RFC3649, December 2003,
            <https://www.rfc-editor.org/info/rfc3649>.

[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>.

[RFC5706]   Harrington, D., "Guidelines for Considering Operations and
            Management of New Protocols and Protocol Extensions",
            RFC 5706, DOI 10.17487/RFC5706, November 2009,
            <https://www.rfc-editor.org/info/rfc5706>.

   [RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF
              Recommendations Regarding Active Queue Management",
              BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015,
              <https://www.rfc-editor.org/info/rfc7567>.

   [RFC8033] Pan, R., Natarajan, P., Baker, F., and G. White,
              "Proportional Integral Controller Enhanced (PIE): A
              Lightweight Control Scheme to Address the Bufferbloat
              Problem", RFC 8033, DOI 10.17487/RFC8033, February 2017,
              <https://www.rfc-editor.org/info/rfc8033>.

   [RFC8034] White, G. and R. Pan, "Active Queue Management (AQM) Based
              on Proportional Integral Controller Enhanced PIE) for
              Data-Over-Cable Service Interface Specifications (DOCSIS)
              Cable Modems", RFC 8034, DOI 10.17487/RFC8034, February
              2017, <https://www.rfc-editor.org/info/rfc8034>.

   [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>.

   [RFC8290] Hoeiland-Joergensen, T., McKenney, P., Taht, D., Gettys,
              J., and E. Dumazet, "The Flow Queue CoDel Packet Scheduler
              and Active Queue Management Algorithm", RFC 8290,
              DOI 10.17487/RFC8290, January 2018,
              <https://www.rfc-editor.org/info/rfc8290>.

   [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>.

   [RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and
              R. Scheffenegger, "CUBIC for Fast Long-Distance Networks",
              RFC 8312, DOI 10.17487/RFC8312, February 2018,
              <https://www.rfc-editor.org/info/rfc8312>.

   [SigQ-Dyn]
              Briscoe, B., "Rapid Signalling of Queue Dynamics",
              Technical Report TR-BB-2017-001 arXiv:1904.07044 [cs.NI],
              September 2017, <https://arxiv.org/abs/1904.07044>.

Appendix A. Example DualQ Coupled PI2 Algorithm

   As a first concrete example, the pseudocode below gives the DualPI2
   algorithm. DualPI2 follows the structure of the DualQ Coupled AQM
   framework in Figure 1. A simple ramp function (configured in units
   of queuing time) with unsmoothed ECN marking is used for the Native

L4S AQM. The ramp can also be configured as a step function. The
PI2 algorithm [PI2] is used for the Classic AQM. PI2 is an improved
variant of the PIE AQM [RFC8033].

The pseudocode will be introduced in two passes. The first pass
explains the core concepts, deferring handling of overload to the
second pass. To aid comparison, line numbers are kept in step
between the two passes by using letter suffixes where the longer code
needs extra lines.

All variables are assumed to be floating point in their basic units
(size in bytes, time in seconds, rates in bytes/second, alpha and
beta in Hz, and probabilities from 0 to 1. Constants expressed in k,
M, G, u, m, %, ... are assumed to be converted to their appropriate
multiple or fraction. A real implementation that wants to use
integer values needs to handle appropriate scaling factors and allow
accordingly appropriate resolution of its integer types (including
temporary internal values during calculations).

A full open source implementation for Linux is available at:
https://github.com/L4STeam/sch_dualpi2_upstream and explained in
[DualPI2Linux]. The specification of the DualQ Coupled AQM for
DOCSIS cable modems and CMTSs is available in [DOCSIS3.1] and
explained in [LLD].

A.1. Pass #1: Core Concepts

The pseudocode manipulates three main structures of variables: the
packet (pkt), the L4S queue (lq) and the Classic queue (cq). The
pseudocode consists of the following six functions:

o the initialization function dualpi2_params_init(...) (Figure 2)
   that sets parameter defaults (the API for setting non-default
   values is omitted for brevity)

o the enqueue function dualpi2_enqueue(lq, cq, pkt) (Figure 3)

o the dequeue function dualpi2_dequeue(lq, cq, pkt) (Figure 4)

o recur(likelihood) for de-randomized ECN marking (shown at the end
   of Figure 4).

o the L4S AQM function laqm(qdelay) (Figure 5) used to calculate the
   ECN-marking probability for the L4S queue

o the base AQM function that implements the PI algorithm
   dualpi2_update(lq, cq) (Figure 6) used to regularly update the

base probability (p'), which is squared for the Classic AQM as
well as being coupled across to the L4S queue.

It also uses the following functions that are not shown in full here:

o scheduler(), which selects between the head packets of the two
   queues; the choice of scheduler technology is discussed later;

o cq.len() or lq.len() returns the current length (aka. backlog) of
   the relevant queue in bytes;

o cq.time() or lq.time() returns the current queuing delay (aka.
   sojourn time or service time) of the relevant queue in units of
   time (see Note a);

o mark(pkt) and drop(pkt) for ECN-marking and dropping a packet;

In experiments so far (building on experiments with PIE) on broadband
access links ranging from 4 Mb/s to 200 Mb/s with base RTTs from 5 ms
to 100 ms, DualPI2 achieves good results with the default parameters
in Figure 2. The parameters are categorised by whether they relate
to the Base PI2 AQM, the L4S AQM or the framework coupling them
together. Constants and variables derived from these parameters are
also included at the end of each category. Each parameter is
explained as it is encountered in the walk-through of the pseudocode
below.

```
1:  dualpi2_params_init(...) {          % Set input parameter defaults
2:    % DualQ Coupled framework parameters
5:    limit = MAX_LINK_RATE * 250 ms              % Dual buffer size
3:    k = 2                                        % Coupling factor
4:    % NOT SHOWN % scheduler-dependent weight or equival't parameter
6:
7:    % PI2 AQM parameters
8:    RTT_max = 100 ms                       % Worst case RTT expected
9:    RTT_typ = 15 ms                               % Typical RTT
11:   % PI2 constants derived from above PI2 parameters
10:   p_Cmax = min(1/k^2, 1)          % Max Classic drop/mark prob
12:   target = RTT_typ          % PI AQM Classic queue delay target
13:   Tupdate = min(RTT_typ, RTT_max/3)       % PI sampling interval
14:   alpha = 0.1 * Tupdate / RTT_max^2     % PI integral gain in Hz
15:   beta = 0.3 / RTT_max             % PI proportional gain in Hz
16:
17:   % L4S ramp AQM parameters
18:   minTh = 475 us        % L4S min marking threshold in time units
19:   range = 525 us              % Range of L4S ramp in time units
20:   Th_len = 2 * MTU         % Min L4S marking threshold in bytes
21:   % L4S constants incl. those derived from other parameters
22:   p_Lmax = 1                            % Max L4S marking prob
23:   floor = Th_len / MIN_LINK_RATE
24:   if (minTh < floor) {
25:     % Shift ramp so minTh >= serialization time of 2 MTU
26:     minTh = floor
27:   }
28:   maxTh = minTh+range   % L4S max marking threshold in time units
29: }
```

Figure 2: Example Header Pseudocode for DualQ Coupled PI2 AQM

The overall goal of the code is to maintain the base probability (p',
p-prime as in [Section 2.4](#)), which is an internal variable from which
the marking and dropping probabilities for L4S and Classic traffic
(p_L and p_C) are derived, with p_L in turn being derived from p_CL.
The probabilities p_CL and p_C are derived in lines 4 and 5 of the
dualpi2_update() function (Figure 6) then used in the
dualpi2_dequeue() function where p_L is also derived from p_CL at
line 6 (Figure 4). The code walk-through below builds up to
explaining that part of the code eventually, but it starts from
packet arrival.

```
1:  dualpi2_enqueue(lq, cq, pkt) { % Test limit and classify lq or cq
2:     if ( lq.len() + cq.len() + MTU > limit)
3:        drop(pkt)                       % drop packet if buffer is full
4:     timestamp(pkt)                     % attach arrival time to packet
5:     % Packet classifier
6:     if ( ecn(pkt) modulo 2 == 1 )         % ECN bits = ECT(1) or CE
7:        lq.enqueue(pkt)
8:     else                               % ECN bits = not-ECT or ECT(0)
9:        cq.enqueue(pkt)
10: }
```

Figure 3: Example Enqueue Pseudocode for DualQ Coupled PI2 AQM

```
1:  dualpi2_dequeue(lq, cq, pkt) {      % Couples L4S & Classic queues
2:     while ( lq.len() + cq.len() > 0 )
3:        if ( scheduler() == lq ) {
4:           lq.dequeue(pkt)                      % Scheduler chooses lq
5:           p'_L = laqm(lq.time())                      % Native L4S AQM
6:           p_L = max(p'_L, p_CL)              % Combining function
7:           if ( recur(p_L) )                      % Linear marking
8:              mark(pkt)
9:        } else {
10:          cq.dequeue(pkt)                       % Scheduler chooses cq
11:          if ( p_C > rand() ) {           % probability p_C = p'^2
12:             if ( ecn(pkt) == 0 ) {       % if ECN field = not-ECT
13:                drop(pkt)                              % squared drop
14:                continue      % continue to the top of the while loop
15:             }
16:             mark(pkt)                               % squared mark
17:          }
18:       }
19:       return(pkt)                          % return the packet and stop
20:    }
21:    return(NULL)                                 % no packet to dequeue
22: }

23: recur(likelihood) {      % Returns TRUE with a certain likelihood
24:    count += likelihood
25:    if (count > 1) {
26:       count -= 1
27:       return TRUE
28:    }
29:    return FALSE
30: }
```

Figure 4: Example Dequeue Pseudocode for DualQ Coupled PI2 AQM

When packets arrive, first a common queue limit is checked as shown
in line 2 of the enqueuing pseudocode in Figure 3. This assumes a
shared buffer for the two queues (Note b discusses the merits of
separate buffers). In order to avoid any bias against larger
packets, 1 MTU of space is always allowed and the limit is
deliberately tested before enqueue.

If limit is not exceeded, the packet is timestamped in line 4. This
assumes that queue delay is measured using the sojourn time technique
(see Note a for alternatives).

At lines 5-9, the packet is classified and enqueued to the Classic or
L4S queue dependent on the least significant bit of the ECN field in
the IP header (line 6). Packets with a codepoint having an LSB of 0
(Not-ECT and ECT(0)) will be enqueued in the Classic queue.
Otherwise, ECT(1) and CE packets will be enqueued in the L4S queue.
Optional additional packet classification flexibility is omitted for
brevity (see [I-D.ietf-tsvwg-ecn-l4s-id]).

The dequeue pseudocode (Figure 4) is repeatedly called whenever the
lower layer is ready to forward a packet. It schedules one packet
for dequeuing (or zero if the queue is empty) then returns control to
the caller, so that it does not block while that packet is being
forwarded. While making this dequeue decision, it also makes the
necessary AQM decisions on dropping or marking. The alternative of
applying the AQMs at enqueue would shift some processing from the
critical time when each packet is dequeued. However, it would also
add a whole queue of delay to the control signals, making the control
loop sloppier (for a typical RTT it would double the Classic queue's
feedback delay).

All the dequeue code is contained within a large while loop so that
if it decides to drop a packet, it will continue until it selects a
packet to schedule. Line 3 of the dequeue pseudocode is where the
scheduler chooses between the L4S queue (lq) and the Classic queue
(cq). Detailed implementation of the scheduler is not shown (see
discussion later).

o If an L4S packet is scheduled, in lines 7 and 8 the packet is ECN-
   marked with likelihood p_L. The recur() function at the end of
   Figure 4 is used, which is preferred over random marking because
   it avoids delay due to randomization when interpreting congestion
   signals, but it still desynchronizes the saw-teeth of the flows.
   Line 6 calculates p_L as the maximum of the coupled L4S
   probability p_CL and the probability from the native L4S AQM p'_L.
   This implements the max() function shown in Figure 1 to couple the
   outputs of the two AQMs together. Of the two probabilities input
   to p_L in line 6:

* p'_L is calculated per packet in line 5 by the laqm() function
  (see Figure 5),

* whereas p_CL is maintained by the dualpi2_update() function
  which runs every Tupdate (Tupdate is set in line 13 of
  Figure 2. It defaults to 16 ms in the reference Linux
  implementation because it has to be rounded to a multiple of 4
  ms).

o If a Classic packet is scheduled, lines 10 to 17 drop or mark the
   packet with probability p_C.

The Native L4S AQM algorithm (Figure 5) is a ramp function, similar
to the RED algorithm, but simplified as follows:

o The extent of the ramp is defined in units of queuing delay, not
   bytes, so that configuration remains invariant as the queue
   departure rate varies.

o It uses instantaneous queueing delay, which avoids the complexity
   of smoothing, but also avoids embedding a worst-case RTT of
   smoothing delay in the network (see Section 2.1).

o The ramp rises linearly directly from 0 to 1, not to a an
   intermediate value of p'_L as RED would, because there is no need
   to keep ECN marking probability low.

o Marking does not have to be randomized. Determinism is used
   instead of randomness; to reduce the delay necessary to smooth out
   the noise of randomness from the signal.

The ramp function requires two configuration parameters, the minimum
threshold (minTh) and the width of the ramp (range), both in units of
queuing time), as shown in lines 18 & 19 of the initialization
function in Figure 2. The ramp function can be configured as a step
(see Note c).

Although the DCTCP paper [Alizadeh-stability] recommends an ECN
marking threshold of 0.17*RTT_typ, it also shows that the threshold
can be much shallower with hardly any worse under-utilization of the
link (because the amplitude of DCTCP's sawteeth is so small). Based
on extensive experiments, for the public Internet a default minimum
ECN marking threshold of about RTT_typ/30 is recommended.

A minimum marking threshold parameter (Th_len) in transmission units
(default 2 MTU) is also necessary to ensure that the ramp does not
trigger excessive marking on slow links. The code in lines 24-27 of
the initialization function (Figure 2) converts 2 MTU into time units

and shifts the ramp so that the min threshold is no shallower than
this floor.

```
1:  laqm(qdelay) {               % Returns native L4S AQM probability
2:    if (qdelay >= maxTh)
3:       return 1
4:    else if (qdelay > minTh)
5:       return (qdelay - minTh)/range  % Divide could use a bit-shift
6:    else
7:       return 0
8:  }
```

Figure 5: Example Pseudocode for the Native L4S AQM

```
1:  dualpi2_update(lq, cq) {                % Update p' every Tupdate
2:    curq = cq.time()  % use queuing time of first-in Classic packet
3:    p' = p' + alpha * (curq - target) + beta * (curq - prevq)
4:    p_CL = k * p'  % Coupled L4S prob = base prob * coupling factor
5:    p_C = p'^2                    % Classic prob = (base prob)^2
6:    prevq = curq
7:  }
```

Figure 6: Example PI-Update Pseudocode for DualQ Coupled PI2 AQM

The coupled marking probability, p_CL depends on the base probability
(p'), which is kept up to date by the core PI algorithm in Figure 6
executed every Tupdate.

Note that p' solely depends on the queuing time in the Classic queue.
In line 2, the current queuing delay (curq) is evaluated from how
long the head packet was in the Classic queue (cq). The function
cq.time() (not shown) subtracts the time stamped at enqueue from the
current time (see Note a) and implicitly takes the current queuing
delay as 0 if the queue is empty.

The algorithm centres on line 3, which is a classical Proportional-
Integral (PI) controller that alters p' dependent on: a) the error
between the current queuing delay (curq) and the target queuing delay
('target' - see [RFC8033]); and b) the change in queuing delay since
the last sample. The name 'PI' represents the fact that the second
factor (how fast the queue is growing) is _P_roportional to load
while the first is the _I_ntegral of the load (so it removes any
standing queue in excess of the target).

The two 'gain factors' in line 3, alpha and beta, respectively weight
how strongly each of these elements ((a) and (b)) alters p'. They
are in units of 'per second of delay' or Hz, because they transform

differences in queueing delay into changes in probability (assuming
probability has a value from 0 to 1).

alpha and beta determine how much p' ought to change after each
update interval (Tupdate). For smaller Tupdate, p' should change by
the same amount per second, but in finer more frequent steps. So
alpha depends on Tupdate (see line 14 of the initialization function
in Figure 2). It is best to update p' as frequently as possible, but
Tupdate will probably be constrained by hardware performance. As
shown in line 13, the update interval should be at least as frequent
as once per the RTT of a typical flow (RTT_typ) as long as it does
not exceed roughly RTT_max/3. For link rates from 4 - 200 Mb/s, a
target RTT of 15ms and a maximum RTT of 100ms, it has been verified
through extensive testing that Tupdate=16ms (as recommended in
[RFC8033]) is sufficient.

The choice of alpha and beta also determines the AQM's stable
operating range. The AQM ought to change p' as fast as possible in
response to changes in load without over-compensating and therefore
causing oscillations in the queue. Therefore, the values of alpha
and beta also depend on the RTT of the expected worst-case flow
(RTT_max).

Recommended derivations of the gain constants alpha and beta can be
approximated for Reno over a PI2 AQM as: alpha = 0.1 * Tupdate /
RTT_max^2; beta = 0.3 / RTT_max, as shown in lines 14 & 15 of
Figure 2. These are derived from the stability analysis in [PI2].
For the default values of Tupdate=16 ms and RTT_max = 100 ms, they
result in alpha = 0.16; beta = 3.2 (discrepancies are due to
rounding). These defaults have been verified with a wide range of
link rates, target delays and a range of traffic models with mixed
and similar RTTs, short and long flows, etc.

In corner cases, p' can overflow the range [0,1] so the resulting
value of p' has to be bounded (omitted from the pseudocode). Then,
as already explained, the coupled and Classic probabilities are
derived from the new p' in lines 4 and 5 of Figure 6 as p_CL = k*p'
and p_C = p'^2.

Because the coupled L4S marking probability (p_CL) is factored up by
k, the dynamic gain parameters alpha and beta are also inherently
factored up by k for the L4S queue. So, the effective gain factor
for the L4S queue is k*alpha (with defaults alpha = 0.16 Hz and k=2,
effective L4S alpha = 0.32 Hz).

Unlike in PIE [RFC8033], alpha and beta do not need to be tuned every
Tupdate dependent on p'. Instead, in PI2, alpha and beta are
independent of p' because the squaring applied to Classic traffic

tunes them inherently. This is explained in [PI2], which also
explains why this more principled approach removes the need for most
of the heuristics that had to be added to PIE.

Notes:

a. The drain rate of the queue can vary if it is scheduled relative
    to other queues, or to cater for fluctuations in a wireless
    medium. To auto-adjust to changes in drain rate, the queue must
    be measured in time, not bytes or packets [AQMmetrics] [CoDel].
    Queuing delay could be measured directly by storing a per-packet
    time-stamp as each packet is enqueued, and subtracting this from
    the system time when the packet is dequeued. If time-stamping is
    not easy to introduce with certain hardware, queuing delay could
    be predicted indirectly by dividing the size of the queue by the
    predicted departure rate, which might be known precisely for some
    link technologies (see for example [RFC8034]).

b. Line 2 of the dualpi2_enqueue() function (Figure 3) assumes an
    implementation where lq and cq share common buffer memory. An
    alternative implementation could use separate buffers for each
    queue, in which case the arriving packet would have to be
    classified first to determine which buffer to check for available
    space. The choice is a trade off; a shared buffer can use less
    memory whereas separate buffers isolate the L4S queue from tail-
    drop due to large bursts of Classic traffic (e.g. a Classic TCP
    during slow-start over a long RTT).

c. There has been some concern that using the step function of DCTCP
    for the Native L4S AQM requires end-systems to smooth the signal
    for an unnecessarily large number of round trips to ensure
    sufficient fidelity. A ramp is no worse than a step in initial
    experiments with existing DCTCP. Therefore, it is recommended
    that a ramp is configured in place of a step, which will allow
    congestion control algorithms to investigate faster smoothing
    algorithms.

    A ramp is more general that a step, because an operator can
    effectively turn the ramp into a step function, as used by DCTCP,
    by setting the range to zero. There will not be a divide by zero
    problem at line 4 of Figure 5 because, if minTh is equal to
    maxTh, the condition for this ramp calculation cannot arise.

A.2. Pass #2: Overload Details

Figure 7 repeats the dequeue function of Figure 4, but with overload
details added. Similarly Figure 8 repeats the core PI algorithm of

Figure 6 with overload details added. The initialization, enqueue,
L4S AQM and recur functions are unchanged.

In line 10 of the initialization function (Figure 2), the maximum
Classic drop probability p_Cmax = min(1/k^2, 1) or 1/4 for the
default coupling factor k=2. p_Cmax is the point at which it is
deemed that the Classic queue has become persistently overloaded, so
it switches to using drop, even for ECN-capable packets. ECT packets
that are not dropped can still be ECN-marked.

In practice, 25% has been found to be a good threshold to preserve
fairness between ECN capable and non ECN capable traffic. This
protects the queues against both temporary overload from responsive
flows and more persistent overload from any unresponsive traffic that
falsely claims to be responsive to ECN.

When the Classic ECN marking probability reaches the p_Cmax threshold
(1/k^2), the marking probability coupled to the L4S queue, p_CL will
always be 100% for any k (by equation (1) in Section 2). So, for
readability, the constant p_Lmax is defined as 1 in line 22 of the
initialization function Figure 2. This is intended to ensure that
the L4S queue starts to introduce dropping once ECN-marking saturates
at 100% and can rise no further. The 'Prague L4S' requirements
[I-D.ietf-tsvwg-ecn-l4s-id] state that, when an L4S congestion
control detects a drop, it falls back to a response that coexists
with 'Classic' TCP. So it is correct that, when the L4S queue drops
packets, it drops them proportional to p'^2, as if they are Classic
packets.

Both these switch-overs are triggered by the tests for overload
introduced in lines 4b and 12b of the dequeue function (Figure 7).
Lines 8c to 8g drop L4S packets with probability p'^2. Lines 8h to
8i mark the remaining packets with probability p_CL. Given p_Lmax =
1, all remaining packets will be marked because, to have reached the
else block at line 8b, p_CL >= 1.

Lines 2c to 2d in the core PI algorithm (Figure 8) deal with overload
of the L4S queue when there is no Classic traffic. This is
necessary, because the core PI algorithm maintains the appropriate
drop probability to regulate overload, but it depends on the length
of the Classic queue. If there is no Classic queue the naive PI
update function in Figure 6 would drop nothing, even if the L4S queue
were overloaded - so tail drop would have to take over (lines 2 and 3
of Figure 3).

Instead, the test at line 2a of the full PI update function in
Figure 8 keeps delay on target using drop. If the test at line 2a of
finds that the Classic queue is empty, line 2d measures the current

queue delay using the L4S queue instead. While the L4S queue is not
overloaded, its delay will always be tiny compared to the target
Classic queue delay. So p_CL will be driven to zero, and the L4S
queue will naturally be governed solely by p'_L from the native L4S
AQM (lines 5 and 6 of the dequeue algorithm in Figure 7). But, if
unresponsive L4S source(s) cause overload, the DualQ transitions
smoothly to L4S marking based on the PI algorithm. If overload
increases further, it naturally transitions from marking to dropping
by the switch-over mechanism already described.

```
1:  dualpi2_dequeue(lq, cq, pkt) {     % Couples L4S & Classic queues
2:    while ( lq.len() + cq.len() > 0 ) {
3:      if ( scheduler() == lq ) {
4a:       lq.dequeue(pkt)                            % L4S scheduled
4b:       if ( p_CL < p_Lmax ) {     % Check for overload saturation
5:          p'_L = laqm(lq.time())                   % Native L4S AQM
6:          p_L = max(p'_L, p_CL)            % Combining function
7:          if ( recur(p_L) )                      % Linear marking
8a:           mark(pkt)
8b:       } else {                           % overload saturation
8c:         if ( p_C > rand() ) {          % probability p_C = p'^2
8e:           drop(pkt)     % revert to Classic drop due to overload
8f:           continue        % continue to the top of the while loop
8g:         }
8h:         if ( p_CL > rand() )         % probability p_CL = k * p'
8i:           mark(pkt)        % linear marking of remaining packets
8j:       }
9:      } else {
10:       cq.dequeue(pkt)                        % Classic scheduled
11:       if ( p_C > rand() ) {          % probability p_C = p'^2
12a:        if ( (ecn(pkt) == 0)              % ECN field = not-ECT
12b:            OR (p_C >= p_Cmax) ) {     % Overload disables ECN
13:           drop(pkt)                     % squared drop, redo loop
14:           continue        % continue to the top of the while loop
15:         }
16:         mark(pkt)                              % squared mark
17:       }
18:     }
19:     return(pkt)                     % return the packet and stop
20:   }
21:   return(NULL)                             % no packet to dequeue
22: }
```

           Figure 7: Example Dequeue Pseudocode for DualQ Coupled PI2 AQM
                          (Including Overload Code)

```
1:  dualpi2_update(lq, cq) {                % Update p' every Tupdate
2a:   if ( cq.len() > 0 )
2b:     curq = cq.time() %use queuing time of first-in Classic packet
2c:   else                                    % Classic queue empty
2d:     curq = lq.time()    % use queuing time of first-in L4S packet
3:    p' = p' + alpha * (curq - target) + beta * (curq - prevq)
4:    p_CL = p' * k  % Coupled L4S prob = base prob * coupling factor
5:    p_C = p'^2                      % Classic prob = (base prob)^2
6:    prevq = curq
7:  }
```

Figure 8: Example PI-Update Pseudocode for DualQ Coupled PI2 AQM
                        (Including Overload Code)

The choice of scheduler technology is critical to overload protection
(see Section 4.1).

o A well-understood weighted scheduler such as weighted round robin
   (WRR) is recommended. As long as the scheduler weight for Classic
   is small (e.g. 1/16), its exact value is unimportant because it
   does not normally determine capacity shares. The weight is only
   important to prevent unresponsive L4S traffic starving Classic
   traffic. This is because capacity sharing between the queues is
   normally determined by the coupled congestion signal, which
   overrides the scheduler, by making L4S sources leave roughly equal
   per-flow capacity available for Classic flows.

o Alternatively, a time-shifted FIFO (TS-FIFO) could be used. It
   works by selecting the head packet that has waited the longest,
   biased against the Classic traffic by a time-shift of tshift. To
   implement time-shifted FIFO, the scheduler() function in line 3 of
   the dequeue code would simply be implemented as the scheduler()
   function at the bottom of Figure 10 in Appendix B. For the public
   Internet a good value for tshift is 50ms. For private networks
   with smaller diameter, about 4*target would be reasonable. TS-
   FIFO is a very simple scheduler, but complexity might need to be
   added to address some deficiencies (which is why it is not
   recommended over WRR):

   * TS-FIFO does not fully isolate latency in the L4S queue from
      uncontrolled bursts in the Classic queue;

   * TS-FIFO is only appropriate if time-stamping of packets is
      feasible;

   * Even if time-stamping is supported, the sojourn time of the
      head packet is always stale. For instance, if a burst arrives
      at an empty queue, the sojourn time will only measure the delay

of the burst once the burst is over, even though the queue knew
about it from the start. At the cost of more operations and
more storage, a 'scaled sojourn time' metric of queue delay can
be used, which is the sojourn time of a packet scaled by the
ratio of the queue sizes when the packet departed and arrived
[SigQ-Dyn].

o A strict priority scheduler would be inappropriate, because it
would starve Classic if L4S was overloaded.

Appendix B. Example DualQ Coupled Curvy RED Algorithm

As another example of a DualQ Coupled AQM algorithm, the pseudocode
below gives the Curvy RED based algorithm. Although the AQM was
designed to be efficient in integer arithmetic, to aid understanding
it is first given using floating point arithmetic (Figure 10). Then,
one possible optimization for integer arithmetic is given, also in
pseudocode (Figure 11). To aid comparison, the line numbers are kept
in step between the two by using letter suffixes where the longer
code needs extra lines.

B.1. Curvy RED in Pseudocode

The pseudocode manipulates three main structures of variables: the
packet (pkt), the L4S queue (lq) and the Classic queue (cq) and
consists of the following five functions:

o the initialization function cred_params_init(...) (Figure 2) that
sets parameter defaults (the API for setting non-default values is
omitted for brevity);

o the dequeue function cred_dequeue(lq, cq, pkt) (Figure 4);

o the scheduling function scheduler(), which selects between the
head packets of the two queues.

It also uses the following functions that are either shown elsewhere,
or not shown in full here:

o the enqueue function, which is identical to that used for DualPI2,
dualpi2_enqueue(lq, cq, pkt) in Figure 3;

o mark(pkt) and drop(pkt) for ECN-marking and dropping a packet;

o cq.len() or lq.len() returns the current length (aka. backlog) of
the relevant queue in bytes;

o cq.time() or lq.time() returns the current queuing delay (aka.
   sojourn time or service time) of the relevant queue in units of
   time (see Note a in Appendix A.1).

Because Curvy RED was evaluated before DualPI2, certain improvements
introduced for DualPI2 were not evaluated for Curvy RED. In the
pseudocode below, the straightforward improvements have been added on
the assumption they will provide similar benefits, but that has not
been proven experimentally. They are: i) a conditional priority
scheduler instead of strict priority ii) a time-based threshold for
the native L4S AQM; iii) ECN support for the Classic AQM. A recent
evaluation has proved that a minimum ECN-marking threshold (minTh)
greatly improves performance, so this is also included in the
pseudocode.

Overload protection has not been added to the Curvy RED pseudocode
below so as not to detract from the main features. It would be added
in exactly the same way as in Appendix A.2 for the DualPI2
pseudocode. The native L4S AQM uses a step threshold, but a ramp
like that described for DualPI2 could be used instead. The scheduler
uses the simple TS-FIFO algorithm, but it could be replaced with WRR.

The Curvy RED algorithm has not been maintained or evaluated to the
same degree as the DualPI2 algorithm. In initial experiments on
broadband access links ranging from 4 Mb/s to 200 Mb/s with base RTTs
from 5 ms to 100 ms, Curvy RED achieved good results with the default
parameters in Figure 9.

The parameters are categorised by whether they relate to the Classic
AQM, the L4S AQM or the framework coupling them together. Constants
and variables derived from these parameters are also included at the
end of each category. These are the raw input parameters for the
algorithm. A configuration front-end could accept more meaningful
parameters (e.g. RTT_max and RTT_typ) and convert them into these
raw parameters, as has been done for DualPI2 in Appendix A. Where
necessary, parameters are explained further in the walk-through of
the pseudocode below.

```
1:  cred_params_init(...) {              % Set input parameter defaults
2:    % DualQ Coupled framework parameters
3:    limit = MAX_LINK_RATE * 250 ms                 % Dual buffer size
4:    k' = 1                       % Coupling factor as a power of 2
5:    tshift = 50 ms               % Time shift of TS-FIFO scheduler
6:    % Constants derived from Classic AQM parameters
7:    k = 2^k'                     % Coupling factor from Equation (1)
6:
7:    % Classic AQM parameters
8:    g_C = 5            % EWMA smoothing parameter as a power of 1/2
9:    S_C = -1          % Classic ramp scaling factor as a power of 2
10:   minTh = 500 ms    % No Classic drop/mark below this queue delay
11:   % Constants derived from Classic AQM parameters
12:   gamma = 2^(-g_C)                     % EWMA smoothing parameter
13:   range_C = 2^S_C                        % Range of Classic ramp
14:
15:   % L4S AQM parameters
16:   T = 1 ms            % Queue delay threshold for native L4S AQM
17:   % Constants derived from above parameters
18:   S_L = S_C - k'       % L4S ramp scaling factor as a power of 2
19:   range_L = 2^S_L                           % Range of L4S ramp
20: }
```

 Figure 9: Example Header Pseudocode for DualQ Coupled Curvy RED AQM

```
1:  cred_dequeue(lq, cq, pkt) {        % Couples L4S & Classic queues
2:    while ( lq.len() + cq.len() > 0 ) {
3:      if ( scheduler() == lq ) {
4:        lq.dequeue(pkt)                              % L4S scheduled
5a:       p_CL = (cq.time() - minTh) / range_L
5b:       if (  ( lq.time() > T )
5c:           OR ( p_CL > maxrand(U) ) )
6:          mark(pkt)
7:      } else {
8:        cq.dequeue(pkt)                       % Classic scheduled
9a:       Q_C = gamma * qc.time() + (1-gamma) * Q_C % Classic Q EWMA
10a:      sqrt_p_C = (Q_C - minTh) / range_C
10b:      if ( sqrt_p_C > maxrand(2*U) ) {
11:         if ( (ecn(pkt) == 0)  {          % ECN field = not-ECT
12:           drop(pkt)                     % Squared drop, redo loop
13:           continue      % continue to the top of the while loop
14:         }
15:         mark(pkt)
16:       }
17:     }
18:     return(pkt)                % return the packet and stop here
19:   }
20:   return(NULL)                            % no packet to dequeue
21: }

22: maxrand(u) {                  % return the max of u random numbers
23:   maxr=0
24:   while (u-- > 0)
25:     maxr = max(maxr, rand())                    % 0 <= rand() < 1
26:   return(maxr)
27: }

28: scheduler() {
29:   if ( lq.time() + tshift >= cq.time() )
30:     return lq;
31:   else
32:     return cq;
33: }
```

Figure 10: Example Dequeue Pseudocode for DualQ Coupled Curvy RED AQM

The dequeue pseudocode (Figure 10) is repeatedly called whenever the
lower layer is ready to forward a packet. It schedules one packet
for dequeuing (or zero if the queue is empty) then returns control to
the caller, so that it does not block while that packet is being
forwarded. While making this dequeue decision, it also makes the
necessary AQM decisions on dropping or marking. The alternative of
applying the AQMs at enqueue would shift some processing from the

critical time when each packet is dequeued. However, it would also add a whole queue of delay to the control signals, making the control loop very sloppy.

The code is written assuming the AQMs are applied on dequeue (Note 1). All the dequeue code is contained within a large while loop so that if it decides to drop a packet, it will continue until it selects a packet to schedule. If both queues are empty, the routine returns NULL at line 20. Line 3 of the dequeue pseudocode is where the conditional priority scheduler chooses between the L4S queue (lq) and the Classic queue (cq). The time-shifted FIFO scheduler is shown at lines 28-33, which would be suitable if simplicity is paramount (see Note 2).

Within each queue, the decision whether to forward, drop or mark is taken as follows (to simplify the explanation, it is assumed that U=1):

L4S: If the test at line 3 determines there is an L4S packet to
   dequeue, the tests at lines 5b and 5c determine whether to mark
   it. The first is a simple test of whether the L4S queue delay
   (lq.time()) is greater than a step threshold T (Note 3). The
   second test is similar to the random ECN marking in RED, but with
   the following differences: ii) marking depends on queuing time,
   not bytes, in order to scale for any link rate without being
   reconfigured; ii) marking of the L4S queue does not depend on
   itself, it depends on the queuing time of the _other_ (Classic)
   queue, where cq.time() is the queuing time of the packet at the
   head of the Classic queue (zero if empty); iii) marking depends on
   the instantaneous queuing time (of the other Classic queue), not a
   smoothed average; iv) the queue is compared with the maximum of U
   random numbers (but if U=1, this is the same as the single random
   number used in RED).

   Specifically, in line 5a the coupled marking probability p_CL is
   set to the excess of the Classic queueing delay qc.time() above
   the minimum queuing delay threshold (minTh) all divided by the L4S
   scaling parameter range_L. range_L represents the queuing delay
   (in seconds) added to minTh at which marking probability would hit
   100%. Then in line 5c (if U=1) the result is compared with a
   uniformly distributed random number between 0 and 1, which ensures
   that marking probability will linearly increase with queueing
   time.

Classic: If the scheduler at line 3 chooses to dequeue a Classic
   packet and jumps to line 7, the test at line 10b determines
   whether to drop or mark it. But before that, line 9a updates Q_C,
   which is an exponentially weighted moving average (Note 4) of the

queuing time in the Classic queue, where qc.time() is the current
instantaneous queueing time of the Classic queue and gamma is the
EWMA constant (default 1/32, see line 12 of the initialization
function).

Lines 10a and 10b implement the Classic AQM. In line 10a the
averaged queuing time Q_C is divided by the Classic scaling
parameter range_C, in the same way that queuing time was scaled
for L4S marking. This scaled queuing time will be squared to
compute Classic drop probability so, before it is squared, it is
effectively the square root of the drop probability, hence it is
given the variable name sqrt_p_C. The squaring is done by
comparing it with the maximum out of two random numbers (assuming
U=1). Comparing it with the maximum out of two is the same as the
logical `AND' of two tests, which ensures drop probability rises
with the square of queuing time.

The AQM functions in each queue (lines 5c & 10b) are two cases of a
new generalization of RED called Curvy RED, motivated as follows.
When the performance of this AQM was compared with fq_CoDel and PIE,
their goal of holding queuing delay to a fixed target seemed
misguided [CRED_Insights]. As the number of flows increases, if the
AQM does not allow TCP to increase queuing delay, it has to introduce
abnormally high levels of loss. Then loss rather than queuing
becomes the dominant cause of delay for short flows, due to timeouts
and tail losses.

Curvy RED constrains delay with a softened target that allows some
increase in delay as load increases. This is achieved by increasing
drop probability on a convex curve relative to queue growth (the
square curve in the Classic queue, if U=1). Like RED, the curve hugs
the zero axis while the queue is shallow. Then, as load increases,
it introduces a growing barrier to higher delay. But, unlike RED, it
requires only two parameters, not three. The disadvantage of Curvy
RED is that it is not adapted to a wide range of RTTs. Curvy RED can
be used as is when the RTT range to be supported is limited,
otherwise an adaptation mechanism is required.

From our limited experiments with Curvy RED so far, recommended
values of these parameters are: S_C = -1; g_C = 5; T = 5 * MTU at the
link rate (about 1ms at 60Mb/s) for the range of base RTTs typical on
the public Internet. [CRED_Insights] explains why these parameters
are applicable whatever rate link this AQM implementation is deployed
on and how the parameters would need to be adjusted for a scenario
with a different range of RTTs (e.g. a data centre). The setting of
k depends on policy (see Section 2.5 and Appendix C respectively for
its recommended setting and guidance on alternatives).

There is also a cUrviness parameter, U, which is a small positive
integer. It is likely to take the same hard-coded value for all
implementations, once experiments have determined a good value. Only
U=1 has been used in experiments so far, but results might be even
better with U=2 or higher.

Notes:

1. The alternative of applying the AQMs at enqueue would shift some
    processing from the critical time when each packet is dequeued.
    However, it would also add a whole queue of delay to the control
    signals, making the control loop sloppier (for a typical RTT it
    would double the Classic queue's feedback delay). On a platform
    where packet timestamping is feasible, e.g. Linux, it is also
    easiest to apply the AQMs at dequeue because that is where
    queuing time is also measured.

2. WRR better isolates the L4S queue from large delay bursts in the
    Classic queue, but it is slightly less simple than TS-FIFO. If
    WRR were used, a low default Classic weight (e.g. 1/16) would
    need to be configured in place of the time shift in line 5 of the
    initialization function (Figure 9).

3. A step function is shown for simplicity. A ramp function (see
    Figure 5 and the discussion around it in Appendix A.1) is
    recommended, because it is more general than a step and has the
    potential to enable L4S congestion controls to converge more
    rapidly.

4. An EWMA is only one possible way to filter bursts; other more
    adaptive smoothing methods could be valid and it might be
    appropriate to decrease the EWMA faster than it increases, e.g.
    by using the minimum of the smoothed and instantaneous queue
    delays, min(Q_C, qc.time()).

B.2. Efficient Implementation of Curvy RED

Although code optimization depends on the platform, the following
notes explain where the design of Curvy RED was particularly
motivated by efficient implementation.

The Classic AQM at line 10b calls maxrand(2*U), which gives twice as
much curviness as the call to maxrand(U) in the marking function at
line 5c. This is the trick that implements the square rule in
equation (1) (Section 2.1). This is based on the fact that, given a
number X from 1 to 6, the probability that two dice throws will both
be less than X is the square of the probability that one throw will
be less than X. So, when U=1, the L4S marking function is linear and

the Classic dropping function is squared. If U=2, L4S would be a
square function and Classic would be quartic. And so on.

The maxrand(u) function in lines 16-21 simply generates u random
numbers and returns the maximum. Typically, maxrand(u) could be run
in parallel out of band. For instance, if U=1, the Classic queue
would require the maximum of two random numbers. So, instead of
calling maxrand(2*U) in-band, the maximum of every pair of values
from a pseudorandom number generator could be generated out-of-band,
and held in a buffer ready for the Classic queue to consume.

```
1:  cred_dequeue(lq, cq, pkt) {       % Couples L4S & Classic queues
2:     while ( lq.len() + cq.len() > 0 ) {
3:       if ( scheduler() == lq ) {
4:         lq.dequeue(pkt)                          % L4S scheduled
5:         if ((lq.time() > T) OR (cq.ns() >> (S_L-2) > maxrand(U)))
6:           mark(pkt)
7:       } else {
8:         cq.dequeue(pkt)                       % Classic scheduled
9:         Q_C += (cq.ns() - Q_C) >> g_C          % Classic Q EWMA
10:        if ( (Q_C >> (S_C-2) ) > maxrand(2*U) ) {
11:          if ( (ecn(pkt) == 0)  {         % ECN field = not-ECT
12:            drop(pkt)                     % Squared drop, redo loop
13:            continue      % continue to the top of the while loop
14:          }
15:          mark(pkt)
16:        }
17:      }
18:      return(pkt)                % return the packet and stop here
19:    }
20:    return(NULL)                          % no packet to dequeue
21: }
```

    Figure 11: Optimised Example Dequeue Pseudocode for Coupled DualQ AQM
                        using Integer Arithmetic

The two ranges, range_L and range_C are expressed as powers of 2 so
that division can be implemented as a right bit-shift (>>) in lines 5
and 10 of the integer variant of the pseudocode (Figure 11).

For the integer variant of the pseudocode, an integer version of the
rand() function used at line 25 of the maxrand(function) in Figure 10
would be arranged to return an integer in the range 0 <= maxrand() <
2^32 (not shown). This would scale up all the floating point
probabilities in the range [0,1] by 2^32.

Queuing delays are also scaled up by 2^32, but in two stages: i) In
lines 5 and 10 queuing times cq.ns() and pkt.ns() are returned in

integer nanoseconds, making the values about 2^30 times larger than
when the units were seconds, ii) then in lines 3 and 9 an adjustment
of -2 to the right bit-shift multiplies the result by 2^2, to
complete the scaling by 2^32.

In line 8 of the initialization function, the EWMA constant gamma is
represented as an integer power of 2, g_C, so that in line 9 of the
integer code the division needed to weight the moving average can be
implemented by a right bit-shift (>> g_C).

Appendix C. Guidance on Controlling Throughput Equivalence

```
+---------------+------+-------+
| RTT_C / RTT_L | Reno | Cubic |
+---------------+------+-------+
|             1 | k'=1 | k'=0  |
|             2 | k'=2 | k'=1  |
|             3 | k'=2 | k'=2  |
|             4 | k'=3 | k'=2  |
|             5 | k'=3 | k'=3  |
+---------------+------+-------+
```

Table 1: Value of k' for which DCTCP throughput is roughly the same
as Reno or Cubic, for some example RTT ratios

k' is related to k in Equation (1) (Section 2.1) by k=2^k'.

To determine the appropriate policy, the operator first has to judge
whether it wants DCTCP flows to have roughly equal throughput with
Reno or with Cubic (because, even in its Reno-compatibility mode,
Cubic is about 1.4 times more aggressive than Reno). Then the
operator needs to decide at what ratio of RTTs it wants DCTCP and
Classic flows to have roughly equal throughput. For example choosing
k'=0 (equivalent to k=1) will make DCTCP throughput roughly the same
as Cubic, _if their RTTs are the same_.

However, even if the base RTTs are the same, the actual RTTs are
unlikely to be the same, because Classic (Cubic or Reno) traffic
needs a large queue to avoid under-utilization and excess drop,
whereas L4S (DCTCP) does not. The operator might still choose this
policy if it judges that DCTCP throughput should be rewarded for
keeping its own queue short.

On the other hand, the operator will choose one of the higher values
for k', if it wants to slow DCTCP down to roughly the same throughput
as Classic flows, to compensate for Classic flows slowing themselves
down by causing themselves extra queuing delay.

The values for k' in the table are derived from the formulae, which
was developed in [DCttH15]:

    2^k' = 1.64 (RTT_reno / RTT_dc)   (2)
    2^k' = 1.19 (RTT_cubic / RTT_dc)   (3)

For localized traffic from a particular ISP's data centre, using the
measured RTTs, it was calculated that a value of k'=3 (equivalant to
k=8) would achieve throughput equivalence, and experiments verified
the formula very closely.

For a typical mix of RTTs from local data centres and across the
general Internet, a value of k'=1 (equivalent to k=2) is recommended
as a good workable compromise.

Authors' Addresses

Koen De Schepper
Nokia Bell Labs
Antwerp
Belgium

Email: koen.de_schepper@nokia.com
URI: https://www.bell-labs.com/usr/koen.de_schepper

Bob Briscoe (editor)
CableLabs
UK

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

Greg White
CableLabs
Louisville, CO
US

Email: G.White@CableLabs.com