DualQ Coupled AQMs for Low Latency, Low Loss and Scalable Throughput (L4S)
draft-ietf-tsvwg-aqm-dualq-coupled-05

Abstract

   Data Centre TCP (DCTCP) was designed to provide predictably low
   queuing latency, near-zero loss, and throughput scalability using
   explicit congestion notification (ECN) and an extremely simple
   marking behaviour on switches. However, DCTCP does not co-exist with
   existing TCP traffic---DCTCP is so aggressive that existing TCP
   algorithms approach starvation. So, until now, DCTCP could only be
   deployed where a clean-slate environment could be arranged, such as
   in private data centres. This specification defines `DualQ Coupled
   Active Queue Management (AQM)' to allow scalable congestion controls
   like DCTCP to safely co-exist with classic Internet traffic. The
   Coupled AQM ensures that a flow runs at about the same rate whether
   it uses DCTCP or TCP Reno/Cubic, but without inspecting transport
   layer flow identifiers. When tested in a residential broadband
   setting, DCTCP achieved 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 eliminates network configuration.

Status of This Memo

   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Table of Contents

1. Introduction

1.1. Problem and Scope

   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
   component of latency.

   The Diffserv architecture provides Expedited Forwarding [RFC3246], so
   that low latency traffic can jump the queue of other traffic.
   However, on access links dedicated to individual sites (homes, small
   enterprises or mobile devices), often all traffic at any one time
   will be latency-sensitive and, if all the traffic on a link is marked
   as EF, Diffserv cannot reduce the delay of any of it. In contrast,
   the Low Latency Low Loss Scalable throughput (L4S) approach removes
   the causes of any unnecessary queuing delay.

   The bufferbloat project has shown that excessively-large buffering
   (`bufferbloat') has been introducing significantly more delay than
   the underlying propagation time. These delays appear only
   intermittently--only when a capacity-seeking (e.g. TCP) flow is long
   enough for the queue to fill the buffer, making every packet in other
   flows sharing the buffer sit through the queue.

   Active queue management (AQM) was originally developed to solve this
   problem (and others). Unlike Diffserv, which gives low latency to
   some traffic at the expense of others, AQM controls latency for _all_
   traffic in a class. In general, AQMs introduce an increasing level
   of discard from the buffer the longer the queue persists above a
   shallow threshold. This gives sufficient signals to capacity-seeking
   (aka. greedy) flows to keep the buffer empty for its intended
   purpose: absorbing bursts. However, RED [RFC2309] and other

algorithms from the 1990s were sensitive to their configuration and
hard to set correctly. So, AQM was not widely deployed.

More recent state-of-the-art AQMs, e.g. fq_CoDel [RFC8290],
PIE [RFC8033], Adaptive RED [ARED01], are easier to configure,
because they define the queuing threshold in time not bytes, so it is
invariant for different link rates. However, no matter how good the
AQM, the sawtoothing rate of TCP will either cause queuing delay to
vary or cause the link to be under-utilized. Even with a perfectly
tuned AQM, the additional queuing delay will be of the same order as
the underlying speed-of-light delay across the network. Flow-queuing
can isolate one flow from another, but it cannot isolate a TCP flow
from the delay variations it inflicts on itself, and it has other
problems - it overrides the flow rate decisions of variable rate
video applications, it does not recognise the flows within IPSec VPN
tunnels and it is relatively expensive to implement.

It seems that further changes to the network alone will now yield
diminishing returns. Data Centre TCP (DCTCP [RFC8257]) teaches us
that a small but radical change to TCP is needed to cut two major
outstanding causes of queuing delay variability:

1. the `sawtooth' varying rate of TCP itself;

2. the smoothing delay deliberately introduced into AQMs to permit
    bursts without triggering losses.

The former causes a flow's round trip time (RTT) to vary from about 1
to 2 times the base RTT between the machines in question. The latter
delays the system's response to change by a worst-case
(transcontinental) RTT, which could be hundreds of times the actual
RTT of typical traffic from localized CDNs.

Latency is not our only concern:

3. It was known when TCP was first developed that it would not scale
    to high bandwidth-delay products.

Given regular broadband bit-rates over WAN distances are
already [RFC3649] beyond the scaling range of `classic' TCP Reno,
`less unscalable' Cubic [RFC8312] and
Compound [I-D.sridharan-tcpm-ctcp] variants of TCP have been
successfully deployed. However, these are now approaching their
scaling limits. Unfortunately, fully scalable TCPs such as DCTCP
cause `classic' TCP to starve itself, which is why they have been
confined to private data centres or research testbeds (until now).

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. The AQM is not like
flow-queuing approaches [RFC8290] that classify packets by flow
identifier into numerous separate queues in order to isolate sparse
flows from the higher latency in the queues assigned to heavier flow.
In contrast, the AQM exploits the behaviour of scalable congestion
controls like DCTCP so that every packet in every flow sharing the
queue for DCTCP-like traffic can be served with very low latency.

This AQM extension can be combined with any single queue AQM that
generates a statistical or deterministic mark/drop probability driven
by the queue dynamics. In many cases it 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.

The overall L4S architecture is described in
[I-D.ietf-tsvwg-l4s-arch]. The supporting papers [PI2] and [DCttH15]
give the full rationale for the AQM's design, both discursively and
in more precise mathematical form.

1.2. Terminology

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

The 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 DCTCP (e.g.
   Relentless [Mathis09]).

Either service can cope with a proportion of unresponsive or less-
responsive traffic as well (e.g. DNS, VoIP, etc), just as a single
queue AQM can. The DualQ Coupled AQM behaviour is similar to a
single FIFO queue with respect to unresponsive and overload traffic.

1.3. Features

The AQM couples marking and/or dropping across the two queues such
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 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. Typical
experiments used base round trip delays up to 100ms between the data
centre and home network, and large amounts of background traffic in
both queues. For every L4S packet, the AQM kept the average queuing
delay below 1ms (or 2 packets if serialization delay is bigger for
slow links), and no losses at all were introduced by the AQM.
Details of the extensive experiments will be made available [PI2]
[DCttH15].

Subjective testing was also conducted using a demanding panoramic
interactive video application run over a stack with DCTCP enabled and
deployed on the testbed. Each user could pan or zoom their own high
definition (HD) sub-window of a larger video scene from a football
match. Even though the user was also downloading large amounts of
L4S and Classic data, latency was so low that the picture appeared to
stick to their finger on the touchpad (all the L4S data achieved the
same ultra-low latency). With an alternative AQM, the video
noticeably lagged behind the finger gestures.

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 like DCTCP 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 DCTCP-like flows
cannot keep latency predictably low and keep utilization high if they
are mixed with legacy TCP flows,

The experiments used the Linux implementation of DCTCP that is
deployed in private data centres, without any modification despite
its known deficiencies. Nonetheless, certain modifications will be

necessary before DCTCP is safe to use on the Internet, which are
recorded in [Appendix A](#) of [[I-D.ietf-tsvwg-ecn-l4s-id](#)]. However, the
focus of this specification is to get the network service in place.
Then, without any management intervention, applications can exploit
it by migrating to scalable controls like DCTCP, which can then
evolve _while_ their benefits are being enjoyed by everyone on the
Internet.

[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 (e.g. DCTCP) flows

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.

TCP Cubic implements a Reno-compatibility mode, which is the only
relevant mode for typical RTTs under 20ms as long as the throughput
of a single flow is less than about 500Mb/s. Therefore it can be
assumed that Cubic traffic behaves similarly to Reno (but with a
slightly different constant of proportionality), and 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 'L4S' traffic will be used for all similar
behaviour.

In order to make a DCTCP flow run at roughly the same rate as a Reno
TCP flow (all other factors being equal), the drop or marking
probability for Classic traffic, p_C has to be distinct from the
marking probability for L4S traffic, p_L (in contrast to [RFC3168](#)
which requires them to be the same). It is necessary to make the
Classic drop probability p_C proportional to the square of the L4S
marking probability p_L. This makes the Reno flow rate roughly equal
the DCTCP flow rate, because it squares the square root of p_C in the

Reno rate equation to make it proportional to the straight p_L in the DCTCP rate equation.

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

    p_C = (p_L / k)^2   (1)

where k is the constant of proportionality.

2.2. Dual Queue

Classic traffic typically builds a large queue to prevent under-utilization. Therefore a separate queue is provided for L4S traffic, and it is scheduled with priority over Classic. 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). The algorithm achieves this without having to inspect flow identifiers.

2.3. Traffic Classification

Both the Coupled AQM and DualQ mechanisms need an identifier to distinguish L and C packets. A separate draft [I-D.ietf-tsvwg-ecn-l4s-id] recommends using the ECT(1) codepoint 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, the classifier MUST NOT alter the ECN field, 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, other identifiers could be used to classify certain additional packet types into the L queue, that are deemed not to risk harming 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 DualQ Coupled AQM only reads these classifiers, it MUST
   NOT re-mark or alter these identifiers (except for marking the ECN
   field with the CE codepoint - with increasing frequency to indicate
   increasing congestion).

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 dropping or marking (p_L and p_C). Nonetheless, the
   AQM for Classic traffic is 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)

   This allows p_L to be coupled to p_C by marking L4S traffic
   proportionately to the intermediate output from the first stage.
   Specifically, the output of the base AQM is coupled across to the L
   queue in proportion to the output of the base AQM:

      $p_{CL} = k*p',$ (3)

   where k is the constant coupling factor (see Appendix C) and p_{CL} is
   the output from the coupling between the C queue and the L queue.

   It can be seen in the following that these two transformations of p'
   implement the required coupling given in equation (1) earlier.
   Substituting for p' from equation (3) into (2):

      $p_C = (p_{CL} / k)^2.$

   The actual L4S marking probability p_L is the maximum of the coupled
   output (p_{CL}) and the output of a native L4S AQM (p'L), shown as
   '(MAX)' in the schematic. While the output of the Native L4S AQM is
   high (p'L > p_{CL}) it will dominate the way L traffic is marked. When
   the native L4S AQM output is lower, the way L traffic is marked will

be driven by the coupling, that is p_L = p_CL. So, whenever the
coupling is needed, as required from equation (1):

    p_C = (p_L / k)^2.

```
                     _____
                    | |     ,------.
                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 |
                 __|_____|     `------'
```
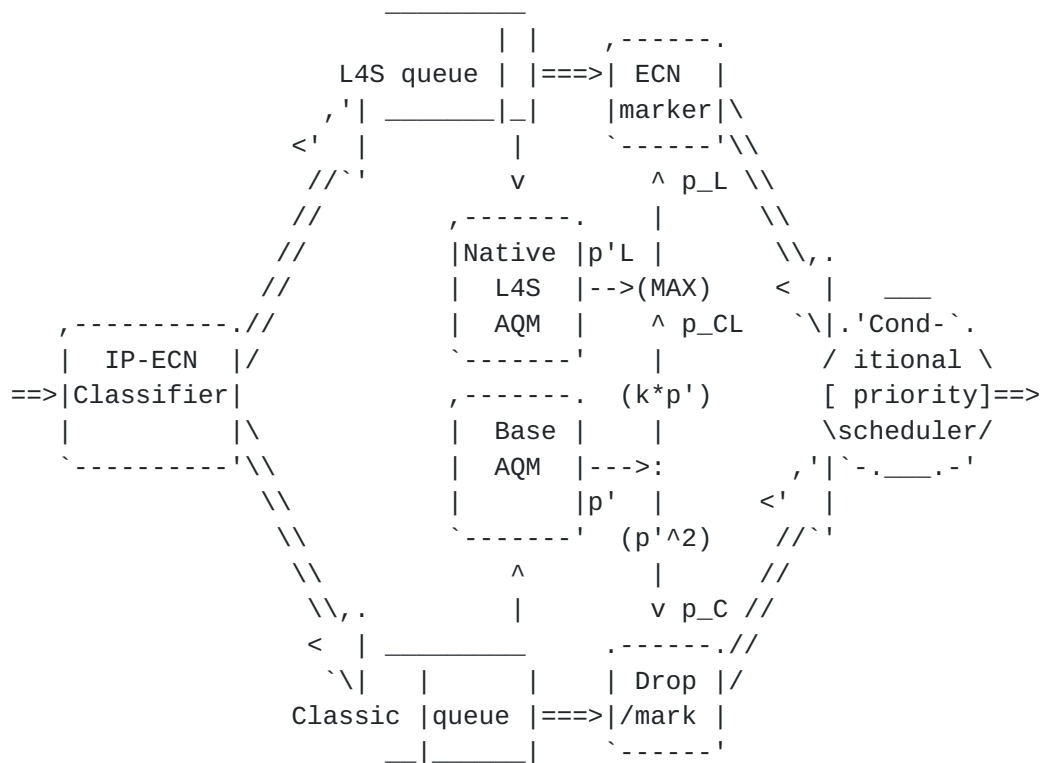
Legend: ===> traffic flow; ---> control dependency.

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

we call it just 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 attention than Curvy RED,
leaving the Curvy RED design incomplete and not so fully evaluated.

Both AQMs regulate their queue in units of time not 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

In the Dual Queue, L4S packets MUST be given priority over Classic,
although priority MUST be bounded in order not to starve Classic
traffic.

Whatever identifier is used for L4S experiments,
[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
prevent starvation of Classic traffic by 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)." In
other words, in any DualQ Coupled AQM, the power to which p_L is
raised in Eqn. (1) MUST be 2. The term 'likelihood' is used to allow
for marking and dropping to be either probabilistic or deterministic.

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.

{ToDo: The requirements for scalable congestion controls on the
Internet (termed the TCP Prague requirements)
[I-D.ietf-tsvwg-ecn-l4s-id] are not necessarily final. If the
aggressiveness of DCTCP is not defined as the benchmark for scalable
controls on the Internet, the recommended value of k will also be
subject to change.}

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 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 users.
However, on the public Internet, access network operators typically
isolate customers from each other with some form of layer-2
multiplexing (TDM in DOCSIS, CDMA in 3G) 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 L4S, and it will not
affect 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 as
        if the packet were ECT(1)

     * 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 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 MUST
           apply drop using the drop probability appropriate to the C
           queue

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

      * the C AQM SHOULD apply CE-marking as if the packet were ECT(0).

   If the DualQ Coupled AQM has detected overload, it will signal
   congestion solely using drop, irrespective of the ECN field.

   Most of 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.

2.5.2. Management Requirements

   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 (a parameter for typical or target queuing
      delay in each queue might be configurable instead);

   o Expected maximum RTT (a stability parameter that depends on
      maximum RTT might be configurable instead);

o Coupling factor, k;

o The limit to the conditional priority of L4S (scheduler-dependent,
   e.g. the scheduler weight for WRR, or the time-shift for time-
   shifted FIFO);

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

An experimental DualQ Coupled AQM SHOULD allow the operator to
monitor the following operational statistics:

o Bits forwarded (total and per queue per sample interval), from
   which utilization can be calculated

o Q delay (per queue over sample interval)

o Total packets arriving, enqueued and dequeued (per queue per
   sample interval)

o ECN packets marked, non-ECN packets dropped, ECN packets dropped
   (per queue per sample interval), from which marking and dropping
   probabilities can be calculated

o Time and duration of each overload event.

The type of statistics produced for variables like Q delay (mean,
percentiles, etc.) will depend on implementation constraints.

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. Some of these choices might need to be determined by operator
policy or by the developer, rather than by the IETF. {ToDo: Reach
consensus on which it is to be in each case.}

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

Priority of L4S is required to be conditional to avoid total
throughput 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 to guarantee a minimum throughput
   service for Classic traffic. 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 a large part of the capacity set aside for L4S, using
   WRR could significantly reduce the capacity left for any
   responsive L4S flows.

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 implementation in Appendix A can implement 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, but saturation can 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, 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.

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 applies only 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

Thanks to Anil Agarwal, Sowmini Varadhan's and Gabi Bracha for
detailed review comments particularly of the appendices and
suggestions on how to make our explanation clearer. Thanks also to
Greg White and Tom Henderson for insights on the choice of schedulers
and queue delay measurement techniques.

6. References

6.1. Normative References

   [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>.

6.2. Informative References

   [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>.

   [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, July
               2015,
               <http://www.bobbriscoe.net/projects/latency/credi_tr.pdf>.

   [DCttH15]   De Schepper, K., Bondarenko, O., Briscoe, B., and I.
               Tsang, "`Data Centre to the Home': Ultra-Low Latency for
               All", 2015, <http://www.bobbriscoe.net/projects/latency/
               dctth_preprint.pdf>.

               (Under submission)

   [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-00 (work in progress),
               March 2018.

   [I-D.ietf-tsvwg-ecn-l4s-id]
               Schepper, K., Briscoe, B., and I. Tsang, "Identifying
               Modified Explicit Congestion Notification (ECN) Semantics
               for Ultra-Low Queuing Delay", draft-ietf-tsvwg-ecn-l4s-
               id-02 (work in progress), March 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-02 (work in
          progress), March 2018.

[I-D.sridharan-tcpm-ctcp]
          Sridharan, M., Tan, K., Bansal, D., and D. Thaler,
          "Compound TCP: A New TCP Congestion Control for High-Speed
          and Long Distance Networks", draft-sridharan-tcpm-ctcp-02
          (work in progress), November 2008.

[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>.

          (To appear)

[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>.

[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>.

   [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>.

   [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>.

   [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>.

   [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>.

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 step threshold (in units of queuing
   time) is used for the Native L4S AQM, but a ramp is also described as
   an alternative. And the PI2 algorithm [PI2] is used for the Classic
   AQM. PI2 is an improved variant of the PIE AQM [RFC8033].

   We will introduce the pseudocode 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.

   A full open source implementation for Linux is available at:
   https://github.com/olgabo/dualpi2.

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 four functions:

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

   o enqueue code (Figure 3)

   o dequeue code (Figure 4)

   o code to regularly update the base probability (p) used in the
      dequeue code (Figure 5).

   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;

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]).

In our 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. 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:    % PI2 AQM parameters
3:    target = 15 ms              % PI AQM Classic queue delay target
4:    Tupdate = 16 ms            % PI Classic queue sampling interval
5:    alpha = 10 Hz^2                         % PI integral gain
6:    beta = 100 Hz^2                       % PI proportional gain
7:    p_Cmax = 1/4                    % Max Classic drop/mark prob
8:    % Derived PI2 AQM variables
9:    alpha_U = alpha *Tupdate % PI integral gain per update interval
10:   beta_U = beta * Tupdate  % PI prop'nal gain per update interval
11:
12:   % DualQ Coupled framework parameters
13:   k = 2                                    % Coupling factor
14:   % scheduler weight or equival't parameter (scheduler-dependent)
15:   limit = MAX_LINK_RATE * 250 ms           % Dual buffer size
16:
17:   % L4S AQM parameters
18:   T_time = 1 ms                    % L4S marking threshold in time
19:   T_len = 2 * MTU          % Min L4S marking threshold in bytes
20:   % Derived L4S AQM variables
21:   p_Lmax = min(k*sqrt(p_Cmax), 1)        % Max L4S marking prob
22: }
```

Figure 2: Example Header Pseudocode for DualQ Coupled PI2 AQM

The overall goal of the code is to maintain the base probability (p),
which is an internal variable from which the marking and dropping
probabilities for L4S and Classic traffic (p_L and p_C) are derived.
The variable named p in the pseudocode and in this walk-through is
the same as p' (p-prime) in Section 2.4. The probabilities p_L and
p_C are derived in lines 3, 4 and 5 of the dualpi2_update() function
(Figure 5) then used in the dualpi2_dequeue() function (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() > limit )
3:      drop(pkt)                      % drop packet if buffer is full
4:    else {                                   % Packet classifier
5:      if ( ecn(pkt) modulo 2 == 1 )      % ECN bits = ECT(1) or CE
6:        lq.enqueue(pkt)
7:      else                             % ECN bits = not-ECT or ECT(0)
8:        cq.enqueue(pkt)
9:    }
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:        if ( ((lq.time() > T_time)           % step marking ...
6:              AND (lq.len() > T_len))
7:            OR (p_CL > rand()) )             % ...or 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: }
```

   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. Note that the
limit is deliberately tested before enqueue to avoid any bias against
larger packets (so the actual buffer has to be one MTU larger than
limit). If limit is not exceeded, the packet will be classified and
enqueued to the Classic or L4S queue dependent on the least
significant bit of the ECN field in the IP header (line 5). 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 very sloppy.

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, lines 5 to 8 mark the packet if
   either the L4S threshold (T_time) is exceeded, or if a random
   marking decision is drawn according to p_CL (maintained by the
   dualpi2_update() function discussed below). This logical 'OR' on
   a per-packet basis implements the max() function shown in Figure 1
   to couple the outputs of the two AQMs together. The L4S threshold
   is usually in units of time (default T_time = 1 ms). However, on
   slow links the packet serialization time can approach the
   threshold T_time, so line 6 sets a floor of T_len (=2 MTU) to the
   threshold, otherwise marking is always too frequent on slow links.

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

There is some concern that using a step function for the Native L4S
AQM requires end-systems to smooth the signal for a lot longer -
until its fidelity is sufficient. The latency benefits of a ramp are

being investigated as a simple alternative to the step. This ramp
would be similar to the RED algorithm, with the following
differences:

o The min and max of the ramp are 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 without smoothing (smoothing
   is done in the end-systems).

o Determinism is being experimented with instead of randomness; to
   reduce the delay necessary to smooth out the noise of randomness
   from the signal. For each packet, the algorithm would accumulate
   p'_L in a counter and mark the packet that took the counter over
   1, then subtract 1 from the counter and continue.

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.

This ramp algorithm would require two configuration parameters (min
and max threshold in units of queuing time), in contrast to the
single parameter of a step.

```
1:  dualpi2_update(lq, cq, target) {         % Update p every Tupdate
2:    curq = cq.time()  % use queuing time of first-in Classic packet
3:    p = p + alpha_U * (curq - target) + beta_U * (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 5: Example PI-Update Pseudocode for DualQ Coupled PI2 AQM

The base probability (p) is kept up to date by the core PI algorithm
in Figure 5, which is 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 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_U and beta_U, 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.

alpha_U and beta_U are derived from the input parameters alpha and
beta (see lines 5 and 6 of Figure 2). These recommended values of
alpha and beta come from the stability analysis in [PI2] so that the
AQM can change p as fast as possible in response to changes in load
without over-compensating and therefore causing oscillations in the
queue.

alpha and beta determine how much p ought to change if it was updated
every second. It is best to update p as frequently as possible, but
the update interval (Tupdate) will probably be constrained by
hardware performance. For link rates from 4 - 200 Mb/s, we found
Tupdate=16ms (as recommended in [RFC8033]) is sufficient. However
small the chosen value of Tupdate, p should change by the same amount
per second, but in finer more frequent steps. So the gain factors
used for updating p in Figure 5 need to be scaled by (Tupdate/1s),
which is done in lines 9 and 10 of Figure 2). The suffix '_U'
represents 'per update time' (Tupdate).

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 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, which is necessary to ensure that
Classic TCP and DCTCP controls have the same stability. So, if alpha
is 10 Hz^2, the effective gain factor for the L4S queue is k*alpha,
which is 20 Hz^2 with the default coupling factor of k=2.

Unlike in PIE [RFC8033], alpha_U and beta_U do not need to be tuned
every Tupdate dependent on p. Instead, in PI2, alpha_U and beta_U
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.

{ToDo: Scaling beta with Tupdate and scaling both alpha & beta with
RTT}

[A.2](#). Pass #2: Overload Details

Figure 6 repeats the dequeue function of Figure 4, but with overload
details added. Similarly Figure 7 repeats the core PI algorithm of
Figure 5 with overload details added. The initialization and enqueue
functions are unchanged.

In line 7 of the initialization function (Figure 2), the default
maximum Classic drop probability p_Cmax = 1/4 or 25%. This is the
point at which it is deemed that the Classic queue has become
persistently overloaded, so it switches to using solely drop, even
for ECN-capable packets. This protects the queue against any
unresponsive traffic that falsely claims that it is responsive to ECN
marking, as required by [[RFC3168](#)] and [[RFC7567](#)].

Line 21 of the initialization function translates this into a maximum
L4S marking probability (p_Lmax) by rearranging Equation (1). With a
coupling factor of k=2 (the default) or greater, this translates to a
maximum L4S marking probability of 1 (or 100%). This is intended to
ensure that the L4S queue starts to introduce dropping once marking
saturates and can rise no further. The 'TCP Prague' 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 the L4S queue drops
packets 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 6).
Lines 8c to 8g drop L4S packets with probability p^2. Lines 8h to 8i
mark the remaining packets with probability p_CL.

Lines 2c to 2d in the core PI algorithm (Figure 7) 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
algorithm in Figure 5 drops nothing, even if the L4S queue is
overloaded - so tail drop would have to take over (lines 3 and 4 of
Figure 3).

If the test at line 2a 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_L will be driven to
zero, and the L4S queue will naturally be governed solely by

threshold marking (lines 5 and 6 of the dequeue algorithm in
Figure 6). But, if unresponsive L4S source(s) cause overload, the
DualQ transitions smoothly to L4S marking based on the PI algorithm.
And as overload increases, it naturally transitions from marking to
dropping by the switch-over mechanism already described.

```
1:  dualpi2_dequeue(lq, cq) { % Couples L4S & Classic queues, lq & cq
2:    while ( lq.len() + cq.len() > 0 )
3:      if ( scheduler() == lq ) {
4a:       lq.dequeue(pkt)
4b:       if ( p_CL < p_Lmax ) {      % Check for overload saturation
5:          if ( ((lq.time() > T_time)          % step marking ...
6:               AND (lq.len > T_len))
7:              OR (p_CL > rand()) )         % ...or 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)
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 6: Example Dequeue Pseudocode for DualQ Coupled PI2 AQM
          (Including Integer Arithmetic and Overload Code)

```
1:  dualpi2_update(lq, cq, target) {        % 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_U * (curq - target) + beta_U * (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 7: 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. The scheduler weight for Classic should be
      low, e.g. 1/16.

   o Alternatively, a time-shifted FIFO could be used. This is a very
      simple scheduler, but it does not fully isolate latency in the L4S
      queue from uncontrolled bursts in the Classic queue. 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 "if (scheduler() == lq)" test in
      line 3 of the dequeue code would simply be replaced by "if (
      lq.time() + tshift >= cq.time())". For the public Internet a
      good value for tshift is 50ms. For private networks with smaller
      diameter, about 4*target would be reasonable.

   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 we used and tested.
   Although we designed the AQM to be efficient in integer arithmetic,
   to aid understanding it is first given using real-number arithmetic.
   Then, one possible optimization for integer arithmetic is given, also
   in pseudocode. To aid comparison, the line numbers are kept in step
   between the two by using letter suffixes where the longer code needs
   extra lines.

```
 1:  dualq_dequeue(lq, cq) {  % Couples L4S & Classic queues, lq & cq
 2:    if ( lq.dequeue(pkt) ) {
3a:      p_L = cq.sec() / 2^S_L
3b:      if ( lq.byt() > T )
3c:        mark(pkt)
3d:      elif ( p_L > maxrand(U) )
 4:        mark(pkt)
 5:      return(pkt)                  % return the packet and stop here
 6:    }
 7:    while ( cq.dequeue(pkt) ) {
8a:      alpha = 2^(-f_C)
8b:      Q_C = alpha * pkt.sec() + (1-alpha)* Q_C    % Classic Q EWMA
9a:      sqrt_p_C = Q_C / 2^S_C
9b:      if ( sqrt_p_C > maxrand(2*U) )
10:        drop(pkt)                        % Squared drop, redo loop
11:      else
12:        return(pkt)               % return the packet and stop here
13:    }
14:    return(NULL)                          % no packet to dequeue
15: }

16: maxrand(u) {                   % return the max of u random numbers
17:      maxr=0
18:      while (u-- > 0)
19:          maxr = max(maxr, rand())            % 0 <= rand() < 1
20:      return(maxr)
21: }
```

Figure 8: Example Dequeue Pseudocode for DualQ Coupled Curvy RED AQM

Packet classification code is not shown, as it is no different from
Figure 3. Potential classification schemes are discussed in
Section 2.3. The Curvy RED algorithm has not been maintained to the
same degree as the DualPI2 algorithm. Some ideas used in DualPI2
would need to be translated into Curvy RED, such as i) the
conditional priority scheduler instead of strict priority ii) the
time-based L4S threshold; iii) turning off ECN as overload
protection; iv) Classic ECN support. These are not shown in the
Curvy RED pseudocode, but would need to be implemented for
production. {ToDo}

At the outer level, the structure of dualq_dequeue() implements
strict priority scheduling. The code is written assuming the AQM is
applied on dequeue (Note 1) . Every time dualq_dequeue() is called,
the if-block in lines 2-6 determines whether there is an L4S packet
to dequeue by calling lq.dequeue(pkt), and otherwise the while-block
in lines 7-13 determines whether there is a Classic packet to
dequeue, by calling cq.dequeue(pkt). (Note 2)

In the lower priority Classic queue, a while loop is used so that, if
the AQM determines that a classic packet should be dropped, it
continues to test for classic packets deciding whether to drop each
until it actually forwards one. Thus, every call to dualq_dequeue()
returns one packet if at least one is present in either queue,
otherwise it returns NULL at line 14. (Note 3)

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

L4S: If the test at line 2 determines there is an L4S packet to
   dequeue, the tests at lines 3a and 3c determine whether to mark
   it. The first is a simple test of whether the L4S queue (lq.byt()
   in bytes) is greater than a step threshold T in bytes (Note 4).
   The second test is similar to the random ECN marking in RED, but
   with the following differences: i) the marking function does not
   start with a plateau of zero marking until a minimum threshold,
   rather the marking probability starts to increase as soon as the
   queue is positive; ii) marking depends on queuing time, not bytes,
   in order to scale for any link rate without being reconfigured;
   iii) marking of the L4S queue does not depend on itself, it
   depends on the queuing time of the _other_ (Classic) queue, where
   cq.sec() is the queuing time of the packet at the head of the
   Classic queue (zero if empty); iv) marking depends on the
   instantaneous queuing time (of the other Classic queue), not a
   smoothed average; v) 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 3a the marking probability p_L is set to the
   Classic queueing time qc.sec() in seconds divided by the L4S
   scaling parameter 2^S_L, which represents the queuing time (in
   seconds) at which marking probability would hit 100%. Then in line
   3d (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. The
   scaling parameter is expressed as a power of 2 so that division
   can be implemented as a right bit-shift (>>) in line 3 of the
   integer variant of the pseudocode (Figure 9).

Classic: If the test at line 7 determines that there is at least one
   Classic packet to dequeue, the test at line 9b determines whether
   to drop it. But before that, line 8b updates Q_C, which is an
   exponentially weighted moving average (Note 5) of the queuing time
   in the Classic queue, where pkt.sec() is the instantaneous
   queueing time of the current Classic packet and alpha is the EWMA
   constant for the classic queue. In line 8a, alpha is represented
   as an integer power of 2, so that in line 8 of the integer code

the division needed to weight the moving average can be
implemented by a right bit-shift (>> f_C).

Lines 9a and 9b implement the drop function. In line 9a the
averaged queuing time Q_C is divided by the Classic scaling
parameter 2^S_C, in the same way that queuing time was scaled for
L4S marking. This scaled queuing time is given the variable name
sqrt_p_C because it will be squared to compute Classic drop
probability, so before it is squared it is effectively the square
root of the drop probability. 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 (Note 6). Again, the scaling
parameter is expressed as a power of 2 so that division can be
implemented as a right bit-shift in line 9 of the integer
pseudocode.

The marking/dropping functions in each queue (lines 3 & 9) are two
cases of a new generalization of RED called Curvy RED, motivated as
follows. When we compared the performance of our AQM with fq_CoDel
and PIE, we came to the conclusion that their goal of holding queuing
delay to a fixed target is 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 one parameter, the scaling, not three. The diadvantage
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 support is limited
otherwise an adaptation mechanism is required.

There follows a summary listing of the two parameters used for each
of the two queues:

Classic:

   S_C :   The scaling factor of the dropping function scales Classic
      queuing times in the range [0, 2^(S_C)] seconds into a dropping
      probability in the range [0,1]. To make division efficient, it
      is constrained to be an integer power of two;

   f_C : To smooth the queuing time of the Classic queue and make
      multiplication efficient, we use a negative integer power of
      two for the dimensionless EWMA constant, which we define as
      alpha = 2^(-f_C).

   L4S :

     S_L (and k'):   As for the Classic queue, the scaling factor of
        the L4S marking function scales Classic queueing times in the
        range [0, 2^(S_L)] seconds into a probability in the range
        [0,1]. Note that S_L = S_C + k', where k' is the coupling
        between the queues. So S_L and k' count as only one parameter;
        k' is related to k in Equation (1) (Section 2.1) by k=2^k',
        where both k and k' are constants. Then implementations can
        avoid costly division by shifting p_L by k' bits to the right.

     T : The queue size in bytes at which step threshold marking
        starts in the L4S queue.

   {ToDo: These are the raw parameters used within the algorithm. A
   configuration front-end could accept more meaningful parameters and
   convert them into these raw parameters.}

   From our experiments so far, recommended values for these parameters
   are: S_C = -1; f_C = 5; T = 5 * MTU 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) {ToDo
   incorporate a summary of that report into this draft}. 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. We
   have solely used U=1 in our experiments so far, but results might be
   even better with U=2 or higher.

   Note that the dropping function at line 9 calls maxrand(2*U), which
   gives twice as much curviness as the call to maxrand(U) in the
   marking function at line 3. 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 (Note 7). 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:  dualq_dequeue(lq, cq) {  % Couples L4S & Classic queues, lq & cq
2:     if ( lq.dequeue(pkt) ) {
3:        if ((lq.byt() > T) || ((cq.ns() >> (S_L-2)) > maxrand(U)))
4:           mark(pkt)
5:        return(pkt)              % return the packet and stop here
6:     }
7:     while ( cq.dequeue(pkt) ) {
8:        Q_C += (pkt.ns() - Q_C) >> f_C          % Classic Q EWMA
9:        if ( (Q_C >> (S_C-2) ) > maxrand(2*U) )
10:          drop(pkt)                   % Squared drop, redo loop
11:       else
12:          return(pkt)         % return the packet and stop here
13:    }
14:    return(NULL)                        % no packet to dequeue
15: }
```

Figure 9: Optimised Example Dequeue Pseudocode for Coupled DualQ AQM
                    using Integer Arithmetic

Notes:

1. 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 [CoDel]. In our Linux
    implementation, it was easiest to measure queuing time at
    dequeue. Queuing time can be estimated when a packet is enqueued
    by measuring the queue length in bytes and dividing by the recent
    drain rate.

2. An implementation has to use priority queueing, but it need not
    implement strict priority.

3. If packets can be enqueued while processing dequeue code, an
    implementer might prefer to place the while loop around both
    queues so that it goes back to test again whether any L4S packets
    arrived while it was dropping a Classic packet.

4. In order not to change too many factors at once, for now, we keep
    the marking function for DCTCP-only traffic as similar as
    possible to DCTCP. However, unlike DCTCP, all processing is at
    dequeue, so we determine whether to mark a packet at the head of
    the queue by the byte-length of the queue _behind_ it. We plan
    to test whether using queuing time will work in all
    circumstances, and if we find that the step can cause
    oscillations, we will investigate replacing it with a steep
    random marking curve.

5. 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.

6. In practice at line 10 the Classic queue would probably test for
    ECN capability on the packet to determine whether to drop or mark
    the packet. However, for brevity such detail is omitted. All
    packets classified into the L4S queue have to be ECN-capable, so
    no dropping logic is necessary at line 3. Nonetheless, L4S
    packets could be dropped by overload code (see Section 4.1).

7. In the integer variant of the pseudocode (Figure 9) real numbers
    are all represented as integers scaled up by 2^{32}. In lines 3 &
    9 the function maxrand() is arranged to return an integer in the
    range $0 <= maxrand() < 2^{32}$. Queuing times are also scaled up by
    2^{32}, but in two stages: i) In lines 3 and 8 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}.

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, we used
the measured RTTs to calculate that a value of k'=3 (equivalant to
k=8) would achieve throughput equivalence, and our 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.

Appendix D. Open Issues

Most of the following open issues are also tagged '{ToDo}' at the
appropriate point in the document:

    Operational guidance to monitor L4S experiment

    PI2 appendix: scaling of alpha & beta, esp. dependence of beta_U
    on Tupdate

    Curvy RED appendix: complete the unfinished parts

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/

    Olga Bondarenko
    Simula Research Lab
    Lysaker
    Norway

    Email: olgabnd@gmail.com
    URI: https://www.simula.no/people/olgabo

    Ing-jyh Tsang
    Nokia
    Antwerp
    Belgium

    Email: ing-jyh.tsang@nokia.com