Active Queue Management (aqm)
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2016

K. De Schepper
Nokia Bell Labs
B. Briscoe, Ed.
O. Bondarenko
Simula Research Lab
I. Tsang
Nokia Bell Labs
March 21, 2016

DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput
draft-briscoe-aqm-dualq-coupled-01

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---throughput starves. 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.

This Internet-Draft will expire on September 22, 2016.

Table of Contents

1. Introduction 2
1.1. Problem and Scope 2
1.2. Terminology 5
1.3. Features 5
2. DualQ Coupled AQM Algorithm 6
2.1. Coupled AQM 6
2.2. Dual Queue 7
2.3. Traffic Classification 8
2.4. Normative Requirements 8
3. IANA Considerations 9
4. Security Considerations 9
4.1. Overload Handling 9
5. Acknowledgements 10
6. References 10
6.1. Normative References 10
6.2. Informative References 10
Appendix A. Example DualQ Coupled Algorithm 13
Appendix B. Guidance on Controlling Throughput Equivalence . . . 19
Appendix C. DCTCP Safety Enhancements 20
Authors' Addresses 21

1. Introduction

1.1. Problem and Scope

Latency is becoming the critical performance factor for many (most?) applications on the public Internet, e.g. Web, voice, conversational video, gaming, finance apps, remote desktop and cloud-based applications. 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. Then Diffserv is of little use. Instead, we need to remove the causes of any unnecessary 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 [I-D.ietf-aqm-fq-codel], PIE [I-D.ietf-aqm-pie], 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 [I-D.bensley-tcpm-dctcp]) 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 [I-D.zimmermann-tcpm-cubic] 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 [I-D.ietf-aqm-fq-codel] 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 qeueu 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 supporting paper [DCttH15] gives 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).

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 [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 for now in Appendix C. 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 Algorithm

There are two main aspects to the algorithm:

- the Coupled AQM that addresses throughput equivalence between Classic (e.g. Reno, Cubic) flows and L4S (e.g. DCTCP) flows

- 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, while the throughput of a single flow is less than about 500Mb/s. Therefore we can assume that Cubic traffic behaves similar to Reno (but with a slightly different constant of proportionality), and we shall use the term 'Classic' for the collection of Reno and Cubic in Reno mode.

In our supporting paper [DCttH15], we derive 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 we use the term 'L4S' traffic 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), we make the drop probability for Classic traffic, p_C distinct from the marking probability for L4S traffic, p_L (in contrast to RFC3168 which requires them to be the same). We make the Classic drop probability p_C proportional to the square of the L4S marking probability p_L. This is because we need to make the Reno flow rate equal the DCTCP flow rate, so we have to square the square root of p_C in the Reno rate equation to make it the same as the straight p_L in the DCTCP rate equation.

There is a really simple way to implement the square of a probability - by testing the queue against two random numbers not one. This is the approach adopted in Appendix A.

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

$$p_C = (p_L / k)^2 \qquad (1)$$

where k is the constant of proportionality. Optionally, k can also be expressed as a power of 2 so that implementations can avoid costly division by shifting p_L by log2(k)=k' bits to the right.

2.2. Dual Queue

Classic traffic builds a large queue, so a separate queue is provided for L4S traffic, and it is scheduled with strict priority. 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 L4S and C packets, which will need to be standardized. This draft does not currently recommend an approach for identifying for the L4S service, which is initially left open for discussion within the IETF. Another draft [I-D.briscoe-tsvwg-ecn-l4s-id] is submitted for that purpose.

2.4. Normative Requirements

In the Dual Queue, L4S packets MUST be given priority over Classic, although strict priority MAY not be appropriate.

All L4S traffic MUST be ECN-capable, although some Classic traffic MAY also be ECN-capable.

Whatever identifier is used for L4S traffic, it will still be necessary to agree on the meaning of an ECN marking on L4S traffic, relative to a drop of Classic traffic. In order to prevent starvation of Classic traffic by scalable L4S traffic (e.g. DCTCP) the drop probability of Classic traffic MUST be proportional to the square of the marking probability of L4S traffic, In other words, the power to which p_L is raised in Eqn. (1) MUST be 2.

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). k does not have to be standardized because differences do not prevent interoperability. However, k has to take some value, and each operator can make that choice.

A value of k=1 is RECOMMENDED as the default for public Internet access networks, assuming the scalable TCP algorithm operates at an comparable rate to classic tcp. Nonetheless choice of k is a matter of operator policy, and operators MAY choose a different value using Table 1 and the guidelines in Appendix B.

Typically, access network operators isolate customers from each other with some form of layer-2 multiplexing (TDM in DOCSIS, CDMA in 3G) or L3 scheduling (WRR in broadband), 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 the customer's access capacity, not between customers. Also, k would not affect rates of small flows, nor long flows at any times when they are all Classic or all L4S.

An example DualQ Coupled AQM algorithm is given in Appendix A. Marking and dropping in each queue is based on an AQM called Curvy RED. Curvy RED requires less operations per packet than RED and can be used if the range of RTTs is limited. Nonetheless, it would be possible to control each queue with an alternative AQM, as long as the above normative requirements (those expressed in capitals) are observed, which are intended to be independent of the specific AQM. Other experiments show that also PIE can be used and is simplified by applying a squared drop probability for classic TCP.

{ToDo: Add management and monitoring requirements}

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 performance. This is a policy issue that needs to be separable from a basic AQM, but the scheme does need to handle overload. A trade-off needs to be made between complexity and the risk of either class harming the other. It is an operator policy to define what must happen if the service time of the classic queue becomes too great. In the following subsections three optional non-exclusive overload protections are defined. Their objective is for the overload behaviour of the DualQ AQM to be similar to a single queue AQM. Other overload protections can be envisaged:

Minimum throughput service: By replacing the priority scheduler with a weighted round robin scheduler, a minimum throughput service can be guaranteed for Classic traffic. Typically the scheduling weight of the Classic queue will be small (e.g. 5%) to avoid interference with the coupling but big enough to avoid complete starvation of Classic traffic.

Drop on overload: On severe overload, e.g. due to non responsive traffic, queues will typically overflow and packet drop will be unavoidable when the queues reach their limits. The drop-limit of each queue should be configured by specifying the maximum supported load and determining the expected maximum size of each queue when that load is separately applied to each queue. The Classic queue limit will typically be larger than the L4S queue limit. Overflow of one traffic type will automatically result in drop in its respective queue. Both traffic types will get a high

congestion signal, due to the coupled marking, which will result in similar starvation of responsive traffic in both queues. Thus, the behaviour will be like a single queue AQM. To further improve the arrival fairness of a single queue an extra overall AQM limit can be applied, which is a limit to the sum of both queues. To be effective, it should be configured to be less than the sum of the limits of both queues, but greater than the maximum individual queue limit. It ensures that the drop probability of unresponsive traffic will be independent of its traffic type.

Delay on overload: To control milder overload of responsive traffic, particularly when close to the maximum congestion signal, delay can be used as an alternative congestion control mechanism. The Dual Queue Coupled AQM can be made to behave like a single FIFO queue with differentiated service times by replacing the priority scheduler with a very simple "biased longest sojourn time first scheduler". The bias is defined as a maximum sojourn time difference (T_m) between the Classic and L4S packets. The scheduler adds T_m to the sojourn time of the next L4S packet, before comparing it with the sojourn time of the next Classic packet, then it selects the packet with the greater adjusted sojourn time. This time shifted FIFO queue behaves just like a single FIFO queue under moderate and high overload.

5. Acknowledgements

Thanks to Anil Agarwal for detailed review comments and suggestions on how to make our explanation clearer.

The authors' contributions are part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed here are solely those of the authors.

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, <http://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>.

[DCTCP_Pitfalls] Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15) 145--157, May 2015, <http://blogs.usenix.org/conference/nsdi15/technical-sessions/presentation/judd>.

[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)

[ECN_Deploy] Trammell, B., Kuehlewind, M., Boppart, D., Learmonth, I., Fairhurst, G., and R. Scheffenegger, "Enabling Internet-Wide Deployment of Explicit Congestion Notification", Proc Passive & Active Measurement (PAM'15) Conference , 2015, <http://ecn.ethz.ch/ecn-pam15.pdf>.

[I-D.bensley-tcpm-dctcp] Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and G. Judd, "Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters", draft-bensley-tcpm-dctcp-05 (work in progress), July 2015.

[I-D.briscoe-tsvwg-ecn-l4s-id] Schepper, K., Briscoe, B., and I. Tsang, "Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay", draft-briscoe-tsvwg-ecn-l4s-id-01 (work in progress), March 2016.

[I-D.ietf-aqm-fq-codel]
Hoeiland-Joergensen, T., McKenney, P., dave.taht@gmail.com, d., Gettys, J., and E. Dumazet, "The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm", draft-ietf-aqm-fq-codel-06 (work in progress), March 2016.

[I-D.ietf-aqm-pie]
Pan, R., Natarajan, P., and F. Baker, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem", draft-ietf-aqm-pie-05 (work in progress), March 2016.

[I-D.ietf-tcpm-accecn-reqs]
Kuehlewind, M., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for a More Accurate ECN Feedback", draft-ietf-tcpm-accecn-reqs-08 (work in progress), March 2015.

[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.

[I-D.zimmermann-tcpm-cubic]
Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", draft-zimmermann-tcpm-cubic-01 (work in progress), April 2015.

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

[RFC0970] Nagle, J., "On Packet Switches With Infinite Storage", RFC 970, DOI 10.17487/RFC0970, December 1985, <http://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, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc3246>.

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

[RFC4774] Floyd, S., "Specifying Alternate Semantics for the Explicit Congestion Notification (ECN) Field", BCP 124, RFC 4774, DOI 10.17487/RFC4774, November 2006, <http://www.rfc-editor.org/info/rfc4774>.

[TCP-sub-mss-w] Briscoe, B. and K. De Schepper, "Scaling TCP's Congestion Window for Small Round Trip Times", BT Technical Report TR-TUB8-2015-002, May 2015, <http://www.bobbriscoe.net/projects/latency/sub-mss-w.pdf>.

Appendix A. Example DualQ Coupled Algorithm

As a concrete example, the pseudocode below gives the DualQ Coupled AQM algorithm we used in testing. 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 1: Example Dequeue Pseudocode for Coupled DualQ AQM

Packet classification code is not shown, as it is no different from regular packet classification. Potential classification schemes are discussed in Section 2. Overload protection code will be included in a future draft {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 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 2).

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 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 (Section 2.1). So S_L and k count as only one parameter;

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 2.4 and Appendix B 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 2: 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 2) 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 B. 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

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 the recommended value of k=0 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]:

$$2^k = 1.64 \, (RTT_{reno} / RTT_{dc}) \quad (2)$$

$$2^k = 1.19 \, (RTT_{cubic} / RTT_{dc}) \quad (3)$$

For localized traffic from a particular ISP's data centre, we used the measured RTTs to calculate that a value of k=3 would achieve throughput equivalence, and our experiments verified the formula very closely.

Appendix C. DCTCP Safety Enhancements

This Appendix is informational not normative. It records changes needed to DCTCP implementations so they can co-exist safely alongside other traffic sources. They are recorded here until a more appropriate draft is available to hold them.

Proposed changes are listed in rough order of criticality. Therefore those later in the list may not be necessary:

- Negotiate its altered feedback semantics, which conveys the extent of ECN marking, not just its existence, and this feedback needs to be robust to loss [I-D.ietf-tcpm-accecn-reqs];

- fall back to Reno or Cubic behaviour on loss;

- use a packet identifier associated with the L4S service;

- average ECN feedback over its own RTT, not the hard-coded RTT suitable only for data-centres, perhaps like Relentless TCP [Mathis09];

- make its throughput less or non-dependent on the base RTT, as smaller queues increase the RTT unfairness of TCP;
- handle a window of less than 2 when the RTT is low, rather than increase the queue [TCP-sub-mss-w].
- test heuristically whether ECN marking is emanating from an RFC3168 AQM.

Other, non-essential enhancements to DCTCP can be envisaged.

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)
Simula Research Lab

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 Bell Labs
Antwerp
Belgium

Email: ing-jyh.tsang@nokia.com