

AQM working group  
Internet-Draft  
Intended status: Experimental  
Expires: September 19, 2016

T. Hoeiland-Joergensen  
Karlstad University  
P. McKenney  
IBM Linux Technology Center  
D. Taht  
Teklibre  
J. Gettys

E. Dumazet  
Google, Inc.  
March 18, 2016

**The FlowQueue-CoDel Packet Scheduler and Active Queue Management  
Algorithm  
draft-ietf-aqm-fq-codel-06**

Abstract

This memo presents the FQ-CoDel hybrid packet scheduler/Active Queue Management algorithm, a powerful tool for fighting bufferbloat and reducing latency.

FQ-CoDel mixes packets from multiple flows and reduces the impact of head of line blocking from bursty traffic. It provides isolation for low-rate traffic such as DNS, web, and videoconferencing traffic. It improves utilisation across the networking fabric, especially for bidirectional traffic, by keeping queue lengths short; and it can be implemented in a memory- and CPU-efficient fashion across a wide range of hardware.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

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

This Internet-Draft will expire on September 19, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

- [1. Introduction](#) . . . . . [3](#)
- [1.1. Conventions used in this document](#) . . . . . [4](#)
- [1.2. Terminology and concepts](#) . . . . . [4](#)
- [1.3. Informal summary of FQ-CoDel](#) . . . . . [5](#)
- [2. CoDel](#) . . . . . [6](#)
- [3. Flow Queueing](#) . . . . . [6](#)
- [4. The FQ-CoDel scheduler](#) . . . . . [7](#)
- [4.1. Enqueue](#) . . . . . [7](#)
- [4.1.1. Alternative classification schemes](#) . . . . . [8](#)
- [4.2. Dequeue](#) . . . . . [9](#)
- [5. Implementation considerations](#) . . . . . [10](#)
- [5.1. Data structures](#) . . . . . [10](#)
- [5.2. Parameters](#) . . . . . [11](#)
- [5.2.1. Interval](#) . . . . . [11](#)
- [5.2.2. Target](#) . . . . . [11](#)
- [5.2.3. Packet limit](#) . . . . . [11](#)
- [5.2.4. Quantum](#) . . . . . [12](#)
- [5.2.5. Flows](#) . . . . . [12](#)
- [5.2.6. Explicit Congestion Notification \(ECN\)](#) . . . . . [12](#)
- [5.2.7. CE threshold](#) . . . . . [13](#)
- [5.3. Probability of hash collisions](#) . . . . . [13](#)
- [5.4. Memory Overhead](#) . . . . . [13](#)
- [5.5. Per-Packet Timestamping](#) . . . . . [14](#)
- [5.6. Limiting queueing in lower layers](#) . . . . . [15](#)
- [5.7. Other forms of "Fair Queueing"](#) . . . . . [15](#)
- [5.8. Differences between CoDel and FQ-CoDel behaviour](#) . . . . . [15](#)
- [6. Limitations of flow queueing](#) . . . . . [16](#)
- [6.1. Fairness between things other than flows](#) . . . . . [16](#)
- [6.2. Flow bunching by opaque encapsulation](#) . . . . . [17](#)
- [6.3. Low-priority congestion control algorithms](#) . . . . . [17](#)
- [7. Deployment status and future work](#) . . . . . [18](#)



<a href="#">8.</a>	Security Considerations . . . . .	<a href="#">18</a>
<a href="#">9.</a>	IANA Considerations . . . . .	<a href="#">19</a>
<a href="#">10.</a>	Acknowledgements . . . . .	<a href="#">19</a>
<a href="#">11.</a>	References . . . . .	<a href="#">19</a>
<a href="#">11.1.</a>	Normative References . . . . .	<a href="#">19</a>
<a href="#">11.2.</a>	Informative References . . . . .	<a href="#">21</a>
	Authors' Addresses . . . . .	<a href="#">22</a>

## [1.](#) Introduction

The FlowQueue-CoDel (FQ-CoDel) algorithm is a combined packet scheduler and Active Queue Management (AQM) [[RFC3168](#)] algorithm developed as part of the bufferbloat-fighting community effort [[BLOATWEB](#)]. It is based on a modified Deficit Round Robin (DRR) queue scheduler [[DRR](#)][[DRRPP](#)], with the CoDel AQM [[I-D.ietf-aqm-codel](#)] algorithm operating on each queue. This document describes the combined algorithm; reference implementations are available for the ns2 [[NS2](#)] and ns3 [[NS3](#)] network simulators, and it is included in the mainline Linux kernel as the fq\_codel queueing discipline [[LINUXSRC](#)].

FQ-CoDel is a general, efficient, nearly parameterless queue management approach combining flow queueing with CoDel. It is a powerful tool for solving bufferbloat [[BLOAT](#)], and we believe it to be safe to turn on by default, as has already happened in a number of Linux distributions. In this document we document the Linux implementation in sufficient detail for an independent implementation, to enable deployment outside of the Linux ecosystem.

Since the FQ-CoDel algorithm was originally developed in the Linux kernel, that implementation is still considered canonical. This document strives to describe the algorithm in the abstract in the first sections and separate out most implementation details in subsequent sections, but does use the Linux implementation as reference for default behaviour in the algorithm description itself.

The rest of this document is structured as follows: This section gives some concepts and terminology used in the rest of the document, and gives a short informal summary of the FQ-CoDel algorithm. [Section 2](#) gives an overview of the CoDel algorithm. [Section 3](#) covers the flow hashing and DRR portion. [Section 4](#) then describes the working of the algorithm in detail, while [Section 5](#) describes implementation details and considerations. [Section 6](#) lists some of the limitations of using flow queueing. Finally, [Section 7](#) outlines the current status of FQ-CoDel deployment and lists some possible future areas of inquiry, and [Section 8](#) reiterates some important security points that must be observed in the implementation.



### **1.1. Conventions used in this document**

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 [[RFC2119](#)] significance.

### **1.2. Terminology and concepts**

**Flow:** A flow is typically identified by a 5-tuple of source IP, destination IP, source port, destination port, and protocol number. It can also be identified by a superset or subset of those parameters, or by media access control (MAC) address, or other means. FQ-CoDel hashes flows into a configurable number of buckets to assign packets to internal Queues.

**Queue:** A queue of packets represented internally in FQ-CoDel. In most instances each flow gets its own queue; however because of the possibility of hash collisions, this is not always the case. In an attempt to avoid confusion, the word 'queue' is used to refer to the internal data structure, and 'flow' to refer to the actual stream of packets being delivered to the FQ-CoDel algorithm.

**Scheduler:** A mechanism to select which queue a packet is dequeued from.

**CoDel AQM:** The Active Queue Management algorithm employed by FQ-CoDel [[I-D.ietf-aqm-codel](#)].

**DRR:** Deficit round-robin scheduling [[DRR](#)].

**Quantum:** The maximum amount of bytes to be dequeued from a queue at once.

**Interval:** Characteristic time period used by the control loop of CoDel to detect when a persistent Queue is developing (see Section 4.3 of [[I-D.ietf-aqm-codel](#)]).

**Target:** Setpoint value of the minimum sojourn time of packets in a Queue used as the target of the control loop in CoDel (see Section 4.4 of [[I-D.ietf-aqm-codel](#)]).



### **1.3. Informal summary of FQ-CoDel**

FQ-CoDel is a `_hybrid_` of DRR [[DRR](#)] and CoDel [[I-D.ietf-aqm-codel](#)], with an optimisation for sparse flows similar to Shortest Queue First (SQF) [[SQF](#)] and DRR++ [[DRRPP](#)]. We call this "Flow Queueing" rather than "Fair Queueing" as flows that build a queue are treated differently from flows that do not.

By default, FQ-CoDel stochastically classifies incoming packets into different queues by hashing the 5-tuple of IP protocol number and source and destination IP and port numbers, perturbed with a random number selected at initiation time (although other flow classification schemes can optionally be configured instead; see [Section 4.1.1](#)). Each queue is managed by the CoDel AQM algorithm [[CODEL](#)]. Packet ordering within a queue is preserved, since queues have FIFO ordering.

The FQ-CoDel algorithm consists of two logical parts: the scheduler which selects which queue to dequeue a packet from, and the CoDel AQM which works on each of the queues. The subtleties of FQ-CoDel are mostly in the scheduling part, whereas the interaction between the scheduler and the CoDel algorithm are fairly straight forward:

At initialisation, each queue is set up to have a separate set of CoDel state variables. By default, 1024 queues are created. The Linux implementation at the time of writing supports anywhere from one to 64K separate queues, and each queue maintains the state variables throughout its lifetime, and so acts the same as the non-FQ CoDel variant would. This means that with only one queue, FQ-CoDel behaves essentially the same as CoDel by itself.

On dequeue, FQ-CoDel selects a queue from which to dequeue by a two-tier round-robin scheme, in which each queue is allowed to dequeue up to a configurable quantum of bytes for each iteration. Deviations from this quantum is maintained as byte credits for the queue, which serves to make the fairness scheme byte-based rather than packet-based. The two-tier round-robin mechanism distinguishes between "new" queues (which don't build up a standing queue) and "old" queues, that have queued enough data to be around for more than one iteration of the round-robin scheduler.

This new/old queue distinction has a particular consequence for queues that don't build up more than a quantum of bytes before being visited by the scheduler: Such queues are removed from the list, and then re-added as a new queue each time a packet arrives for it, and so will get priority over queues that do not empty out each round (except for a minor modification to protect against starvation, detailed below). Exactly how little data a flow has to send to keep





its queue in this state is somewhat difficult to reason about, because it depends on both the egress link speed and the number of concurrent flows. However, in practice many things that are beneficial to have prioritised for typical internet use (ACKs, DNS lookups, interactive SSH, HTTP requests, VoIP) *tend* to fall in this category, which is why FQ-CoDel performs so well for many practical applications. However, the implicitness of the prioritisation means that for applications that require guaranteed priority (for instance multiplexing the network control plane over the network itself), explicit classification is still needed.

This scheduling scheme has some subtlety to it, which is explained in detail in the remainder of this document.

## 2. CoDel

CoDel is described in the ACM Queue paper [[CODEL](#)], and the IETF document [[I-D.ietf-aqm-codel](#)]. The basic idea is to control queue length, maintaining sufficient queueing to keep the outgoing link busy, but avoiding building up the queue beyond that point. This is done by preferentially dropping packets that remain in the queue for "too long". Packets are dropped by head drop, which lowers the time for the drop signal to propagate back to the sender by the length of the queue, and helps trigger TCP fast retransmit sooner.

The CoDel algorithm itself will not be described here; instead we refer the reader to the CoDel draft [[I-D.ietf-aqm-codel](#)].

## 3. Flow Queueing

The intention of FQ-CoDel's scheduler is to give each *flow* its own queue, hence the term *Flow Queueing*. Rather than a perfect realisation of this, a hashing-based scheme is used, where flows are hashed into a number of buckets which each has its own queue. The number of buckets is configurable, and presently defaults to 1024 in the Linux implementation. This is enough to avoid hash collisions on a moderate number of flows as seen for instance in a home gateway. Depending on the characteristics of the link, this can be tuned to trade off memory for a lower probability of hash collisions. See [Section 6](#) for a more in-depth discussion of this.

By default, the flow hashing is performed on the 5-tuple of source and destination IP addresses and port numbers and IP protocol number. While the hashing can be customised to match on arbitrary packet bytes, care should be taken when doing so: Much of the benefit of the FQ-CoDel scheduler comes from this per-flow distinction. However, the default hashing does have some limitations, as discussed in [Section 6](#).



FQ-CoDel's DRR scheduler is byte-based, employing a deficit round-robin mechanism between queues. This works by keeping track of the current number `_byte_credits_` of each queue. This number is initialised to the configurable quantum; each time a queue gets a dequeue opportunity, it gets to dequeue packets, decreasing the number of credits by the packet size for each packet. This continues until the value of `_byte_credits_` becomes zero or less, at which point it is increased by one quantum, and the dequeue opportunity ends.

This means that if one queue contains packets of, for instance, size `quantum/3`, and another contains quantum-sized packets, the first queue will dequeue three packets each time it gets a turn, whereas the second only dequeues one. This means that flows that send small packets are not penalised by the difference in packet sizes; rather, the DRR scheme approximates a (single-)byte-based fairness queueing scheme. The size of the quantum determines the scheduling granularity, with the tradeoff from too small a quantum being scheduling overhead. For small bandwidths, lowering the quantum from the default MTU size can be advantageous.

Unlike plain DRR there are two sets of flows - a "new" list for flows that have not built a queue recently, and an "old" list for queues that build a backlog. This distinction is an integral part of the FQ-CoDel scheduler and is described in more detail in [Section 4](#).

#### 4. The FQ-CoDel scheduler

To make its scheduling decisions, FQ-CoDel maintains two ordered lists of active queues, called "new" and "old" queues. When a packet is added to a queue that is not currently active, that queue becomes active by being added to the list of new queues. Later on, it is moved to the list of old queues, from which it is removed when it is no longer active. This behaviour is the source of some subtlety in the packet scheduling at dequeue time, explained below.

##### 4.1. Enqueue

The packet enqueue mechanism consists of three stages: classification into a queue, timestamping and bookkeeping, and optionally dropping a packet when the total number of enqueued packets goes over the maximum.

When a packet is enqueued, it is first classified into the appropriate queue. By default, this is done by hashing (using a Jenkins hash function [[JENKINS](#)]) on the 5-tuple of IP protocol, and source and destination IP addresses and port numbers (if they exist), and taking the hash value modulo the number of queues. The hash is



salted by modulo addition of a random value selected at initialisation time, to prevent possible DoS attacks if the hash is predictable ahead of time (see [Section 8](#)). The Linux kernel implements the Jenkins hash function by mixing three 32-bit values into a single 32-bit output value. Inputs larger than 96 bits are reduced by additional mixing steps, 96 bits at a time.

Once the packet has been successfully classified into a queue, it is handed over to the CoDel algorithm for timestamping. It is then added to the tail of the selected queue, and the queue's byte count is updated by the packet size. Then, if the queue is not currently active (i.e., if it is not in either the list of new or the list of old queues), it is added to the end of the list of new queues, and its number of credits is initiated to the configured quantum. Otherwise, the queue is left in its current queue list.

Finally, the total number of enqueued packets is compared with the configured limit, and if it is `_above_` this value (which can happen since a packet was just enqueued), a packet is dropped from the head of the queue with the largest current byte count. Note that this in most cases means that the packet that gets dropped is different from the one that was just enqueued, and may even be from a different queue.

#### **4.1.1. Alternative classification schemes**

As mentioned previously, it is possible to modify the classification scheme to provide a different notion of a 'flow'. The Linux implementation provides this option in the form of the "tc filter" command. While this can add capabilities (for instance, matching on other possible parameters such as MAC address, diffserv code point values, firewall rules, flow specific markings, IPv6 flow label, etc.), care should be taken to preserve the notion of 'flow' as much of the benefit of the FQ-CoDel scheduler comes from keeping flows in separate queues.

For protocols that do not contain a port number (such as ICMP), the Linux implementation simply sets the port numbers to zero and performs the hashing as usual. In practice, this results in such protocols to each get their own queue (except in the case of hash collisions). An implementation can perform other classifications for protocols that have their own notion of a flow, but SHOULD fall back to simply hashing on source and destination IP address and IP protocol number in the absence of other information.

The default classification scheme can additionally be improved by performing decapsulation of tunnelled packets prior to hashing on the 5-tuple in the encapsulated payload. The Linux implementation does



this for common encapsulations known to the kernel, such as 6in4 [[RFC4213](#)], IP-in-IP [[RFC2003](#)] and GRE (Generic Routing Encapsulation) [[RFC2890](#)]. This helps to distinguish between flows that share the same (outer) 5-tuple, but of course is limited to unencrypted tunnels (see [Section 6.2](#)).

#### 4.2. Dequeue

Most of FQ-CoDel's work is done at packet dequeue time. It consists of three parts: selecting a queue from which to dequeue a packet, actually dequeuing it (employing the CoDel algorithm in the process), and some final bookkeeping.

For the first part, the scheduler first looks at the list of new queues; for the queue at the head of that list, if that queue has a negative number of credits (i.e., it has already dequeued at least a quantum of bytes), it is given an additional quantum of credits, the queue is put onto the end of the list of old queues, and the routine selects the next queue and starts again.

Otherwise, that queue is selected for dequeue. If the list of new queues is empty, the scheduler proceeds down the list of old queues in the same fashion (checking the credits, and either selecting the queue for dequeuing, or adding credits and putting the queue back at the end of the list).

After having selected a queue from which to dequeue a packet, the CoDel algorithm is invoked on that queue. This applies the CoDel control law, which is the mechanism CoDel uses to determine when to drop packets (see [[I-D.ietf-agm-codel](#)]). As a result of this, one or more packets may be discarded from the head of the selected queue, before the packet that should be dequeued is returned (or nothing is returned if the queue is or becomes empty while being handled by the CoDel algorithm).

Finally, if the CoDel algorithm does not return a packet, then the queue must be empty, and the scheduler does one of two things: if the queue selected for dequeue came from the list of new queues, it is moved to the end of the list of old queues. If instead it came from the list of old queues, that queue is removed from the list, to be added back (as a new queue) the next time a packet arrives that hashes to that queue. Then (since no packet was available for dequeue), the whole dequeue process is restarted from the beginning.

If, instead, the scheduler did get a packet back from the CoDel algorithm, it subtracts the size of the packet from the byte credits for the selected queue and returns the packet as the result of the dequeue operation.





The step that moves an empty queue from the list of new queues to \_the end of\_ the list of old queues before it is removed is crucial to prevent starvation. Otherwise the queue could reappear (the next time a packet arrives for it) before the list of old queues is visited; this can go on indefinitely even with a small number of active flows, if the flow providing packets to the queue in question transmits at just the right rate. This is prevented by first moving the queue to \_the end of\_ the list of old queues, forcing a pass through that, and thus preventing starvation. Moving it to the end of the list, rather than the front, is crucial for this to work.

The resulting migration of queues between the different states is summarised in the following state diagram:

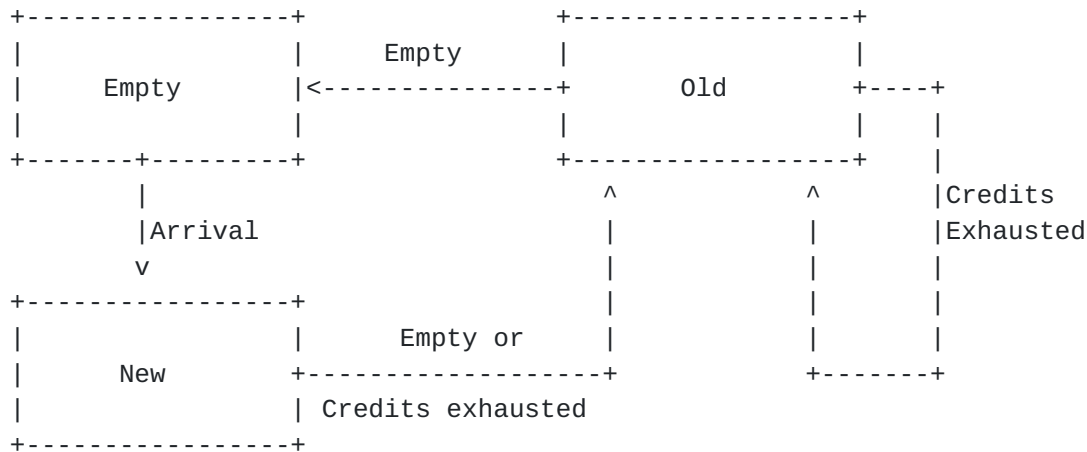


Figure 1: Partial state diagram for queues between different states. Both the new and old queue states can additionally have arrival and dequeue events that do not change the state; these are omitted here.

## 5. Implementation considerations

This section contains implementation details for the FQ-CoDel algorithm. This includes the data structures and parameters used in the Linux implementation, as well as discussion of some required features of the target platform and other considerations.

### 5.1. Data structures

The main data structure of FQ-CoDel is the array of queues, which is instantiated with the number of queues specified by the `_flows_` parameter at instantiation time. Each queue consists simply of an ordered list of packets with FIFO semantics, two state variables tracking the queue credits and total number of bytes enqueued, and the set of CoDel state variables. Other state variables to track



queue statistics can also be included: for instance, the Linux implementation keeps a count of dropped packets.

In addition to the queue structures themselves, FQ-CoDel maintains two ordered lists containing references to the subset of queues that are currently active. These are the list of 'new' queues and the list of 'old' queues, as explained in [Section 4](#) above.

In the Linux implementation, queue space is shared: there's a global limit on the number of packets the queues can hold, but not one per queue.

## **[5.2. Parameters](#)**

The following are the user configuration parameters exposed by the Linux implementation of FQ-CoDel.

### **[5.2.1. Interval](#)**

The `_interval_` parameter has the same semantics as CoDel and is used to ensure that the minimum sojourn time of packets in a queue used as an estimator by the CoDel control algorithm is a relatively up-to-date value. That is, CoDel only reacts to delay experienced in the last epoch of length `interval`. It SHOULD be set to be on the order of the worst-case RTT through the bottleneck to give end-points sufficient time to react.

The default interval value is 100 ms.

### **[5.2.2. Target](#)**

The `_target_` parameter has the same semantics as CoDel. It is the acceptable minimum standing/persistent queue delay for each FQ-CoDel Queue. This minimum delay is identified by tracking the local minimum queue delay that packets experience.

The default target value is 5 ms, but this value should be tuned to be at least the transmission time of a single MTU-sized packet at the prevalent egress link speed (which for, e.g., 1Mbps and MTU 1500 is ~15ms), to prevent CoDel from being too aggressive at low bandwidths. It should otherwise be set to on the order of 5-10% of the configured interval.

### **[5.2.3. Packet limit](#)**

Routers do not have infinite memory, so some packet limit MUST be enforced.



The `_limit_` parameter is the hard limit on the real queue size, measured in number of packets. This limit is a global limit on the number of packets in all queues; each individual queue does not have an upper limit. When the limit is reached and a new packet arrives for enqueue, a packet is dropped from the head of the largest queue (measured in bytes) to make room for the new packet.

In Linux, the default packet limit is 10240 packets, which is suitable for up to 10 Gigabit Ethernet speeds. In practice, the hard limit is rarely, if ever, hit, as drops are performed by the CoDel algorithm long before the limit is hit. For platforms that are severely memory constrained, a lower limit can be used.

#### **5.2.4. Quantum**

The `_quantum_` parameter is the number of bytes each queue gets to dequeue on each round of the scheduling algorithm. The default is set to 1514 bytes which corresponds to the Ethernet MTU plus the hardware header length of 14 bytes.

In systems employing TCP Segmentation Offload (TSO), where a "packet" consists of an offloaded packet train, it can presently be as large as 64K bytes. In systems using Generic Receive Offload (GRO), they can be up to 17 times the TCP max segment size (or 25K bytes). These mega-packets severely impact FQ-CoDel's ability to schedule traffic, and hurt latency needlessly. There is ongoing work in Linux to make smarter use of offload engines.

#### **5.2.5. Flows**

The `_flows_` parameter sets the number of queues into which the incoming packets are classified. Due to the stochastic nature of hashing, multiple flows may end up being hashed into the same slot.

This parameter can be set only at initialisation time in the current implementation, since memory has to be allocated for the hash table.

The default value is 1024 in the current Linux implementation.

#### **5.2.6. Explicit Congestion Notification (ECN)**

ECN is `_enabled_` by default. Rather than do anything special with misbehaved ECN flows, FQ-CoDel relies on the packet scheduling system to minimise their impact, thus the number of unresponsive packets in a flow being marked with ECN can grow to the overall packet limit, but will not otherwise affect the performance of the system.

It can be disabled by specifying the `_noecn_` parameter.



### **5.2.7. CE threshold**

This parameter enables Date Centre TCP (DCTCP)-like processing resulting in CE (Congestion Encountered) marking on ECN-Capable Transport (ECT) packets [[RFC3168](#)] starting at a lower sojourn delay setpoint than the default CoDel Target. Details of DCTCP can be found in [[I-D.ietf-tcpm-dctcp](#)].

The parameter, `_ce_threshold_`, is disabled by default and can be set to a number of microseconds to enable.

### **5.3. Probability of hash collisions**

Since the Linux FQ-CoDel implementation by default uses 1024 hash buckets, the probability that (say) 100 flows will all hash to the same bucket is something like ten to the power of minus 300. Thus, at least one of the flows will almost certainly hash to some other queue.

Expanding on this, based on analytical equations for hash collision probabilities, for 100 flows, the probability of no collision is 90.78%; the probability that no more than two of the 100 flows will be involved in any given collision = 99.57%; and the probability that no more than three of the 100 flows will be involved in any given collision = 99.99%. These probabilities assume a hypothetical perfect hashing function, so in practice they may be a bit lower. We have not found this difference to matter in practice.

These probabilities can be improved upon by using set-associative hashing, a technique used in the Cake algorithm currently being developed as a further development upon the FQ-CoDel principles. For a 4-way associative hash with the same number of total queues, the probability of no collisions for 100 flows is 99.93%, while for an 8-way associative hash it is ~100%.

### **5.4. Memory Overhead**

FQ-CoDel can be implemented with a low memory footprint (less than 64 bytes per queue on 64 bit systems). These are the data structures used in the Linux implementation:

<CODE BEGINS>





```

struct codel_vars {
    u32          count;          /* number of dropped packets */
    u32          lastcount;     /* count entry to dropping state */
    bool         dropping;      /* currently dropping? */
    u16          rec_inv_sqrt;   /* reciprocal sqrt computation */
    codel_time_t first_above_time; /* when delay above target */
    codel_time_t drop_next;     /* next time to drop */
    codel_time_t ldelay; /* sojourn time of last dequeued packet */
};

struct fq_codel_flow {
    struct sk_buff *head;
    struct sk_buff *tail;
    struct list_head flowchain;
    int credits; /* current number of queue credits */
    u32 dropped; /* # of drops (or ECN marks) on flow */
    struct codel_vars cvars;
};

```

<CODE ENDS>

The master table managing all queues looks like this:

<CODE BEGINS>

```

struct fq_codel_sched_data {
    struct tcf_proto *filter_list; /* optional external classifier */
    struct fq_codel_flow *flows; /* Flows table [flows_cnt] */
    u32 *backlogs; /* backlog table [flows_cnt] */
    u32 flows_cnt; /* number of flows */
    u32 perturbation; /* hash perturbation */
    u32 quantum; /* psched_mtu(qdisc_dev(sch)); */
    struct codel_params cparams;
    struct codel_stats cstats;
    u32 drop_overlimit;
    u32 new_flow_count;

    struct list_head new_flows; /* list of new flows */
    struct list_head old_flows; /* list of old flows */
};

```

<CODE ENDS>

## 5.5. Per-Packet Timestamping

The CoDel portion of the algorithm requires per-packet timestamps be stored along with the packet. While this approach works well for software-based routers, it may be impossible to retrofit devices that



do most of their processing in silicon and lack space or mechanism for timestamping.

Also, while perfect resolution is not needed, timestamp resolution finer than the CoDel target setting is necessary. Furthermore, timestamping functions in the core OS need to be efficient as they are called at least once on each packet enqueue and dequeue.

### **5.6. Limiting queueing in lower layers**

When deploying a queue management algorithm such as FQ-CoDel, it is important to ensure that the algorithm actually runs in the right place to control the queue. In particular lower layers of the operating system networking stack can have queues of their own, as can device drivers and hardware. Thus, it is desirable that the queue management algorithm runs as close to the hardware as possible. However, scheduling such complexity at interrupt time is difficult, so a small standing queue between the algorithm and the wire is often needed at higher transmit rates.

In Linux, the mechanism to ensure these different needs are balanced is called "Byte Queue Limits" [[BQL](#)], which controls the device driver ring buffer (for physical line rates). For cases where this functionality is not available, the queue can be controlled by means of a software rate limiter such as Hierarchical Token Bucket [[HTB](#)] or Hierarchical Fair-Service Curve [[HFSC](#)]. The Cake algorithm [[CAKE](#)] integrates a software rate limiter for this purpose.

Other issues with queues at lower layers are described in [[CODEL](#)].

### **5.7. Other forms of "Fair Queueing"**

Much of the scheduling portion of FQ-CoDel is derived from DRR and is substantially similar to DRR++. Versions based on Stochastic Fair Queueing [[SFQ](#)] have also been produced and tested in ns2. Other forms of Fair Queueing, such as Weighted Fair Queueing [[WFQ](#)] or Quick Fair Queueing [[QFQ](#)], have not been thoroughly explored, but there's no a priori reason why the round-robin scheduling of FQ-CoDel couldn't be replaced with something else.

For a comprehensive discussion of fairness queueing algorithms and their combination with AQM, see [[I-D.ietf-aqm-fq-implementation](#)].

### **5.8. Differences between CoDel and FQ-CoDel behaviour**

CoDel can be applied to a single queue system as a straight AQM, where it converges towards an "ideal" drop rate (i.e., one that



minimises delay while keeping a high link utilisation), and then optimises around that control point.

The scheduling of FQ-CoDel mixes packets of competing flows, which acts to pace bursty flows to better fill the pipe. Additionally, a new flow gets substantial leeway over other flows until CoDel finds an ideal drop rate for it. However, for a new flow that exceeds the configured quantum, more time passes before all of its data is delivered (as packets from it, too, are mixed across the other existing queue-building flows). Thus, FQ-CoDel takes longer (as measured in time) to converge towards an ideal drop rate for a given new flow, but does so within fewer delivered `_packets_` from that flow.

Finally, the flow isolation FQ-CoDel provides means that the CoDel drop mechanism operates on the flows actually building queues, which results in packets being dropped more accurately from the largest flows than CoDel alone manages. Additionally, flow isolation radically improves the transient behaviour of the network when traffic or link characteristics change (e.g., when new flows start up or the link bandwidth changes); while CoDel itself can take a while to respond, FQ-CoDel reacts almost immediately.

## **6. Limitations of flow queueing**

While FQ-CoDel has been shown in many scenarios to offer significant performance gains compared to alternative queue management strategies, there are some scenarios where the scheduling algorithm in particular is not a good fit. This section documents some of the known cases which either may require tweaking the default behaviour, or where alternatives to flow queueing should be considered.

### **6.1. Fairness between things other than flows**

In some parts of the network, enforcing flow-level fairness may not be desirable, or some other form of fairness may be more important. An example of this can be an Internet Service Provider that may be more interested in ensuring fairness between customers than between flows. Or a hosting or transit provider that wishes to ensure fairness between connecting Autonomous Systems or networks. Another issue can be that the number of simultaneous flows experienced at a particular link can be too high for flow-based fairness queueing to be effective.

Whatever the reason, in a scenario where fairness between flows is not desirable, reconfiguring FQ-CoDel to match on a different characteristic can be a way forward. The implementation in Linux can leverage the packet matching mechanism of the `_tc_` subsystem to use



any available packet field to partition packets into virtual queues, to for instance match on address or subnet source/destination pairs, application layer characteristics, etc.

Furthermore, as commonly deployed today, FQ-CoDel is used with three or more tiers of service classification: priority, best effort and background, based on diffserv markings. Some products do more detailed classification, including deep packet inspection and destination-specific filters to achieve their desired result.

## **6.2. Flow bunching by opaque encapsulation**

Where possible, FQ-CoDel will attempt to decapsulate packets before matching on the header fields for the flow hashing. However, for some encapsulation techniques, most notably encrypted VPNs, this is not possible. If several flows are bunched into one such encapsulated tunnel, they will be seen as one flow by the FQ-CoDel algorithm. This means that they will share a queue, and drop behaviour, and so flows inside the encapsulation will not benefit from the implicit prioritisation of FQ-CoDel, but will continue to benefit from the reduced overall queue length from the CoDel algorithm operating on the queue. In addition, when such an encapsulated bunch competes against other flows, it will count as one flow, and not assigned a share of the bandwidth based on how many flows are inside the encapsulation.

Depending on the application, this may or may not be desirable behaviour. In cases where it is not, changing FQ-CoDel's matching to not be flow-based (as detailed in the previous subsection above) can be a mitigation. Going forward, having some mechanism for opaque encapsulations to express to the outer layer which flow a packet belongs to, could be a way to mitigate this. Naturally, care needs to be taken when designing such a mechanism to ensure no new privacy and security issues are raised by exposing information from inside the encapsulation to the outside world. Keeping the extra information out-of-band and dropping it before it hits the network could be one way to achieve this.

## **6.3. Low-priority congestion control algorithms**

In the presence of queue management schemes that limit latency under load, low-priority congestion control algorithms such as LEDBAT [[RFC6817](#)] (or, in general, algorithms that try to voluntarily use up less than their fair share of bandwidth) experiences little added latency when the link is congested. Thus, they lack the signal to back off that added latency previously afforded them. This effect is seen with FQ-CoDel as well as with any effective AQM [[GONG2014](#)].





As such, these delay-based algorithms tend to revert to loss-based congestion control, and will consume the fair share of bandwidth afforded to them by the FQ-CoDel scheduler. However, low-priority congestion control mechanisms may be able to take steps to continue to be low priority, for instance by taking into account the vastly reduced level of delay afforded by an AQM, or by using a coupled approach to observing the behaviour of multiple flows.

## 7. Deployment status and future work

The FQ-CoDel algorithm as described in this document has been shipped as part of the Linux kernel since version 3.5, released on the 21st of July, 2012, with the `ce_threshold` being added in version 4.2. The algorithm has seen widespread testing in a variety of contexts and is configured as the default queueing discipline in a number of mainline Linux distributions (as of this writing at least OpenWRT, Arch Linux and Fedora). We believe it to be a safe default and encourage people running Linux to turn it on: It is a massive improvement over the previous default FIFO queue.

Of course there is always room for improvement, and this document has listed some of the known limitations of the algorithm. As such, we encourage further research into algorithm refinements and addressing of limitations. One such effort is undertaken by the `bufferbloat` community in the form of the Cake queue management scheme [[CAKE](#)]. In addition to this we believe the following (non-exhaustive) list of issues to be worthy of further enquiry:

- o Variations on the flow classification mechanism to fit different notions of flows. For instance, an ISP might want to deploy per-subscriber scheduling, while in other cases several flows can share a 5-tuple, as exemplified by the RTCWEB QoS recommendations [[I-D.ietf-tsvwg-rtcweb-qos](#)].
- o Interactions between flow queueing and delay-based congestion control algorithms and scavenger protocols.
- o Other scheduling mechanisms to replace the DRR portion of the algorithm, e.g., QFQ or WFQ.
- o Sensitivity of parameters, most notably the number of queues and the CoDel parameters.

## 8. Security Considerations

There are no specific security exposures associated with FQ-CoDel that are not also present in current FIFO systems. On the contrary, some vulnerabilities of FIFO systems are reduced with FQ-CoDel (e.g.,



simple minded packet floods). However, some care is needed in the implementation to ensure this is the case. These are included in the description above, however we reiterate them here:

- o To prevent packets in the new queues from starving old queues, it is important that when a queue on the list of new queues empties, it is moved to the end of the list of old queues. This is described at the end of [Section 4.2](#).
- o To prevent an attacker targeting a specific flow for a denial of service attack, the hash that maps packets to queues should not be predictable. To achieve this, FQ-CoDel salts the hash, as described in the beginning of [Section 4.1](#). The size of the salt and the strength of the hash function is obviously a tradeoff between performance and security. The Linux implementation uses a 32 bit random value as the salt and a Jenkins hash function. This makes it possible to achieve high throughput, and we consider it sufficient to ward off the most obvious attacks.
- o Packet fragments without a layer 4 header can be hashed into different bins than the first fragment with the header intact. This can cause reordering and/or adversely affect the performance of the flow. Keeping state to match the fragments to the beginning of the packet, or simply putting all packet fragments (including the first fragment of each fragmented packet) into the same queue, are two ways to alleviate this.

## **[9.](#) IANA Considerations**

This document has no actions for IANA.

## **[10.](#) Acknowledgements**

Our deepest thanks to Kathie Nichols, Van Jacobson, and all the members of the [bufferbloat.net](#) effort for all the help on developing and testing the algorithm. In addition, our thanks to Anil Agarwal for his help with getting the hash collision probabilities in this document right.

## **[11.](#) References**

### **[11.1.](#) Normative References**

[I-D.ietf-aqm-codel]

Nichols, K., Jacobson, V., McGregor, A., and J. Iyengar, "Controlled Delay Active Queue Management", [draft-ietf-aqm-codel-03](#) (work in progress), March 2016.



## [I-D.ietf-aqm-fq-implementation]

Baker, F. and R. Pan, "On Queuing, Marking, and Dropping", [draft-ietf-aqm-fq-implementation-05](#) (work in progress), November 2015.

## [I-D.ietf-tcpm-dctcp]

Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and G. Judd, "Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters", [draft-ietf-tcpm-dctcp-01](#) (work in progress), November 2015.

## [I-D.ietf-tsvwg-rtcweb-qos]

Jones, P., Dhesikan, S., Jennings, C., and D. Druta, "DSCP and other packet markings for WebRTC QoS", [draft-ietf-tsvwg-rtcweb-qos-14](#) (work in progress), March 2016.

[RFC2003] Perkins, C., "IP Encapsulation within IP", [RFC 2003](#), DOI 10.17487/RFC2003, October 1996, <<http://www.rfc-editor.org/info/rfc2003>>.

[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>>.

[RFC2890] Dommety, G., "Key and Sequence Number Extensions to GRE", [RFC 2890](#), DOI 10.17487/RFC2890, September 2000, <<http://www.rfc-editor.org/info/rfc2890>>.

[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>>.

[RFC4213] Nordmark, E. and R. Gilligan, "Basic Transition Mechanisms for IPv6 Hosts and Routers", [RFC 4213](#), DOI 10.17487/RFC4213, October 2005, <<http://www.rfc-editor.org/info/rfc4213>>.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", [RFC 6817](#), DOI 10.17487/RFC6817, December 2012, <<http://www.rfc-editor.org/info/rfc6817>>.



## **11.2. Informative References**

- [BLOAT] Gettys, J., "Bufferbloat: Dark buffers in the Internet.", in IEEE Internet Comput. 15, 3, DOI <http://dx.doi.org/10.1109/MIC.2011.56>, 2011, <<http://www.bufferbloat.net/attachments/27/IC-15-03-Backspace.pdf>>.
- [BLOATWEB] "Bufferbloat web site", <<https://www.bufferbloat.net>>.
- [BQL] Herbert, T., "Network Byte Queue Limits", August 2011, <<https://lwn.net/Articles/454390/>>.
- [CAKE] "Cake comprehensive queue management system", <<http://www.bufferbloat.net/projects/codel/wiki/Cake>>.
- [CODEL] Nichols, K. and V. Jacobson, "Controlling Queue Delay", July 2012, <<http://queue.acm.org/detail.cfm?id=2209336>>.
- [DRR] Shreedhar, M. and G. Varghese, "Efficient Fair Queueing Using Deficit Round Robin", in IEEE/ACM Trans. Netw. 4, 3, June 1996, <<http://users.ece.gatech.edu/~siva/ECE4607/presentations/DRR.pdf>>.
- [DRRPP] MacGregor, M. and W. Shi, "Deficits for Bursty Latency-critical Flows: DRR++", in Proceedings IEEE International Conference on Networks 2000 (ICON 2000), 2000, <[http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=875803](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=875803)>.
- [GONG2014] Gong, Y., Rossi, D., Testa, C., Valenti, S., and D. Taht, "Fighting the bufferbloat: on the coexistence of AQM and low priority congestion control", in 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPs), July 2014, <<http://perso.telecom-paristech.fr/~drossi/paper/rossi14comnet-b.pdf>>.
- [HFSC] Stoica, I., Zhang, H., and T. Eugene, "Hierarchical fair-service curve", in Sigcomm 1997 proceedings, 1997, <<http://conferences.sigcomm.org/sigcomm/1997/papers/p011.pdf>>.
- [HTB] "Hierarchical Token Bucket", <[https://en.wikipedia.org/wiki/Token\\_bucket#Variations](https://en.wikipedia.org/wiki/Token_bucket#Variations)>.





- [JENKINS] Jenkins, B., "A Hash Function for Hash Table Lookup", 1996, <<http://www.burtleburtle.net/bob/hash/doobs.html>>.
- [LINUXSRC] "Current FQ-CoDel Linux source code", <[https://git.kernel.org/cgiit/linux/kernel/git/torvalds/linux.git/tree/net/sched/sch\\_fq\\_codel.c](https://git.kernel.org/cgiit/linux/kernel/git/torvalds/linux.git/tree/net/sched/sch_fq_codel.c)>.
- [NS2] "NS2 web site", <<http://nslam.sourceforge.net/wiki>>.
- [NS3] "NS3 web site", <<https://www.nslam.org/wiki>>.
- [QFQ] Checconi, F., Rizzo, L., and P. Valente, "QFQ: Efficient packet scheduling with tight guarantees", in IEEE/ACM Transactions on Networking (TON), 2013, <<http://dl.acm.org/citation.cfm?id=2525552>>.
- [SFQ] McKenney, P., "Stochastic fairness queueing", published as technical report, 2002, <<https://web.archive.org/web/20151003174154/http://www2.rdrop.com/~paulmck/scalability/paper/sfq.2002.06.04.pdf>>.
- [SQF] Bonald, T., Muscariello, L., and N. Ostallo, "On the impact of TCP and per-flow scheduling on Internet Performance", in IEEE/ACM transactions on Networking, April 2012, <[http://perso.telecom-paristech.fr/~bonald/Publications\\_files/BM02011.pdf](http://perso.telecom-paristech.fr/~bonald/Publications_files/BM02011.pdf)>.
- [WFQ] Demers, A., Keshav, S., and S. Shenker, "Analysis and simulation of a fair queueing algorithm", in SIGCOMM Comput. Commun. Rev., September 1989, <<http://doi.acm.org/10.1145/75247.75248>>.

#### Authors' Addresses

Toke Hoiland-Joergensen  
Karlstad University  
Dept. of Computer Science  
Karlstad 65188  
Sweden

Email: [toke.hoiland-jorgensen@kau.se](mailto:toke.hoiland-jorgensen@kau.se)



Paul McKenney  
IBM Linux Technology Center  
1385 NW Amberglen Parkway  
Hillsboro, OR 97006  
USA

Email: paulmck@linux.vnet.ibm.com  
URI: <http://www2.rdrop.com/~paulmck/>

Dave Taht  
Teklibre  
2104 W First street  
Apt 2002  
FT Myers, FL 33901  
USA

Email: dave.taht@gmail.com  
URI: <http://www.teklibre.com/>

Jim Gettys  
21 Oak Knoll Road  
Carlisle, MA 993  
USA

Email: jg@freedesktop.org  
URI: [https://en.wikipedia.org/wiki/Jim\\_Gettys](https://en.wikipedia.org/wiki/Jim_Gettys)

Eric Dumazet  
Google, Inc.  
1600 Amphitheater Pkwy  
Mountain View, CA 94043  
USA

Email: edumazet@gmail.com

