AQM working group
Internet-Draft
Intended status: Informational
Expires: April 21, 2016

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

E. Dumazet
Google, Inc.
October 19, 2015

FlowQueue-Codel
draft-ietf-aqm-fq-codel-02

Abstract

This memo presents the FQ-CoDel hybrid packet scheduler/AQM 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

Copyright Notice

Table of Contents

1. Introduction

The FQ-CoDel algorithm is a combined packet scheduler and AQM developed as part of the bufferbloat-fighting community effort. It is based on a modified Deficit Round Robin (DRR) queue scheduler, with the CoDel AQM algorithm operating on each queue. This document describes the combined algorithm; reference implementations are available for ns2 and ns3 and it is included in the mainline Linux kernel as the fq_codel queueing discipline.

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 defines the parameters and data structures employed by FQ-CoDel. Section 5 describes the working of the algorithm in detail. Section 6 describes implementation considerations, and Section 7 lists some of the limitations of using flow queueing. Section 8 outlines the current status of FQ-CoDel deployment and lists some possible future areas of inquiry, and finally, Section 12 concludes.

1.1. Terminology and concepts

Flow: A flow is typically identified by a 5-tuple of source IP, destination IP, source port, destination port, and protocol. It can also be identified by a superset or subset of those parameters, or by mac address, or other means.

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.

DRR: Deficit round-robin scheduling.

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

1.2. Informal summary of FQ-CoDel

FQ-CoDel is a _hybrid_ of DRR [DRR] and CoDel [CODELDRAFT], with an optimisation for sparse flows similar to SQF [SQF2012] and DRR++ [DRRPP]. We call this "Flow Queueing" rather than "Fair Queueing" as flows that build a queue are treated differently than flows that do not.

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). Each queue is managed by the CoDel AQM algorithm. 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 current implementation 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 a deficit for the queue, which serves to make the fairness scheme byte-based rather than a 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, ARP, RA, ICMP, 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 the ACM Queue paper [CODEL2012], and the AQM working group draft [CODELDRAFT]. 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 [CODELDRAFT].

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 are 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 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 7.

FQ-CoDel's DRR scheduler is byte-based, employing a deficit round-robin mechanism between queues. This works by keeping track of the current byte _deficit_ of each queue. This deficit is initialised to the configurable quantum; each time a queue gets a dequeue opportunity, it gets to dequeue packets, decreasing the deficit by the packet size for each packet, until the deficit runs into the negative, 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. 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 flow-building queues. This distinction is an integral part of the FQ-CoDel scheduler and is described in more detail in Section 5.

4. FQ-CoDel Parameters and Data Structures

This section goes into the parameters and data structures in FQ-CoDel.

4.1. Parameters

4.1.1. Interval

The _interval_ parameter has the same semantics as CoDel and is used to ensure that the measured minimum delay does not become too stale. The minimum delay MUST be experienced in the last epoch of length

interval. It SHOULD be set 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.

4.1.2. 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.

4.1.3. 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 10GigE 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.

4.1.4. 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 TSO-enabled systems, where a "packet" consists of an offloaded packet train, it can presently be as large as 64K bytes. In GRO-enabled systems, 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.

4.1.5. 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 load time since memory has to be allocated for the hash table in the current implementation.

The default value is 1024 in the current Linux implementation.

4.1.6. 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 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.

4.1.7. CE threshold

This parameter enables DCTCP-like processing to enable CE marking ECT packets at a lower setpoint than the default codel target.

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

4.2. Data structures

4.2.1. Internal queues

The main data structure of FQ-CoDel is the array of queues, which is instantiated to 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 deficit 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.

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

4.2.2. New and old queues lists

FQ-CoDel maintains two lists of active queues, called "new" and "old" queues. Each list is an ordered list containing references to the array of 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.

5. The FQ-CoDel scheduler

This section describes the operation of the FQ-CoDel scheduler and AQM. It is split into two parts explaining the enqueue and dequeue operations.

5.1. 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) on the 5-tuple of IP protocol, and source and destination IP and port numbers, permuted by a random value selected at initialisation time, and taking the hash value modulo the number of queues.

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 deficit is initiated to the configured quantum. Otherwise it is added to the old 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.

5.1.1. 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, firewall and flow specific markings, 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. We are not aware of any deployments utilising the custom classification feature.

An alternative to changing the classification scheme is to perform decapsulation prior to hashing. The Linux implementation does this for common encapsulations known to the kernel, such as 6in4, IPIP and GRE tunnels. This helps to distinguish between flows that share the same (outer) 5-tuple, but of course is limited to unencrypted tunnels (see Section 7.2).

5.2. 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 each queue in that list, if that queue has a negative deficit (i.e. it has already dequeued at least a quantum of bytes), its deficit is increased by one quantum, and 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 deficit, and either selecting the queue for dequeuing, or increasing the deficit 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, and may discard one or more packets from the head of that queue, before returning the packet that should be dequeued (or nothing if the queue is or becomes empty while being handled by the CoDel algorithm).

Finally, if the CoDel algorithm did not return a packet, the queue is 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 updates the byte deficit for the selected queue before returning 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:

```
   +-----------------+                +------------------+
   |                 |     Empty      |                  |
   |     Empty       |<---------------+       Old        +-----+
   |                 |                |                  |     |
   +-------+---------+                +------------------+     |
           |                             ^            ^        |Quantum
           |Arrival                      |            |        |Exceeded
           v                             |            |        |
   +-----------------+                   |            |        |
   |                 |     Empty or      |            |        |
   |      New        +-------------------+            +--------+
   |                 |  Quantum exceeded
   +-----------------+
```

6. Implementation considerations

6.1. Probability of hash collisions

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

Expanding on this, based on analytical equations for hash collision probabilities, for 100 VoIP sessions, the probability of no collision is 90.78%; the probability that no more than two of the 100 VoIP sessions will be involved in any given collision = 99.57%; and the probability that no more than three of the 100 VoIP sessions will be involved in any given collision = 99.99%.

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 VoIP sessions is 99.93%, while for an 8-way associative hash it is ~100%.

6.2. Memory Overhead

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

```
struct codel_vars {
   u32             count;
   u32             lastcount;
   bool            dropping;
   u16             rec_inv_sqrt;
   codel_time_t    first_above_time;
   codel_time_t    drop_next;
   codel_time_t    ldelay;
};

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

The master table managing all queues looks like this:

```
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 */
};
```

6.3. 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 below the target 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.

6.4. Other forms of "Fair Queueing"

Much of the scheduling portion of FQ-CoDel is derived from DRR and is substantially similar to DRR++. SFQ-based versions have also been produced and tested in ns2. Other forms of Fair Queueing, such as WFQ or QFQ, have not been thoroughly explored.

6.5. 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 "credit" 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 doesn't miss a beat.

7. Limitations of flow queueing

While FQ-CoDel has been shown in many scenarios to offer significant performance gains, 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.

7.1. 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 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.

7.2. 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.

7.3. Low-priority congestion control algorithms

In the presence of queue management schemes that contain 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 very 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.

8. 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 know 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 [1] queue management scheme. In addition to this we believe the following (non-exhaustive) list of issues to be worthy of further enquiry:

- 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 [RTCWEB-QOS].

- Interactions between flow queueing and delay-based congestion control algorithms and scavenger protocols.

- Other scheduling mechanisms to replace the DRR portion of the algorithm, e.g. QFQ or WFQ.

- Sensitivity of parameters, most notably the number of queues and the CoDel parameters.

9. Security Considerations

There are no specific security exposures associated with FQ-CoDel that are not also present in current FIFO systems. And some are in fact reduced (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:

- 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 5.2.

- 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 5.1. 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 very high throughput, and we consider it sufficient to ward off the most obvious attacks.

10. IANA Considerations

This document has no actions for IANA.

11. 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.

12. Conclusions

FQ-CoDel is a very general, efficient, nearly parameterless queue management approach combining flow queueing with CoDel. It is a very powerful tool for solving bufferbloat, 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 have documented the Linux implementation in sufficient detail for an independent implementation, and we encourage such implementations be widely deployed.

13. References

13.1. Normative References

[CODELDRAFT]
Nichols, K., Jacobson, V., McGregor, A., and J. Iyengar, "Controlled Delay Active Queue Management", October 2014, <https://datatracker.ietf.org/doc/draft-ietf-aqm-codel/>.

[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>.

[RTCWEB-QOS]
Dhesikan, S., Jennings, C., Druta, D., Jones, P., and J. Polk, "DSCP and other packet markings for RTCWeb QoS", December 2014, <https://datatracker.ietf.org/doc/draft-dhesikan-tsvwg-rtcweb-qos/>.

13.2. Informative References

[CODEL2012]
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", 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++", 2000, <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", July 2014, <http://perso.telecom-paristech.fr/~drossi/paper/rossi14comnet-b.pdf>.

[SQF2012] Bonald, T., Muscariello, L., and N. Ostallo, "On the impact of TCP and per-flow scheduling on Internet Performance - IEEE/ACM transactions on Networking", April 2012, <http://perso.telecom-paristech.fr/~bonald/Publications_files/BMO2011.pdf>.

13.3. URIs

[1] http://www.bufferbloat.net/projects/codel/wiki/Cake

Authors' Addresses

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

Email: 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

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

Email: edumazet@gmail.com