Transport Working Group
Internet-Draft
Intended status: Informational
Expires: 4 January 2020

J. Morton
P. Heist

3 July 2019

Lightweight Fair Queueing
draft-morton-tsvwg-lightweight-fair-queueing-00

Abstract

This note presents Lightweight Fair Queueing (LFQ), a fair queueing algorithm with a small code footprint, low memory requirements, no multiply operations, only two physical queues, and only one set of AQM state. LFQ provides throughput fairness, sparse flow prioritization and ordering guarantees, making it suitable for a mixture of traffic flow types.

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 https://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 4 January 2020.

Table of Contents

1. Introduction

Flow isolation is a powerful tool for congestion management in today's Internet. Early implementations, such as SFQ [SFQ], aimed simply to have inter-flow induced latency dependent on the number of flows, rather than the total length of the queue. Today, DRR++ [DRRPP] explicitly shares throughput capacity between flows, and prioritises "sparse" flows that use less than their fair share, on the grounds that these are probably latency-sensitive traffic. This reduces the inter-flow induced latency to near zero for sparse flows, regardless of the number of saturating flows.

Unfortunately, the relatively complex algorithms and considerable dynamic state of a DRR++ queue set with individual AQM (Active Queue Management) [RFC7567] instances has proved disheartening to hardware implementors, and thus to deployment on high-capacity links. Ordinary CPE devices implementing DRR++ in software work well up to 100Mbps or so. A scheme involving only a small number of queues and AQM instances might be more suitable for the 1Gbps and up category.

This note therefore presents LFQ, a fair queueing algorithm suitable for implementation in hardware, making fair queueing possible on high-throughput routers and low-cost middleboxes.

2. Background

LFQ is inspired by DRR++'s facility for identifying "sparse" flows and giving them strict priority over "saturating" flows. DRR++ does this by maintaining separate lists of queues (each queue containing

one flow) meeting "sparseness" criteria or not.

Queues are first placed into the sparse list when they become non-empty, then moved to the saturating list when their deficit exceeds a threshold called "quantum". Every queue's deficit is incremented by

the packet size when packets are delivered from it, and decremented by the quantum when they come up in the list rotation. Queues are removed from the saturating list only when they are found empty for a full rotation.

This "sparseness" heuristic over observed per-flow queue occupancy characteristics is relatively robust, compared to relying on the correct behaviour of each source's congestion control algorithms and/or explicit traffic marking. This is especially relevant with the recent development of high-fidelity congestion signalling schemes, such as DCTCP [RFC8257] and SCE (Some Congestion Experienced), whose expected congestion-signal response is markedly different from previous standards.

In fq_codel [RFC8290] and Cake [CAKE], AQM is applied individually to each DRR++ flow, thus avoiding unnecessary leakage of AQM action from flows requiring it to well-behaved traffic which does not. This arrangement has been shown to work well in practice, and is widely deployed as part of the Linux kernel, including in many CPE devices. However the per-queue AQM state dominates the memory requirements of DRR++.

LFQ attempts to retain most of these characteristics while simplifying implementation requirements considerably. This still requires identifying individual traffic flows and keeping some per-flow state, but there is no longer an individual queue per state nor any lists of such queues. Instead there are only two queues and only one set of AQM state. The operations required are believed to be amenable to hardware implementation.

3. The Algorithm

3.1. Overview

Unlike conventional fair queueing, with Lightweight Fair Queueing, packets are not distributed to queues by a flow mapping, but by a

sparseness metric associated with that mapping. Thus, the number of queues is reduced to two.

The number of flows which can be handled is far greater, however, being limited by the number of flow buckets indexed by the flow hash. An implementation might define a flow as traffic to one subscriber, and provide a perfect mapping between subscribers and buckets. Alternatively it might provide a stochastic mapping based on the traditional 5-tuple of addresses, port numbers, and protocol number.

The per-flow state is just two integers (one signed, one unsigned) and one binary flag, in contrast to DRR++ which requires a whole

queue and a set of AQM state per flow. These integers are B, tracking the backlog of the flow in packets, and D, tracking a deficit value analogous to that used in DRR++. The range of D is [-MTU..+MTU], so for a typical 1500-byte MTU, a 12-bit register suffices. The binary flag K indicates whether packets should be skipped for the rest of this pass through the queue. This small per-flow state makes tracking a large number of flows practical.

The two queues provided are SQ and BQ:

SQ is the "sparse queue" which handles flows classed as sparse, including the first packets in newly active flows. This queue tends to remain short and drain quickly, which are ideal characteristics for latency-sensitive traffic, and young flows still establishing connections or probing for capacity. This queue does not maintain AQM state nor apply AQM signals.

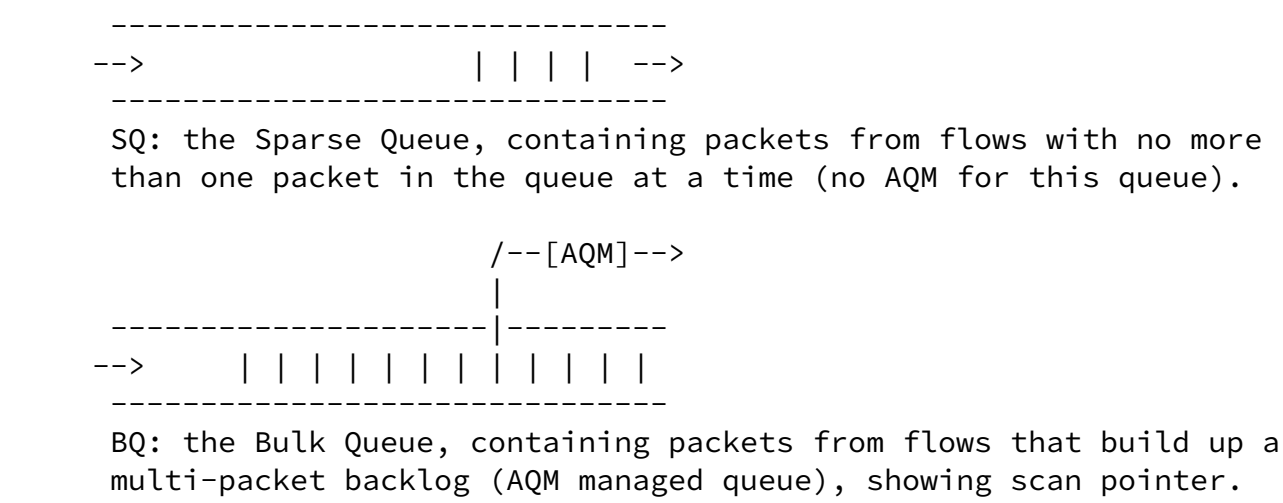
BQ is the "bulk queue" which handles all traffic not classed as sparse, including at least the second and subsequent packets in a burst. BQ has not only the typical "head" and "tail", but also a "scan" pointer which iterates over the packets in the queue from head to tail. Packets are delivered from the "scan" position, not from the "head"; this is key to the capacity-sharing mechanism. A full set of AQM state is maintained on BQ, and applied to all traffic delivered from it.

In case of queue overflow, packets are removed from the "head" of BQ to make room for the new arrivals; this head-dropping behaviour minimises the delay before the lost packets can be retransmitted.

This simplification of state and algorithm has some drawbacks in terms of resultant behaviour. The sharing of link capacity between flows will not be as smooth as with DRR++, and the relatively coarse provision of AQM may result in a noticeable degradation of congestion signalling.

3.2. Declarations

The following queues are defined:



The following constants and variables are defined:

* B: the flow backlog, in packets

* D: the flow deficit, in bytes

* K: the flow scan skip flag

* N: the number of flow buckets (each bucket containing a value of B, D, and K)

* S: the size of a packet

* T: the packet's timestamp, for later use by AQM

* H: the packet's flow hash, cached

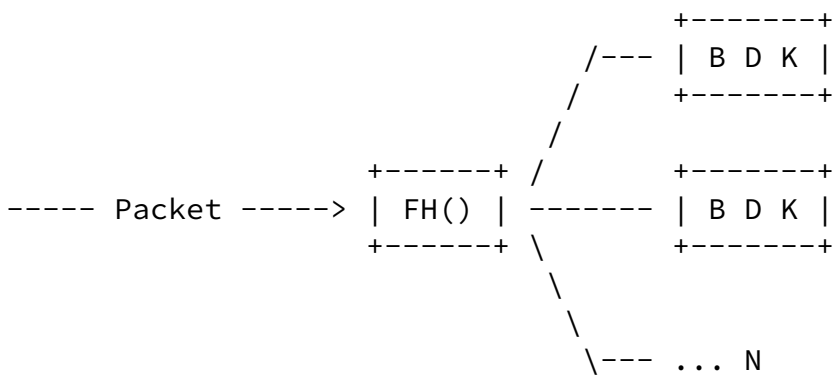
* MTU: the MTU of the link

* MAXSIZE: the maximum size for all packets in the queue

* NOW: the current timestamp

* FLOWS: all flow buckets

Finally, the hash function FH() maps a packet to a flow bucket:



3.3. Pseudo-code

In the following pseudo-code:

* Lowercase is used for internal variables, and uppercase for constants, variables and queues defined in Section 3.2.

* The send() function applies AQM logic before actually transmitting the packet given; if it drops the packet, dequeue() is immediately re-called.

The following functions and variables are defined for both the sparse and bulk queues:

* The push() function adds a packet to the tail of the specified queue.

* The pop() function removes and returns the packet from the head of the specified queue. BQ's scan pointer must point to the same packet afterwards, if it is still present, otherwise to the head of the queue.

* The .size variable (BQ.size and SQ.size) refers to the sum of the sizes of all packets in the queue, and may be maintained during push(), pop(), and pull().

* The .head variable is the current head pointer for the queue.

The following functions and variables are defined only for BQ:

* The pull() function removes and returns the packet at the scan pointer.

* The scan() function returns the packet at the scan pointer without removing it.

* The head() function returns the packet at the head of the specified queue without removing it.

* The .scan variable is the current scan pointer for the queue.

The logic for the enqueue operation is as follows:

```
enqueue(packet p) {
```

```
        while (SQ.size + BQ.size + S > MAXSIZE) {
            ; Queue overflow - drop from BQ head, then from SQ
            dp := pop(BQ)
            if (!dp)
                dp := pop(SQ)
            bkt := dp.H
            bkt.B -= 1
        }

        bkt := FH(p)
        p.T = NOW
        p.H = bkt
        if (bkt.B == 0 && bkt.D >= 0 && !bkt.K)
            push(SQ, p)
        else
            push(BQ, p)
        bkt.B += 1
    }
```

The logic for the dequeue operation is as follows:


```
    dequeue() {
```

```
    ; SQ gets strict priority
    p := pop(SQ)
    if (p) {
        send(p)
        bkt := p.H
        bkt.B -= 1
        bkt.D -= S
        if (bkt.D < 0) {
            bkt.K = true
            bkt.D += MTU
        }
        return
    }

    ; Process BQ if SQ was empty
    while (head(BQ)) {
        p := scan(BQ)
        if (!p) {
            ; Scan has reached tail of queue
            forall(f in FLOWS where f.B == 0 && !f.K)
                f.D = 0
            forall(f in FLOWS where f.K)
                f.K = false
            BQ.scan = BQ.head
            p := scan(BQ)
        }

        bkt := p.H
        if (!bkt.K) {
            ; Packet eligible for immediate delivery
            send(p)
            pull(BQ)
            bkt.B -= 1
            bkt.D -= S
            if (bkt.D < 0) {
                bkt.K = true
                bkt.D += MTU
            }
            return
        } else {
            ; Packet to stay in queue
            BQ.scan = BQ.scan.next
        }
    }
}
```

3.4. Simulator

A discrete time simulator for LFQ has been implemented, which acts as a supporting demonstration, verification of the algorithm's effectiveness and a test bed for exploration [LFQSIM].

4. Security Considerations

As with all FQ algorithms, an attacker may degrade service by flooding the queue with traffic that hashes into random buckets, or obtain enhanced service by using multiple flows where one would normally suffice. The latter may be mitigated by a flow mapping for individual hosts, or subscribers, rather than the 5-tuple.

5. IANA Considerations

There are no IANA considerations.

6. Informative References

[CAKE] Hoiland-Jorgensen, T., Taht, D., and J. Morton, "Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways", May 2018, <https://arxiv.org/abs/1804.07617>.

[DRRPP] MacGregor, M.H. and W. Shi, "Deficits for Bursty Latency-critical Flows: DRR++", September 2000, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=875803>.

[LFQSIM] "Lightweight Fair Queueing Simulator GitHub Repository", July 2019, <https://github.com/heistp/lfqsim/>.

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

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

[RFC8290] Hoeiland-Joergensen, T., McKenney, P., Taht, D., Gettys, J., and E. Dumazet, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm", RFC 8290, DOI 10.17487/RFC8290, January 2018,

[SFQ] McKenney, P.E., "Stochastic Fairness Queueing", June 2002, <http://www2.rdrop.com/~paulmck/scalability/paper/sfq.2002.06.04.pdf>.

Authors' Addresses

Jonathan Morton
Kokkonranta 21
FI-31520 Pitkajarvi
Finland

Phone: +358 44 927 2377
Email: chromatix99@gmail.com

Peter G. Heist
Redacted
463 11 Liberec 30
Czech Republic

Email: pete@heistp.net