Workgroup: Transport Working Group
Internet-Draft:
draft-morton-tsvwg-cheap-nasty-queueing-01
Published: 4 November 2019
Intended Status: Informational
Expires: 7 May 2020
Authors: J. Morton P. Heist

Cheap Nasty Queueing

Abstract

This note presents Cheap Nasty Queueing (CNQ), a queueing algorithm intended as a bare-minimum functionality standard for hardware implementations. It provides stateless or single-instance AQM and basic sparse-flow prioritisation.

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 7 May 2020.

Table of Contents

1. Introduction

Flow isolation is a powerful tool for congestion management in today's Internet. 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 and in consumer-grade hardware.

This note therefore presents CNQ, a queueing algorithm suitable for implementation in low-cost hardware, providing the absolute minimum functionality to improve perceived network performance over that of a dumb FIFO.

2. Background

CNQ 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++.

CNQ attempts to retain some 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 at most one set of AQM state. The operations required are believed to be amenable to low-cost hardware implementation.

3. The Algorithm

3.1. Overview

Unlike conventional fair queueing, with Cheap Nasty 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 latter would be appropriate for low-cost consumer hardware, in which the notion of a "subscriber" is neither well-defined nor useful.

The per-flow state is just one unsigned integer, in contrast to DRR++ which requires a whole queue and a set of AQM state per flow. This integer is B, tracking the backlog of the flow in packets. 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, and will contain at most one packet from each flow at any one time.

BQ is the "bulk queue" which handles all traffic not classed as sparse, including at least the second and subsequent packets in a burst. An AQM algorithm is applied to all traffic delivered from it.

To prevent well-paced traffic from dominating the queue by keeping exactly one packet in SQ at all times, a dummy packet is sent into BQ in parallel with every packet enqueued in SQ, and the B value for the flow effectively tracks the number of packets (including dummies) in BQ. A flow is therefore considered sparse IFF the interval between its packets is longer than the sojourn time of packets in BQ. This can be a much stricter criterion than for true derivatives of DRR++ such as LFQ.

The maximum throughput of a sparse flow is thus defined by the size of the packets composing that flow, divided by the sojourn time of BQ, under the assumption that the flow is well paced. For a typical example where the packet size is 1500 bytes and the sojourn time of BQ is controlled to 5ms by Codel AQM action, flows of up to about 2.5Mbps throughput may be treated as sparse.

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 compared to DRR++. The sharing of link capacity between flows is dependent mainly on the RTT-fair properties of the flows' own congestion control, in response to congestion signalling from the single AQM. However, this should be seen as an improvement in performance for sparse flows as compared to a plain FIFO queue.

3.2. Declarations

The following queues are defined:

The following constants and variables are defined:

- *B: the flow backlog, in packets
- *N: the number of flow buckets (each bucket containing a value of B)
- *S: the size of a packet
- *T: the packet's timestamp, for later use by AQM
- *H: the packet's flow hash, cached
- *MAXSIZE: the maximum size for all packets in the queue
- *NOW: the current timestamp

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 transmits the packet.
- *The aqm_action() function updates the AQM state (if any) based on the current sojourn time, and returns an action code indicating whether a CE or SCE mark (or no mark) should be applied. This function may be stateless and merely return results from a threshold function or probability ramp, or it may implement Codel or similar stateful AQMs, or a hybrid of the two for separate CE and SCE marking strategies.

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.

*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().

*The .head variable is the current head 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) {
        push(SQ, p)
        dp := zero-length dummy packet
        dp.T = NOW
        dp.H = bkt
        push(BQ, dp)
        bkt.B += 2
    } 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
        return
    }

    ; Process BQ if SQ was empty
    repeat {
        p := pop(BQ)
        if (!p) {
            ; Queue is empty
            return
        }

        bkt := p.H
        bkt.B -= 1

        if (p.S == 0) {
            ; Dummy packet for sparseness metric - drop
            continue
        }

        ; Apply AQM logic based on sojourn time
        t := NOW - p.T

        ; drop unresponsive traffic
        if (t > 500ms)
            continue

        switch(aqm_action(t)) {
        case MARK_CE:
            ; legacy congestion signalling
            if (t.ECN == Not-ECT)
                continue
            ; RFC-3168
            if (t.ECN == ECT || t.ECN == SCE)
                t.ECN = CE  ; and update IP header checksum
            break

        case MARK_SCE:
            ; Some Congestion Experienced
            if (t.ECN == ECT)
                t.ECN = SCE ; and update IP header checksum
            break

        default:
```

```
            ; no marking request
            break
        }

        send(p)
        return
    }
}
```

4. Security Considerations

This is a very weak FQ algorithm, not much better than a dumb FIFO - but still better.

5. IANA Considerations

There are no IANA considerations.

6. Informative References

[RFC8290] Hoeiland-Joergensen, T., McKenney, P., Taht, D., Gettys, J., and E. Dumazet, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm", RFC 8290, DOI 10.17487/RFC8290, January 2018, <https://www.rfc-editor.org/info/rfc8290>.

[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>.

[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>.

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