Active Queue Management and Packet Scheduling (aqm)
Internet-Draft
Intended status: Informational
Expires: September 28, 2015

G. White
CableLabs
R. Pan
Cisco Systems

March 27, 2015

A PIE-Based AQM for DOCSIS Cable Modems
draft-ietf-aqm-docsis-pie-00

Abstract

DOCSIS cable modems provide broadband Internet access to over one hundred million users worldwide. They are commonly positioned at the head of the bottleneck link for traffic in the upstream direction (from the customer), and as a result, the impact of buffering and bufferbloat in the cable modem can have a significant effect on user experience. The CableLabs DOCSIS 3.1 specification includes requirements for cable modems to support an Active Queue Management (AQM) algorithm that is intended to alleviate the impact that buffering has on latency sensitive traffic, while preserving bulk throughput performance. In addition, the CableLabs DOCSIS 3.0 specifications have also been amended to contain similar requirements.

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 28, 2015.

Table of Contents

1. Overview of DOCSIS AQM Requirements

CableLabs' DOCSIS 3.1 specification [DOCSIS_3.1] mandates that cable modems implement a specific variant of the Proportional Integral controller Enhanced (PIE) [I-D.ietf-aqm-pie] active queue management algorithm. This specific variant is provided for reference in Appendix A. CableLabs' DOCSIS 3.0 specification [DOCSIS_3.0] has been amended to recommend that cable modems implement the same algorithm. Both specifications allow that cable modems can optionally implement additional algorithms, that can then be selected for use by the operator via the modem's configuration file.

These requirements on the cable modem apply to upstream transmissions.

Both specifications also include requirements (mandatory in DOCSIS 3.1 and recommended in DOCSIS 3.0) that the Cable Modem Termination

System (CMTS) implement active queue management for downstream traffic, however no specific algorithm is defined for downstream use.

2. The DOCSIS MAC Layer and Service Flows

The DOCSIS Media Access Control (sub-)layer provides tools for configuring differentiated Quality of Service for different applications by the use of Packet Classifiers and Service Flows.

Each cable modem can be configured with multiple Packet Classifiers and Service Flows. The maximum number of such entities that a cable modem supports is an implementation decision for the manufacturer, but modems typically support 16 or 32 Service Flows and at least that many Packet Classifiers.

Each Service Flow has an associated Quality of Service (QoS) parameter set that defines the treatment of the packets that traverse the Service Flow. These parameters include (for example) Minimum Reserved Traffic Rate, Maximum Sustained Traffic Rate, Peak Traffic Rate, Maximum Traffic Burst, Traffic Priority. Each upstream Service Flow corresponds to a queue in the cable modem, and each downstream Service Flow corresponds to a queue in the CMTS. The DOCSIS AQM requirements mandate that the CM and CMTS implement the AQM algorithm (and allow it to be disabled if need be) on each Service Flow queue independently.

Packet Classifiers can match packets based upon several fields in the packet/frame headers including the Ethernet header, IP header, and TCP/UDP header. Matched packets are then queued in the associated Service Flow queue.

It is typical that upstream and downstream Service Flows used for broadband Internet access are configured with a Maximum Sustained Traffic Rate. This QoS parameter rate-shapes the traffic onto the DOCSIS link, and is the main parameter that defines the service offering. Additionally, it is common that upstream and downstream Service Flows are configured with a Maximum Traffic Burst and a Peak Traffic Rate. These parameters allow the service to burst at a higher (sometimes significantly higher) rate than is defined in the Maximum Sustained Traffic Rate for the amount of bytes configured in Maximum Traffic Burst, as long as the long-term average data rate remains at or below the Maximum Sustained Traffic Rate.

Mathematically, what is enforced is that the traffic placed on the DOCSIS link in the time interval (t1,t2) complies with the following rate shaping equations:

$$TxBytes(t1,t2) <= (t2-t1)*R/8 + B$$

TxBytes(t1,t2) <= (t2-t1)*P/8 + 1522

for all values t2>t1, where:

R = Maximum Sustained Traffic Rate (bps)

P = Peak Traffic Rate (bps)

B = Maximum Traffic Burst (bytes)

The result of this configuration is that the link rate available to the Service Flow varies based on the pattern of load. If the load that the Service Flow places on the link is less than the Maximum Sustained Traffic Rate, the Service Flow "earns" credit that it can then use (should the load increase) to burst at the Peak Traffic Rate. This dynamic is important since these rate changes (particularly the decrease in data rate once the traffic burst credit is exhausted) can induce a step increase in buffering latency.

3. DOCSIS-PIE vs. PIE

There are a number of differences between the version of the PIE algorithm that is mandated for cable modems in the DOCSIS specifications and the version described in [I-D.ietf-aqm-pie].

- 10 ms default latency target, configurable per service flow
- departure rate estimation
- expanded auto-tuning range
- trigger for exponential decay

3.1. Latency Target

The latency target (aka delay reference) is a key parameter that affects, among other things, the tradeoff in performance between latency-sensitive applications and bulk TCP applications. Via simulation studies, a value of 10ms was identified as providing a good balance of performance. However, it is recognized that there may be service offerings for which this value doesn't provide the best performance balance. As a result, this is provided as a configuration parameter that the operator can set independently on each upstream service flow. If not explicitly set by the operator, the modem will use 10 ms as the default value.

3.2. Departure rate estimation

The PIE algorithm utilizes a departure rate estimator to track fluctuations in the egress rate for the queue and to generate a smoothed estimate of this rate for use in the drop probability calculation. This estimator may be well suited to many link technologies, but is not ideal for DOCSIS upstream links for a number of reasons.

First, the bursty nature of the upstream transmissions, in which the queue drains at line rate (up to ~100 Mbps for DOCSIS 3.0 and ~1 Gbps for DOCSIS 3.1) and then is blocked until the next transmit opportunity, results in the potential for inaccuracy in measurement, given that the PIE departure rate estimator starts each measurement during a transmission burst and ends each measurement during a (possibly different) transmission burst. For example, in the case where the start and end of measurement occur within a single burst, the PIE estimator will calculate the egress rate to be equal to the line rate, rather than the average rate available to the modem.

Second, the latency introduced by the DOCSIS request-grant mechanism can result in some further inaccuracy. In typical conditions, the request-grant mechanism can add between ~4 ms and ~8 ms of latency to the forwarding of upstream traffic. Within that range, the amount of additional latency that affects any individual data burst is effectively random, being influenced by the arrival time of the burst relative to the next request transmit opportunity, among other factors.

Third, in the significant majority of cases, the departure rate, while variable, is controlled by the modem itself via the pair of token bucket rate shaping equations described in Section 2. Together, these two equations enforce a maximum sustained traffic rate, a peak traffic rate, and a maximum traffic burst size for the modem's requested bandwidth. The implication of this is that the modem, in the significant majority of cases, will know precisely what the departure rate will be, and can predict exactly when transitions between peak rate and maximum sustained traffic rate will occur. Compare this to the PIE estimator, which would be simply reacting to (and smoothing its estimate of) those rate transitions after the fact.

Finally, since the modem is already implementing the dual token bucket traffic shaper, it contains enough internal state to calculate predicted queuing delay with a minimum of computations. Furthermore, these computations only need to be run every drop probability update interval, as opposed to the PIE estimator, which runs a similar number of computations on each packet dequeue event.

For these reasons, the DOCSIS-PIE algorithm utilizes the configuration and state of the dual token bucket traffic shaper to translate queue depth into predicted queuing delay, rather than implementing the departure rate estimator defined in PIE.

3.3. Expanded auto-tuning range

The PIE algorithm scales the PI coefficients based on the current drop probability. The DOCSIS-PIE algorithm extends this scaling to drop probabilities below 1e-4.

3.4. Trigger for exponential decay

The PIE algorithm includes a mechanism by which the drop probability is allowed to decay exponentially (rather than linearly) when it is detected that the buffer is empty. In the DOCSIS case, recently arrived packets may reside in buffer due to the request-grant latency even if the link is effectively idle. As a result, the buffer may not be identically empty in the situations for which the exponential decay is intended. To compensate for this, we trigger exponential decay when the buffer occupancy is less than 5ms * Peak Traffic Rate.

4. Implementation Guidance

The AQM space is an evolving one, and it is expected that continued research in this field may in the future result in improved algorithms.

As part of defining the DOCSIS-PIE algorithm, we split the pseudocode definition into two components, a "data path" component and a "control path" component. The control path component contains the packet drop probability update functionality, whereas the data path component contains the per-packet operations, including the drop decision logic.

It is understood that some aspects of the cable modem implementation may be done in hardware, particularly functions that handle packet-processing.

While the DOCSIS specifications don't mandate the internal implementation details of the cable modem, modem implementers are strongly advised against implementing the control path functionality in hardware. The intent of this advice is to retain the possibility that future improvements in AQM algorithms can be accommodated via software updates to deployed devices.

5. References

[DOCSIS_3.0]
CableLabs, "DOCSIS 3.0 MAC and Upper Layer Protocols Specification", November 2013, <http://www.cablelabs.com/wp-content/uploads/specdocs/CM-SP-MULPIv3.0-I23-131120.pdf>.

[DOCSIS_3.1]
CableLabs, "DOCSIS 3.1 MAC and Upper Layer Protocols Specification", October 2013, <http://www.cablelabs.com/wp-content/uploads/specdocs/CM-SP-MULPIv3.1-I01-131029.pdf>.

[I-D.ietf-aqm-pie]
Pan, R., Natarajan, P., Baker, F., and G. White, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem", draft-ietf-aqm-pie-00 (work in progress), October 2014.

Appendix A. DOCSIS-PIE Algorithm definition

PIE defines two functions organized here into two design blocks:

1. Control path block, a periodically running algorithm that calculates a drop probability based on the estimated queuing latency and queuing latency trend.

2. Data path block, a function that occurs on each packet enqueue: per-packet drop decision based on the drop probability.

It is desired to have the ability to update the Control path block based on operational experience with PIE deployments.

A.1. DOCSIS-PIE AQM Constants and Variables

A.1.1. Configuration parameters

- LATENCY_TARGET. AQM Latency Target for this Service Flow

- PEAK_RATE. Service Flow configured Peak Traffic Rate, expressed in Bytes/sec.

- MSR. Service Flow configured Max. Sustained Traffic Rate, expressed in Bytes/sec.

- BUFFER_SIZE. The size (in bytes) of the buffer for this Service Flow.

A.1.2. Constant values

- A=0.25, B=2.5. Weights in the drop probability calculation
- INTERVAL=16 ms. Update interval for drop probability.
- DELAY_HIGH=200 ms.
- BURST_RESET_TIMEOUT = 1 s.
- MAX_BURST = 142 ms (150 ms-8 ms(update error))
- MEAN_PKTSIZE = 1024 bytes
- MIN_PKTSIZE = 64 bytes
- PROB_LOW = 0.85
- PROB_HIGH = 8.5
- LATENCY_LOW = 5 ms

A.1.3. Variables

- drop_prob_. The current packet drop probability.
- accu_prob_. accumulated drop prob. since last drop
- qdelay_old_. The previous queue delay estimate.
- burst_allowance_. Countdown for burst protection, initialize to 0
- burst_reset_. counter to reset burst
- burst_state_. Burst protection state encoding 3 states:

  NOBURST - no burst yet

  FIRST_BURST - first burst detected, no protection yet

  PROTECT_BURST - first burst detected, protecting burst if burst_allowance_ > 0

- queue_. Holds the pending packets.

A.1.4. Public/system functions:

- drop(packet). Drops/discards a packet
- random(). Returns a uniform r.v. in the range 0 ~ 1
- queue_.is_full(). Returns true if queue_ is full
- queue_.byte_length(). Returns current queue_ length in bytes, including all MAC PDU bytes without DOCSIS MAC overhead
- queue_.enque(packet). Adds packet to tail of queue_
- msrtokens(). Returns current token credits (in bytes) from the Max Sust. Traffic Rate token bucket
- packet.size(). Returns size of packet

A.2. DOCSIS-PIE AQM Control Path

The DOCSIS-PIE control path performs the following:

- Calls control_path_init() at service flow creation
- Calls calculate_drop_prob() at a regular INTERVAL (16ms)

```
================
//  Initialization function
control_path_init() {
    drop_prob_ = 0;
    qdelay_old_ = 0;
    burst_reset_ = 0;
    burst_state_ = NOBURST;
}

//  Background update, occurs every INTERVAL
calculate_drop_prob() {

    if (queue_.byte_length() <= msrtokens()) {
        qdelay = queue_.byte_length() / PEAK_RATE;
    } else {
        qdelay = ((queue_.byte_length() - msrtokens()) / MSR \
                  +  msrtokens() / PEAK_RATE);
    }

    if (burst_allowance_ > 0) {
        drop_prob_ = 0;
    } else {
```

```
            p = A * (qdelay - LATENCY_TARGET) + \
                B * (qdelay - qdelay_old_);
            // Since A=0.25 & B=2.5, can be implemented
            // with shift and add

            if (drop_prob_ < 0.000001) {
                p /= 2048;
            } else if (drop_prob_ < 0.00001) {
                p /= 512;
            } else if (drop_prob_ < 0.0001) {
                p /= 128;
            } else if (drop_prob_ < 0.001) {
                p /= 32;
            } else if (drop_prob_ < 0.01) {
                p /= 8;
            } else if (drop_prob_ < 0.1) {
                p /= 2;
            } else if (drop_prob_ < 1) {
                p /= 0.5;
            } else if (drop_prob_ < 10) {
                p /= 0.125;
            } else {
                p /= 0.03125;
            }

            if ((drop_prob_ >= 0.1) && (p > 0.02)) {
                p = 0.02;
            }
            drop_prob_ += p;

            /* for non-linear drop in prob */
            if (qdelay < LATENCY_LOW && qdelay_old_ < LATENCY_LOW) {
                drop_prob_ *= 0.98;    // (1-1/64) is sufficient
            } else if (qdelay > DELAY_HIGH) {
                drop_prob_ += 0.02;
            }

            drop_prob_ = max(0, drop_prob_);
            drop_prob_ = min(drop_prob_, \
                         PROB_LOW * MEAN_PKTSIZE/MIN_PKTSIZE);
        }

        if (burst_allowance_ < INTERVAL)
            burst_allowance_ = 0;
        else
            burst_allowance_ = burst_allowance_ - INTERVAL;

    // both old and new qdelay is well better than the
```

```
   // target and drop_prob_ == 0, time to clear burst tolerance
       if ((qdelay < 0.5 * LATENCY_TARGET)
           && (qdelay_old_ < 0.5 * LATENCY_TARGET)
           && (drop_prob_ == 0)
           && (burst_allowance_ == 0)){

           if (burst_state_ == PROTECT_BURST) {
               burst_state_ = FIRST_BURST;
               burst_reset_ = 0;

           } else if (burst_state_ == FIRST_BURST) {
               burst_reset_ += INTERVAL ;
               if (burst_reset_ > BURST_RESET_TIMEOUT) {
                   burst_reset_ = 0;
                   burst_state_ = NOBURST;
               }
           }
       } else if (burst_state_ == FIRST_BURST) {
               burst_reset_ = 0;
       }

       qdelay_old_ = qdelay;

   }
```

A.3. DOCSIS-PIE AQM Data Path

The DOCSIS-PIE data path performs the following:

o Calls enque() in response to an incoming packet from the CMCI

```
   ================
   enque(packet) {
       if (queue_.is_full()) {
           drop(packet);
           accu_prob_ = 0;
       } else if (drop_early(packet, queue_.byte_length())) {
           drop(packet);
       } else {
           queue_.enque(packet);
       }
   }

   ////////////////
   drop_early(packet, queue_length) {
       if (burst_allowance_ > 0) {
           return FALSE;
       }
```

```
       if (drop_prob_ == 0) {
           accu_prob_ = 0;
       }

       if (burst_state_ == NOBURST) {                    //first burst?
           if (queue_.byte_length() < BUFFER_SIZE/3) {
               return FALSE;
           } else {
               burst_state_ = FIRST_BURST;             //burst detected
           }
       }

       //The CM can quantize packet.size to 64, 128, 256, 512, 768,
       // 1024, 1280, 1536, 2048 in the calculation below
       p1 = drop_prob_ * packet.size() / MEAN_PKTSIZE;
       p1 = min(p1, PROB_LOW);

       accu_prob_ += p1;

      // If latency is low, don't drop packets
      if ( (qdelay_old_ < 0.5 * LATENCY_TARGET && drop_prob_ < 0.2)
             || (queue_.byte_length() <= 2 * MEAN_PKTSIZE) ) {
           return FALSE;
      }

       drop = TRUE;
       if (accu_prob_ < PROB_LOW) {  // avoid dropping too fast due
            drop = FALSE;            // to bad luck of coin tosses...
       } else if (accu_prob_ >= PROB_HIGH) { // ...and avoid droppping
           drop = TRUE;                       // too slowly
       } else {                        //Random drop
           double u = random();        // 0 ~ 1
           if (u > p1) {
              drop = FALSE;
           }
       }

       if (drop == FALSE) return FALSE;

       // In case of packet drop:
       accu_prob_ = 0;

       // Not protecting burst yet? Start protecting burst.
       // This will set the burst_allowance_ value, and
       // calculate_drop_prob() will decrement it.
       // Could implement this as a 150ms timer instead.
       if (burst_state_ == FIRST_BURST) {
           burst_state_ = PROTECT_BURST;
```

```
            burst_allowance_ = MAX_BURST;
        }
        return TRUE;
    }
```

Authors' Addresses

Greg White
CableLabs
858 Coal Creek Circle
Louisville, CO 80027-9750
USA

Email: g.white@cablelabs.com

Rong Pan
Cisco Systems
510 McCarthy Blvd
Milpitas, CA 95134
USA

Email: ropan@cisco.com