

Internet Draft
Active Queue Management
Working Group
Intended Status: Standards Track

R. Pan, P. Natarajan, F. Baker
G. White, B. VerSteeg, M.S. Prabhu
C. Piglione, V. Subramanian

Expires: February 11, 2016

August 10, 2015

**PIE: A Lightweight Control Scheme To Address the
Bufferbloat Problem**

[draft-ietf-aqm-pie-02](#)

Abstract

Bufferbloat is a phenomenon where excess buffers in the network cause high latency and jitter. As more and more interactive applications (e.g. voice over IP, real time video streaming and financial transactions) run in the Internet, high latency and jitter degrade application performance. There is a pressing need to design intelligent queue management schemes that can control latency and jitter; and hence provide desirable quality of service to users.

This document presents a lightweight active queue management design, called PIE (Proportional Integral controller Enhanced), that can effectively control the average queueing latency to a target value. Simulation results, theoretical analysis and Linux testbed results have shown that PIE can ensure low latency and achieve high link utilization under various congestion situations. The design does not require per-packet timestamp, so it incurs very small overhead and is simple enough to implement in both hardware and software.

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/1id-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

Copyright and License Notice

Copyright (c) 2012 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	4
2.	Terminology	5
3.	Design Goals	5
4.	The Basic PIE Scheme	6
4.1	Random Dropping (ECN Support is described later in this document)	6
4.2	Drop Probability Calculation	7
4.3	Latency Calculation	8
4.4	Protecting Bursts	9
4.5	Exponential Decay Drop Probability	9
5.	Optional Design Elements of PIE	9
5.1	ECN Support	10
5.2	Departure Rate Estimation	10
5.3	Turning PIE on and off	12
5.4	De-randomization	12
6.	Implementation Cost	13
7.	Future Research	14
8.	Incremental Deployment	14
9.	IANA Considerations	14
10.	References	14
10.1	Normative References	15
10.2	Informative References	15

10.3	Other References	15
	Authors' Addresses	16
11	The Basic PIE pseudo Code	17
12	Pseudo code for PIE with optional enhancement	19

1. Introduction

The explosion of smart phones, tablets and video traffic in the Internet brings about a unique set of challenges for congestion control. To avoid packet drops, many service providers or data center operators require vendors to put in as much buffer as possible. With rapid decrease in memory chip prices, these requests are easily accommodated to keep customers happy. While this solution succeeds in assuring low packet loss and high TCP throughput, it suffers from a major downside. The TCP protocol continuously increases its sending rate and causes network buffers to fill up. TCP cuts its rate only when it receives a packet drop or mark that is interpreted as a congestion signal. However, drops and marks usually occur when network buffers are full or almost full. As a result, excess buffers, initially designed to avoid packet drops, would lead to highly elevated queueing latency and jitter. It is a delicate balancing act to design a queue management scheme that not only allows short-term burst to smoothly pass, but also controls the average latency in the presence of long-running greedy flows.

Active queue management (AQM) schemes, such as Random Early Discard (RED), have been around for well over a decade. AQM schemes could potentially solve the aforementioned problem. [RFC 2309](#)[\[RFC2309\]](#) strongly recommends the adoption of AQM schemes in the network to improve the performance of the Internet. RED is implemented in a wide variety of network devices, both in hardware and software. Unfortunately, due to the fact that RED needs careful tuning of its parameters for various network conditions, most network operators don't turn RED on. In addition, RED is designed to control the queue length which would affect delay implicitly. It does not control latency directly. Hence, the Internet today still lacks an effective design that can control buffer latency to improve the quality of experience to latency-sensitive applications. Notably, a recent IETF AQM working group draft [[IETF-AQM](#)] calls for new methods of controlling network latency.

New algorithms are beginning to emerge to control queueing latency directly to address the bufferbloat problem [[CoDel](#)]. Along these lines, PIE also aims to keep the benefits of RED: such as easy implementation and scalability to high speeds. Similar to RED, PIE randomly drops an incoming packet at the onset of the congestion. The congestion detection, however, is based on the queueing latency instead of the queue length like RED. Furthermore, PIE also uses the derivative (rate of change) of the queueing latency to help determine congestion levels and an appropriate response. The design parameters of PIE are chosen via control theory stability analysis. While these parameters can be fixed to work in various traffic conditions, they could be made self-tuning to optimize system performance.

Separately, it is assumed that any delay-based AQM scheme would be applied over a Fair Queueing (FQ) structure or one of its approximate designs, Flow Queueing or Class Based Queueing (CBQ). FQ is one of the most studied scheduling algorithms since it was first proposed in 1985 [[RFC970](#)]. CBQ has been a standard feature in most network devices today[CBQ]. Any AQM scheme that is built on top of FQ or CBQ could benefit from these advantages. Furthermore, these advantages such as per flow/class fairness are orthogonal to the AQM design whose primary goal is to control latency for a given queue. For flows that are classified into the same class and put into the same queue, one needs to ensure their latency is better controlled and their fairness is not worse than those under the standard DropTail or RED design. More details about the relationship between FQ and AQM can be found in IETF draft [[FQ-Implement](#)].

In October 2013, CableLabs' DOCSIS 3.1 specification [[DOCSIS 3.1](#)] mandated that cable modems implement a specific variant of the PIE design as the active queue management algorithm. In addition to cable specific improvements, the PIE design in DOCSIS 3.1 [[DOCSIS-PIE](#)] has improved the original design in several areas: de-randomization of coin tosses, enhanced burst protection and expanded range of auto-tuning.

This draft separates the PIE design into the basic elements that are MUST to be implemented and optional SHOULD/MAY enhancement elements.

2. Terminology

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 [RFC 2119](#) [[RFC2119](#)].

3. Design Goals

A queue management framework is designed to improve the performance of interactive and delay-sensitive applications. It should follow the general guidelines set by the AQM working group document "IETF Recommendations Regarding Active Queue Management" [[IETF-AQM](#)]. More specifically PIE design has the following basic criteria.

- * First, queueing latency, instead of queue length, is controlled. Queue sizes change with queue draining rates and various flows' round trip times. Delay bloat is the real issue that needs to be addressed as it impairs real time applications.

If latency can be controlled, bufferbloat is not an issue. In fact, once latency is under control it frees up buffers for sporadic bursts.

* Secondly, PIE aims to attain high link utilization. The goal of low latency shall be achieved without suffering link under-utilization or losing network efficiency. An early congestion signal could cause TCP to back off and avoid queue building up. On the other hand, however, TCP's rate reduction could result in link under-utilization. There is a delicate balance between achieving high link utilization and low latency.

* Furthermore, the scheme should be simple to implement and easily scalable in both hardware and software. PIE strives to maintain similar design simplicity to RED, which has been implemented in a wide variety of network devices.

* Finally, the scheme should ensure system stability for various network topologies and scale well with arbitrary number streams. Design parameters shall be set automatically. Users only need to set performance-related parameters such as target queue delay, not design parameters.

In the following, the design of PIE and its operation are described in deta.

4. The Basic PIE Scheme

As illustrated in Fig. 1, PIE conceptually comprises three simple MUST components: a) random dropping at enqueueing; b) periodic drop probability update; c) latency calculation. The following sections describe these components in further detail, and explain how they interact with each other.

4.1 Random Dropping (ECN Support is described later in this document)

PIE MUST drop packets randomly according to a drop probability, p , that is obtained from the drop-probability-calculation component:

- * upon a packet arrival MUST

- randomly drop a packet with a probability p .

Random Drop

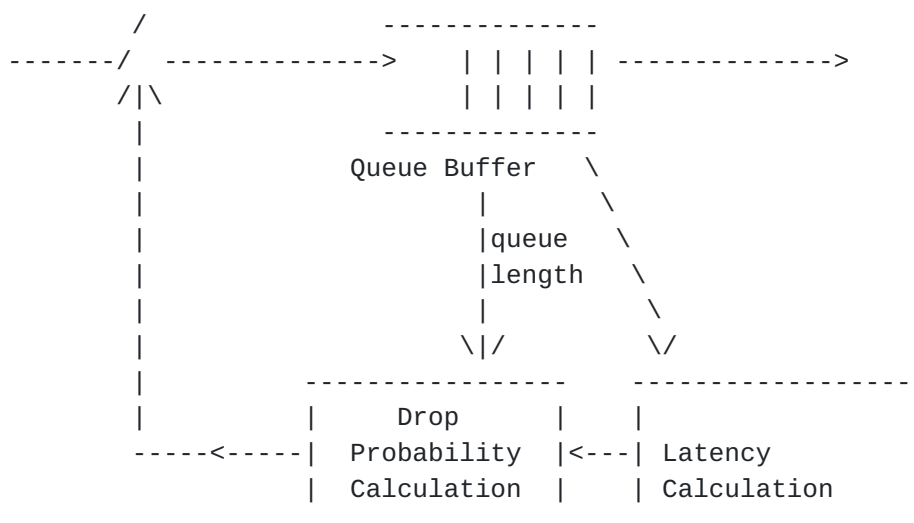


Figure 1. The PIE Structure

PIE optionally supports ECN and will be discussed in [Section 5.1](#).

4.2 Drop Probability Calculation

The PIE algorithm MUST periodically adjust the drop probability every Tupdate interval:

* MUST calculate drop probability p as:

$$p = p + \alpha * (\text{est_del} - \text{target_del}) + \beta * (\text{est_del} - \text{est_del_old});$$

$$\text{est_del_old} = \text{est_del}.$$

* MUST autotune the α and β parameters based on drop probability p :

```
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 {  
        p = p;  
    }  
}
```

Here, the current queue length is denoted by qlen. The update interval is denoted as Tupdate. Tupdate is defaulted to be 16ms. It can be reduced on high speed links in order to provide smoother response. The target latency value is expressed in target_del, which SHOULD be set to 15ms. Variables, est_del and est_del_old represent the current and previous estimation of the queueing delay.

Note that the adjustment to drop probability is based not only on the current estimation of the queueing delay, but also on the rate of change of queueing delay. This rate of change can simply be measured as the difference between est_del and est_del_old. The result is the classical Proportional Integral (PI) controller design which is known for eliminating steady state errors. It is adopted here to control queueing latency so that, at the steady state, the difference between the queueing latency and the target value is zero even under heavy load. The controller parameters, in the unit of hz, are designed using feedback loop analysis where TCP's behaviors are modeled using the results from well-studied prior art[TCP-Models].

This type of controller has been studied before for controlling the queue length [[PI](#), [QCN](#)]. PIE adopts the Proportional Integral controller for controlling delay and makes the scheme auto-tuning. The theoretical analysis of PIE can be found in [[HPSR-PIE](#)]. As a rule of thumb, if we cut Tupdate in half, we should also cut alpha by half and increase beta by alpha/4 in order to keep the same feedback loop dynamics.

[4.3 Latency Calculation](#)

The PIE algorithm MUST use latency to calculate drop probability.

- * It MAY estimate current queueing delay using Little's law:

```
est_del = qlen/depart_rate;
```

- * or MAY use other techniques for calculating queueing delay, ex: timestamp packets at enqueue and use the same to calculate delay during dequeue.

4.4 Protecting Bursts

PIE MUST NOT penalize short-term packet bursts [[IETF-AQM](#)]. PIE MUST give users precise control of how much burst to allow without penalty. A parameter, `max_burst`, is introduced that is similar to the burst tolerance in the token bucket design. By default, the parameter SHOULD be set to be 150ms (MUST be > 0). The PIE algorithm MUST do the following:

```
* if p == 0 and est_del < del_ref and est_del_old < del_ref

    burst_allowance = max_burst;

* upon packet arrival

    if burst_allowance > 0 enqueue packet;

* upon probability update

    burst_allowance = burst_allowance - Tupdate;
```

The burst allowance, noted by `burst_allowance`, is initialized to `max_burst`. As long as `burst_allowance` is above zero, an incoming packet will be enqueued bypassing the random drop process. During each update instance, the value of `burst_allowance` is decremented by the update period, `Tupdate`. When the congestion goes away, defined here as `p` equals to 0 and both the current and previous samples of estimated delay are less than `target_del`, `burst_allowance` is reset to `max_burst`.

4.5 Exponential Decay Drop Probability

The PIE algorithm MUST include a mechanism by which the drop probability decay exponentially (rather than linearly) when the system is not congested. This would help the drop probability converge to 0 much faster than the PI controller dictates. The decay parameter of 2% gives us around 750ms time constant, a few RTT.

```
* upon probability update (Tupdate interval):

    if (est_delay == 0 && est_delay_old_ == 0) {

        p = p*0.98;    //1- 1/64 is sufficient

    }
```

5. Optional Design Elements of PIE

The above forms the basic MUST have elements of the PIE algorithm. There are several enhancements that are added to further augment the performance of the basic algorithm. For clarity purpose, they are included in this section.

5.1 ECN Support

PIE SHOULD support ECN by marking (rather than dropping) ECN capable packets. However, as a safeguard, an additional threshold, `mark_ecnth`, is introduced. If the calculated drop probability exceeds `mark_ecnth`, PIE MUST revert to packet drop for ECN capable packets. The variable `mark_ecnth` SHOULD be set at 10%.

```
* if rand() < p

    if p < mark_ecnth && ecn_capable_packet == TRUE:

        mark packet;

    else:

        drop packet;
```

5.2 Departure Rate Estimation

One way to calculate latency is to obtain an estimation of the departure rate as discussed in [Section 4.3](#). The draining rate of a queue in the network often varies either because other queues are sharing the same link, or the link capacity fluctuates. Rate fluctuation is particularly common in wireless networks. Hence, one MAY decide to measure the departure rate directly as follows.

```
* if rate_measurement == FALSE and qlen > dq_threshold:

    rate_measurement = TRUE;

    start = now;

* if rate_measurement == TRUE:

    upon a packet departure:

        dq_count = dq_count + deque_pkt_size;

        if dq_count > dq_threshold then
```



```

depart_rate = dq_count/(now-start);

dq_count=0;

start = now

```

The departure rate is measured when there are sufficient data in the buffer, i.e., when the queue length is over a certain threshold, DQ_THRESHOLD. Short, non-persistent bursts of packets result in empty queues from time to time, this would make the measurement less accurate. The parameter, dq_count, represents the number of bytes departed since the last measurement. Once dq_count is over a certain threshold, DQ_THRESHOLD, a measurement sample is obtained. The threshold is recommended to be set to 16KB assuming a typical packet size of around 1KB or 1.5KB. This threshold would allow us a long enough period to obtain an average draining rate but also fast enough to reflect sudden changes in the draining rate. This threshold is not crucial for the system's stability. Please note that the update interval for calculating the drop probability is different from the rate measurement cycle. The drop probability calculation is done periodically per [section 4.2](#) and it is done even when the algorithm is not in a measurement cycle; in this case the previously latched value of depart_rate is used.

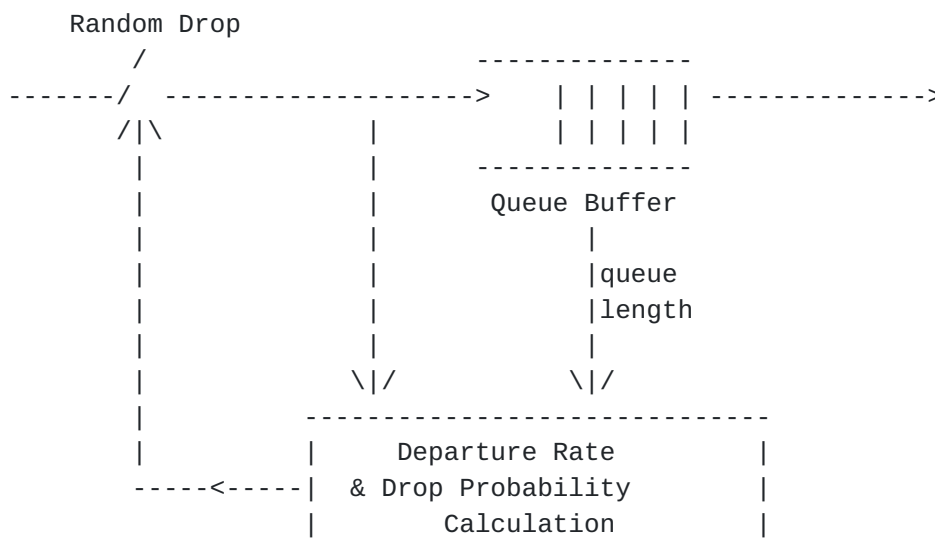


Figure 2. The Enqueue-based PIE Structure

In some platforms, enqueueing and dequeueing functions belong to different modules that are independent to each other. In such situations, a pure enqueue-based design MAY be designed. As shown in Figure 2, an enqueue-based design is depicted. The departure rate is deduced from the number of packets enqueued and the queue length. The design is based on the following key observation: over a certain time interval, the number of departure packets = the number of enqueued packets - the number of remaining packets in queue. In this design, everything can be triggered by a packet arrival including the background update process. The design complexity here is similar to the original design.

5.3 Turning PIE on and off

Traffic naturally fluctuates in a network. It would be preferable not to unnecessarily drop packets due to a spurious uptick in queueing latency. PIE can be optionally turned on and off. IT SHOULD only be turned on (from off) when the buffer occupancy is over a certain threshold, which SHOULD be set to 1/3 of the tail drop threshold. If it is on, PIE SHOULD be turned off when congestion is over, i.e. when the drop probability, queue length and estimated queue delay all reach 0.

Ideally PIE should be turned on or off based on the latency. However, calculating latency when PIE is off would introduce unnecessary packet processing overhead. Weighing the trade-offs, it is decided to compare against tail drop threshold to keep things simple.

When PIE is optionally turned on and off, the burst protection logic is modified as follows:

```
* if PIE_active == FALSE

    burst_allowance = MAX_BURST;

* upon packet arrival

    if burst_allowance > 0 enqueue packet;

* upon probability update when PIE_active == TRUE

    burst_allowance = burst_allowance - Tupdate;
```

5.4 De-randomization

Although PIE adopts random dropping to achieve latency control, independent coin tosses could introduce outlier situations where packets are dropped too close to each other or too far from each other. This

would cause real drop percentage to temporarily deviate from the intended drop probability p . In certain scenarios, such as small number of simultaneous TCP flows, these deviations can cause significant deviations in link utilization and queueing latency. PIE MAY introduce a de-randomization mechanism to avoid such scenarios. A parameter, called `accu_prob`, is reset to 0 after a drop. Upon a packet arrival, `accu_prob` is incremented by the amount of drop probability, p . If `accu_prob` is less than a low threshold, e.g. 0.85, the arriving packet is enqueued; on the other hand, if `accu_prob` is more than a high threshold, e.g. 8.5, a packet is forced to be dropped. A packet is only randomly dropped if `accu_prob` falls in between the two thresholds. Since `accu_prob` is reset to 0 after a drop, another drop will not happen until $0.85/p$ packets later. This avoids packets being dropped too close to each other. In the other extreme case where $8.5/p$ packets have been enqueued without incurring a drop, PIE would force a drop that prevents much fewer drops than desired. Further analysis can be found in [[DOCSIS-PIE](#)].

6. Implementation Cost

PIE can be applied to existing hardware or software solutions. There are three steps involved in PIE as discussed in [Section 4](#). their complexities are examined below.

Upon packet arrival, the algorithm simply drops a packet randomly based on the drop probability p . This step is straightforward and requires no packet header examination and manipulation. If the implementation doesn't rely on packet timestamps for calculating latency, PIE does not require extra memory. Furthermore, the input side of a queue is typically under software control while the output side of a queue is hardware based. Hence, a drop at enqueueing can be readily retrofitted into existing hardware or software implementations.

The drop probability calculation is done in the background and it occurs every Tudpate interval. Given modern high speed links, this period translates into once every tens, hundreds or even thousands of packets. Hence the calculation occurs at a much slower time scale than packet processing time, at least an order of magnitude slower. The calculation of drop probability involves multiplications using α and β . Since PIE's control law is robust to minor changes in α and β values, an implementation MAY choose these values to the closest multiples of 2 or 1/2 (ex: $\alpha=0.125$, $\beta=1.25$) such that the multiplications can be done using simple adds and shifts. As no complicated functions are required, PIE can be easily implemented in both hardware and software. The state requirement is only two variables per queue: `est_del` and `est_del_old`. Hence the memory overhead is small.

If one chooses to implement the departure rate estimation, PIE uses a

counter to keep track of the number of bytes departed for the current interval. This counter is incremented per packet departure. Every Tupdate, PIE calculates latency using the departure rate, which can be implemented using a multiplication. Note that many network devices keep track of an interface's departure rate. In this case, PIE might be able to reuse this information, simply skip the third step of the algorithm and hence incurs no extra cost. If platform already leverages packet timestamps for other purposes, PIE MAY make use of these packet timestamps for latency calculation instead of estimating departure rate.

Since the PIE design is separated into data path and control path, if control path is implemented in software, any further improvement in control path can be easily accommodated.

In summary, PIE is simple to implement. SFQ can be combined with PIE to provide further improvement of latency for various flows with different priorities.

7. Future Research

The design of the PIE algorithm is presented in this document. It effectively controls the average queueing latency to a target value. The following areas can be further studied:

- * Autotuning of target delay without losing utilization;
- * Autotuning for average RTT of traffic;

8. Incremental Deployment

PIE scheme can be independently deployed and managed without any need for interoperability.

Although all network nodes cannot be changed altogether to adopt latency-based AQM schemes, a gradual adoption would eventually lead to end-to-end low latency service for all applications.

9. IANA Considerations

There are no actions for IANA.

10. References

10.1 Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

10.2 Informative References

[RFC970] Nagle, J., "On Packet Switches With Infinite Storage", [RFC970](#), December 1985.

10.3 Other References

[IETF-AQM] Baker, F. and Fairhurst, G., "IETF Recommendations Regarding Active Queue Management", [draft-ietf-aqm-recommendation-11](#).

[CoDel] Nichols, K., Jacobson, V., "Controlling Queue Delay", ACM Queue. ACM Publishing. doi:10.1145/2209249.22W.09264.

[CBQ] Cisco White Paper, "http://www.cisco.com/en/US/docs/12_0t/12_0tfeature/guide/cbwfq.html".

[FQ-Implement] Baker, F. and Pan, R. "On Queueing, Marking and Dropping", IETF [draft-ietf-aqm-fq-implementation](#).

[DOCSIS_3.1] <http://www.cablelabs.com/wp-content/uploads/specdocs/CM-SP-MULPIv3.1-I01-131029.pdf>.

[DOCSIS-PIE] White, G. and Pan, R., "A PIE-Based AQM for DOCSIS Cable Modems", IETF [draft-white-aqm-docsis-pie-00](#).

[HPSR-PIE] Pan, R., Natarajan, P. Piglione, C., Prabhu, M.S., Subramanian, V., Baker, F. Steeg and B. V., "PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem", IEEE HPSR 2013.

[AQM DOCSIS] http://www.cablelabs.com/wp-content/uploads/2014/06/DOCSIS-AQM_May2014.pdf

[TCP-Models] Misra, V., Gong, W., and Towsley, D., "Fluid-base Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED", SIGCOMM 2000

[PI] Hollot, C.V., Misra, V., Towsley, D. and Gong, W., "On Designing Improved Controller for AQM Routers Supporting TCP Flows", Infocom 2001.

[QCN] "Data Center Bridging - Congestion Notification",
<http://www.ieee802.org/1/pages/802.1au.html>.

Authors' Addresses

Rong Pan
Cisco Systems
3625 Cisco Way,
San Jose, CA 95134, USA
Email: ropan@cisco.com

Preethi Natarajan,
Cisco Systems
725 Alder Drive,
Milpitas, CA 95035, USA
Email: prenatar@cisco.com

Fred Baker
Cisco Systems
725 Alder Drive,
Milpitas, CA 95035, USA
Email: fred@cisco.com

Bill Ver Steeg
Cisco Systems
5030 Sugarloaf Parkway
Lawrenceville, GA, 30044, USA
Email: versteb@cisco.com

Mythili Prabhu*
Akamai Technologies
3355 Scott Blvd
Santa Clara, CA - 95054
Email: mythili@akamai.com

Chiara Piglione*
Broadcom Corporation
3151 Zanker Road
San Jose, CA 95134
Email: chiara@broadcom.com

Vijay Subramanian*
PLUMgrid, Inc.
350 Oakmead Parkway,
Suite 250
Sunnyvale, CA 94085
Email: vns@plumgrid.com

Greg White
CableLabs
858 Coal Creek Circle
Louisville, CO 80027, USA
Email: g.white@cablelabs.com

* Formerly at Cisco Systems

11. The Basic PIE pseudo Code

Configurable Parameters:

- QDELAY_REF. AQM Latency Target (default: 16ms)
- MAX_BURST. AQM Max Burst Allowance (default: 150ms)

Internal Parameters:

- Weights in the drop probability calculation (1/s):
alpha (default: 1/8), beta(default: 1+1/4)
- T_UPDATE: a period to calculate drop probability (default:16ms)

Table which stores status variables (ending with "_"):

- burst_allowance_: current burst_allowance
- drop_prob_: The current packet drop probability. reset to 0
- current_qdelay_: The current queue delay. reset to 0
- qdelay_old_: The previous queue delay. reset to 0

Public/system functions:

- queue_. Holds the pending packets.
- drop(packet). Drops/discards a packet
- now(). Returns the current time
- random(). Returns a uniform r.v. in the range 0 ~ 1
- queue_.byte_length(). Returns current queue_ length in bytes
- queue_.enqueue(packet). Adds packet to tail of queue_
- queue_.deque(). Returns the packet from the head of queue_
- packet.size(). Returns size of packet
- packet.timestamp_delay(). Returns timestamped packet latency

=====

//called on each packet arrival

```
enqueue(Packet packet) {  
    if (PIE->drop_prob_ == 0 && PIE->current_qdelay_ < del_ref  
        && PIE->qdelay_old < del_ref) {  
        burst_allowance = MAX_BURST;  
    }  
}
```



```
    if (PIE->burst_allowance_ < 0 && drop_early() == DROP
        && PIE->burst_allowance_ <= 0) {
        drop(packet);
    } else {
        queue_.enqueue(packet);
    }
}
```

=====

```
drop_early() {

    //Safeguard PIE to be work conserving
    if ( (PIE->qdelay_old_ < QDELAY_REF/2 && PIE->drop_prob_ < 20%)
        || (queue_.byte_length() <= 2 * MEAN_PKTSIZE) ) {
        return ENQUEUE;
    }

    double u = random();
    if (u < PIE->drop_prob_) {
        return DROP;
    } else {
        return ENQUEUE;
    }
}
```

=====

```
//we choose the timestamp option of obtaining latency for clarity
//rate estimation method can be found in the extended PIE pseudo code
```

```
deque(Packet packet) {

    PIE->current_qdelay_ = packet.timestamp_delay();

}
```

=====

```
//update periodically, T_UPDATE = 16ms
```

```
calculate_drop_prob() {

    //can be implemented using integer multiply,
    qdelay = PIE->current_qdelay_;

    p = alpha*(qdelay - QDELAY_REF) + \
        beta*(qdelay-PIE->qdelay_old_);
```



```

//Expanding scaling range can help improve performance.
//Please see DOCSIS-PIE design.
//We keep it simple here
if (PIE->drop_prob_ < 0.1%) {
    p = p/128
} else if (PIE->drop_prob_ < 1%) {
    p = p/16;
} else if (PIE->drop_prob_ < 10%) {
    p = p/2;
} else {
    p = p;
}

PIE->drop_prob_ += p;

//Exponentially decay drop prob when congestion goes away
if (qdelay == 0 && PIE->qdelay_old_ == 0) {
    PIE->drop_prob_ *= 0.98;    //1- 1/64 is sufficient
}

//bound drop probability
if (PIE->drop_prob_ < 0)
    PIE->drop_prob_ = 0
if (PIE->drop_prob_ > 1)
    PIE->drop_prob_ = 1

PIE->qdelay_old_ = qdelay;
PIE->last_timestamp_ = now;
if (PIE->burst_allowance_ > 0) {
    PIE->burst_allowance_ = PIE->burst_allowance_ - T_UPDATE;
}
}
}

```

12. Pseudo code for PIE with optional enhancement

Configurable Parameters:

- QDELAY_REF. AQM Latency Target (default: 16ms)
- MAX_BURST. AQM Max Burst Allowance (default: 150ms)
- MAX_ECINTH. AQM Max ECN Marking Threshold (default: 10%)

Internal Parameters:

- Weights in the drop probability calculation (1/s):
 alpha (default: 1/8), beta(default: 1+1/4)

- DQ_THRESHOLD: (in bytes, default: 2^{14} (in a power of 2))
- T_UPDATE: a period to calculate drop probability (default:16ms)
- TAIL_DROP: each queue has a tail drop threshold, pass it to PIE

Table which stores status variables (ending with "_"):

- active_: INACTIVE/ACTIVE
- burst_allowance_: current burst_allowance
- drop_prob_: The current packet drop probability. reset to 0
- accu_prob_: Accumulated drop probability. reset to 0
- qdelay_old_: The previous queue delay estimate. reset to 0
- last_timestamp_: Timestamp of previous status update
- dq_count_, measurement_start_, in_measurement_, avg_dq_time_. variables for measuring avg_dq_rate_.

Public/system functions:

- queue_. Holds the pending packets.
- drop(packet). Drops/discards a packet
- mark(packet). Marks ECN for a packet
- now(). Returns the current time
- random(). Returns a uniform r.v. in the range 0 ~ 1
- queue_.byte_length(). Returns current queue_ length in bytes
- queue_.enqueue(packet). Adds packet to tail of queue_
- queue_.dequeue(). Returns the packet from the head of queue_
- packet.size(). Returns size of packet
- packet.ecn(). Returns whether packet is ECN capable or not

=====

//called on each packet arrival

```
enqueue(Packet packet) {
    if (queue_.byte_length()+packet.size() > TAIL_DROP) {
        drop(packet);
        PIE->accu_prob_ = 0;
    } else if (PIE->active_ == TRUE && drop_early() == DROP
               && PIE->burst_allowance_ <= 0) {
        if (PIE->drop_prob_ < MAX_ECNTH && packet.ecn() == TRUE)
            mark(packet);
        else
            drop(packet);
        PIE->accu_prob_ = 0;
    } else {
        queue_.enqueue(packet);
    }
}
```

//If the queue is over a certain threshold, turn on PIE

```
if (PIE->active_ == INACTIVE
    && queue_.byte_length() >= TAIL_DROP/3) {
```



```
    PIE->active_ = ACTIVE;
    PIE->qdelay_old_ = 0;
    PIE->drop_prob_ = 0;
    PIE->in_measurement_ = TRUE;
    PIE->dq_count_ = 0;
    PIE->avg_dq_time_ = 0;
    PIE->last_timestamp_ = now;
    PIE->burst_allowance_ = MAX_BURST;
    PIE->accu_prob_ = 0;
    PIE->measurement_start_ = now;
}

//If the queue has been idle for a while, turn off PIE
//reset counters when accessing the queue after some idle
//period if PIE was active before
if ( PIE->drop_prob_ == 0 && PIE->qdelay_old_ == 0
    && queue_.byte_length() == 0 ) {
    PIE->active_ = INACTIVE;
    PIE->in_measurement_ = FALSE;
}
}

=====

drop_early() {

    //PIE is active but the queue is not congested, return ENQUE
    if ( (PIE->qdelay_old_ < QDELAY_REF/2 && PIE->drop_prob_ < 20%)
        || (queue_.byte_length() <= 2 * MEAN_PKTSIZE) ) {
        return ENQUE;
    }

    if (PIE->drop_prob_ == 0) {
        PIE->accu_prob_ = 0;
    }

    //For practical reasons, drop probability can be further scaled
    //according to packet size. but need to set a bound to
    //avoid unnecessary bias

    //Random drop
    PIE->accu_prob_ += PIE->drop_prob_;
    if (PIE->accu_prob_ < 0.85)
        return ENQUE;
    if (PIE->accu_prob_ >= 8.5)
        return DROP;
}
```



```

    double u = random();
    if (u < PIE->drop_prob_) {
        PIE->accu_prob_ = 0;
        return DROP;
    } else {
        return ENQUE;
    }
}

```

=====

```

//update periodically, T_UPDATE = 16ms
calculate_drop_prob() {
    if ( (now - PIE->last_timestampe_) >= T_UPDATE &&
        PIE->active_ == ACTIVE) {
        //can be implemented using integer multiply,
        //DQ_THRESHOLD is power of 2 value
        qdelay = queue_.byte_length() * avg_dq_time_/DQ_THRESHOLD;

        p = alpha*(qdelay - QDELAY_REF) + \
            beta*(qdelay-PIE->qdelay_old_);

        //Expanding scaling range can help improve performance.
        //Please see DOCSIS-PIE design.
        //We keep it simple here
        if (PIE->drop_prob_ < 0.1%) {
            p = p/128
        } else if (PIE->drop_prob_ < 1%) {
            p = p/16;
        } else if (PIE->drop_prob_ < 10%) {
            p = p/2;
        } else {
            p = p;
        }

        if (PIE->drop_prob_ >= 10% && p > 2%) {
            p = 0.02;
        }
        PIE->drop_prob_ += p;

        //Exponentially decay drop prob when congestion goes away
        if (qdelay == 0 && PIE->qdelay_old_ == 0) {
            PIE->drop_prob_ *= 0.98;    //1- 1/64 is sufficient

```



```
    }

    //bound drop probability
    if (PIE->drop_prob_ < 0)
        PIE->drop_prob_ = 0
    if (PIE->drop_prob_ > 1)
        PIE->drop_prob_ = 1

    PIE->qdelay_old_ = qdelay;
    PIE->last_timestamp_ = now;
    if (PIE->burst_allowance_ > 0) {
        PIE->burst_allowance_ = PIE->burst_allowance_ - T_UPDATE;
    }
}

}

=====
//called on each packet departure
deque(Packet packet) {

    //dequeue rate estimation
    if (PIE->in_measurement_ == TRUE) {
        PIE->dq_count_ = packet.size() + PIE->dq_count_;
        //start a new measurement cycle if we have enough packets
        if (PIE->dq_count_ >= DQ_THRESHOLD) {
            dq_time = now - PIE->measurement_start_;
            if(PIE->avg_dq_time_ == 0) {
                PIE->avg_dq_time_ = dq_time;
            } else {
                weight = DQ_THRESHOLD/2^16
                PIE->avg_dq_time_ = dq_time*weight + PIE->avg_dq_time*(1-
weight);
            }
            PIE->in_measurement = FALSE;
        }
    }

    //start a measurement cycle if we have enough data in the queue:
    if (queue_.byte_length() >= DQ_THRESHOLD &&
        PIE->in_measurement_ == FALSE) {
        PIE->in_measurement_ = TRUE;
        PIE->measurement_start_ = now;
        PIE->dq_count_ = 0;
    }
}
```