PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem

draft-ietf-aqm-pie-07

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

Table of Contents

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 Detection
(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, including de-
randomization of coin tosses and enhanced burst protection.

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 across an arbitrary number of
            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
detail.

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. When a packet arrives, a
random decision is made regarding whether to drop the packet. The drop
probability is updated periodically based on how far the current delay
is away from the target and whether the queueing delay is currently
trending up or down. The queueing delay can be obtained using direct
measurements or using estimations calculated from the queue length and
the dequeue rate.

The detailed definition of parameters can be found in the pseudo code
section of this document (Section 11). Any state variables that PIE
maintains are noted using "PIE->". For full description of the
algorithm, one can refer to the full paper [HPSR-PIE].

```
        Random Drop
            /            --------------
    -------/  -------------->   | | | | | -------------->
         /|\                    | | | | |
          |                     --------------
          |              Queue Buffer    \
          |                     |          \
          |                     |queue      \
```

```
                     |             |length     \
                     |             |            \
                     |             \|/          \/
                     |     ----------------   ------------------
                     |     |    Drop      |   |                |
              -----<-----| Probability  |<---| Latency        |
                     |     | Calculation  |   | Calculation    |
                     |     ----------------   ------------------
```
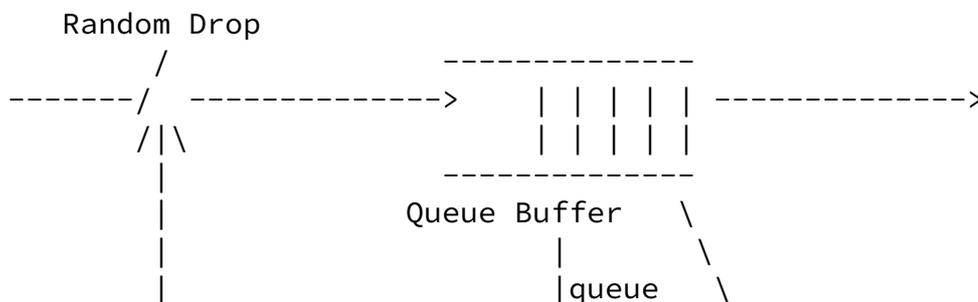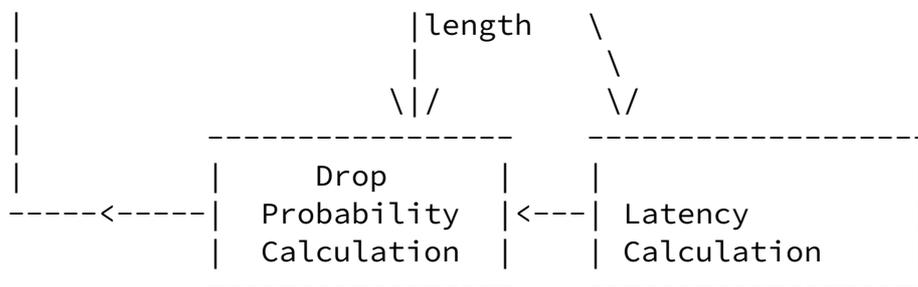
Figure 1. The PIE Structure

<u>4.1</u> Random Dropping(ECN Support is described later in this document)

PIE MUST drop a packet upon its arrival to a queue according to a drop
probability, PIE->drop_prob_, that is obtained from the drop-
probability-calculation component. The random drop is triggered by a
packet arrival before enqueueing into a queue.

    * Upon a packet enqueue, PIE MUST:

      randomly drop the packet with a probability PIE->drop_prob_.

To ensure that PIE is work conserving, we MAY bypass the random drop if
the delay sample, PIE->qdelay_old_, is smaller than half of QDELAY_REF
when the drop probability is not too high, PIE->drop_prob_ < 0.2; or if
the queue has less than a couple of packets.

    * Upon a packet enqueue, PIE MAY:

      //Safeguard PIE to be work conserving
      if ((PIE->qdelay_old_ < QDELAY_REF/2 && PIE->drop_prob_ < 0.2)
          || (queue_.byte_length() <= 2 * MEAN_PKTSIZE)) {
           return ENQUE;
      else
          randomly drop the packet with a probability PIE->drop_prob_.

PIE optionally supports ECN and see <u>Section 5.1</u>.

Pan et al.   Expires October 21, 2016   [Page 7]
```

<u>4.2</u> Drop Probability Calculation

The PIE algorithm periodically updates the drop probability based on the delay samples: not only the current delay sample but also the trend where the delay is going, up or down. This is the classical Proportional Integral (PI) controller method which is known for eliminating steady state errors. This type of controller has been studied before for controlling the queue length [PI, QCN]. PIE adopts the Proportional Integral controller for controlling delay. The algorithm also auto-adjusts the control parameters based on how heavy the congestion is, which is reflected in the current drop probability. Note that the current drop probability is a direct measure of current congestion level, no need to measure the arrival rate and departure rate mismatches.

When a congestion period goes away, we might be left with a high drop probability with light packet arrivals. Hence, 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 faster while the PI controller ensures that it would eventually reaches zero. The decay parameter of 2% gives us around 750ms time constant, a few RTT.

Specifically, the PIE algorithm MUST periodically adjust the drop probability every T_UPDATE interval:

    * MUST calculate drop probability PIE->drop_prob_ and auto-tune it
      as:

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



        if (PIE->drop_prob_ < 0.000001) {
            p /= 2048;
        } else if (PIE->drop_prob_ < 0.00001) {
            p /= 512;
        } else if (PIE->drop_prob_ < 0.0001) {
            p /= 128;
        } else if (PIE->drop_prob_ < 0.001) {
            p /= 32;
        } else if (PIE->drop_prob_ < 0.01) {
            p /= 8;
        } else if (PIE->drop_prob_ < 0.1) {
            p /= 2;
        } else {
            p = p;
```

```
        }
      PIE->drop_prob_ += p;
```

* MUST decay the drop probability exponentially:

```
    if (current_qdelay == 0 &&  PIE->qdelay_old_ == 0) {

        PIE->drop_prob_ = PIE->drop_prob_*0.98;   //1- 1/64
                                                 //is sufficient

    }
```

* MUST bound the drop probability

```
  if (PIE->drop_prob_ < 0)
          PIE->drop_prob_ = 0.0
  if (PIE->drop_prob_ > 1)
          PIE->drop_prob_ = 1.0
```

* MUST store current delay value:

```
    PIE->qdelay_old_ = current_qdelay.
```

The update interval, T_UPDATE, is defaulted to be 15ms. It MAY be
reduced on high speed links in order to provide smoother response. The
target delay value, QDELAY_REF, SHOULD be set to 15ms. Variables,
current_qdelay and PIE->qdelay_old_ represent the current and previous
samples of the queueing delay, which are calculated by the "Latency
Calculation" component (see Section 4.3). The variable current_qdelay is
actually a temporary variable while PIE->qdelay_old_ is a state variable
that PIE keeps. The drop probability is a value between 0 and 1.
However, implementations can certainly use integers.

The controller parameters, alpha and beta(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]. Note that the
above adjustment of p effectively scales the alpha and beta parameters
based on current congestion level indicated by the drop probability.

The theoretical analysis of PIE can be found in [HPSR-PIE]. As a rule of
thumb, to keep the same feedback loop dynamics, if we cut T_UPDATE in
half, we should also cut alpha by half and increase beta by alpha/4. If
the target delay is reduced, e.g. for data center use, the values of
alpha and beta SHOULD be increased by the same order of magnitude that
the target latency is reduced. For example, if QDELAY_REF is reduced

changed from 15ms to 150us, a reduction of two orders of magnitude, then alpha and beta values should be increased to alpha*100 and beta*100.

4.3 Latency Calculation

The PIE algorithm MUST use latency to calculate drop probability.

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

        current_qdelay = queue_.byte_length()/dequeue_rate;

     Details can be found in Section 5.2.

   * 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 Burst Tolerance

PIE MUST also NOT penalize short-term packet bursts [IETF-AQM]. PIE MUST allow bursts of traffic that create finite-duration events in which current queueing delay exceeds the QDELAY_REF, without triggering packet drops. A parameter, MAX_BURST, is introduced that defines the burst duration that will be protected. By default, the parameter SHOULD be set to be 150ms. For simplicity, the PIE algorithm MAY effectively round MAX_BURST up to an integer multiple of T_UPDATE.

To implement the burst tolerance function, two basic components of PIE are involved: "random dropping" and "drop probability calculation". The PIE algorithm MUST do the following:

   * In "Random Dropping" block and upon a packet arrival , PIE MUST
   check:

     Upon a packet enqueue:
     if PIE->burst_allowance_ > 0 enqueue packet;
     else randomly drop a packet with a probability PIE->drop_prob_.

     if (PIE->drop_prob_ == 0 and current_qdelay < QDELAY_REF/2 and
     PIE->qdelay_old_ < QDELAY_REF/2)

```
          PIE->burst_allowance_ = MAX_BURST;
```

   * In "Drop Probability Calculation" block, PIE MUST additionally
     calculate:

```
          PIE->burst_allowance_ = max(0,PIE->burst_allowance_ -
          T_UPDATE);
```

The burst allowance, noted by PIE->burst_allowance_, is initialized to

MAX_BURST. As long as PIE->burst_allowance_ is above zero, an incoming
packet will be enqueued bypassing the random drop process. During each
update instance, the value of PIE->burst_allowance_ is decremented by
the update period, T_UPDATE and is bottomed at 0. When the congestion
goes away, defined here as PIE->drop_prob_ equals 0 and both the current
and previous samples of estimated delay are less than half of
QDELAY_REF, PIE->burst_allowance_ is reset to MAX_BURST.

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 purposes, they are
included in this section.

5.1 ECN Support

PIE SHOULD support ECN by marking (rather than dropping) ECN capable
packets [IETF-ECN]. 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 0.1(10%).

   * To support ECN, the "random drop with a probability
     PIE->drop_prob_" function in "Random Dropping" block SHOULD be
     changed to the following:

   * Upon a packet enqueue:

```
   if rand() < PIE->drop_prob_:

      if PIE->drop_prob_ < mark_ecnth && ecn_capable_packet == TRUE:

         mark packet;

      else:

         drop packet;
```

5.2 Departure Rate Estimation

One way to calculate latency is to obtain the departure rate. 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. One MAY
measure directly at the dequeue operation. Short, non-persistent bursts
of packets result in empty queues from time to time, this would make the
measurement less accurate. PIE SHOULD measure when a sufficient data in
the buffer, i.e., when the queue length is over a certain threshold
(DQ_THRESHOLD). PIE measures how long it takes to drain DQ_THRESHOLD of
packets. More specifically, PIE MAY implement the rate estimation as
follows:

```
   current_qdelay = queue_.byte_length() *
                 PIE->avg_dq_time_/DQ_THRESHOLD;

   * Upon a packet deque:

     if PIE->in_measurement_ == FALSE and queue.byte_length() >=
     DQ_THRESHOLD:
       PIE->in_measurement_ = TRUE;
       PIE->measurement_start_ = now;
       PIE->dq_count_ = 0;

     if PIE->in_measurement_ == TRUE:
       PIE->dq_count_ = PIE->dq_count_ + deque_pkt_size;
       if PIE->dq_count_ >= DQ_THRESHOLD then
         weight = DQ_THRESHOLD/2^16
         PIE->avg_dq_time_ = (now-PIE->measurement_start_)*weight
```

```
                              + PIE->avg_dq_time_*(1-weight);
          PIE->dq_count_=0;
          PIE->measurement_start_ = now
        else
          PIE->in_measurement_ = FALSE;
```

The parameter, PIE->dq_count_, represents the number of bytes departed
since the last measurement. Once PIE->dq_count_ is over 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 sufficient data to obtain an average draining rate
but also fast enough (< 64KB) to reflect sudden changes in the draining
rate. IF DQ_THRESHOLD is smaller than 64KB, a small weight is used to
smooth out the dequeue time and obtain PIE->avg_dq_time_. The dequeue
rate is simply DQ_THRESHOLD divided by PIE->avg_dq_time_. 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 PIE-
>avg_dq_time_ is used.

           Random Drop
             / --------------
    -------/ -------------------> | | | | | -------------->
          /|\ | | | | | |
           | | --------------
           | | Queue Buffer
           | | |
           | | |queue
           | | |length
           | | |
           | \|/   \|/
           | ------------------------------
           | | Departure Rate |
        -----<-----| & Drop Probability |
                    | Calculation |

Figure 2. The Enqueue-based PIE Structure

In some platforms, enqueueing and dequeueing functions belong to
different modules that are independent of 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
reaches 0, current and previous delay samples are all below half of
QDELAY_REF.

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 in
Section 4.4 MAY be modified as follows:

    * "Random Dropping" block, PIE MAY add:

      Upon packet arrival:

      if PIE->active_ == FALSE && queue_length >= TAIL_DROP/3:

```
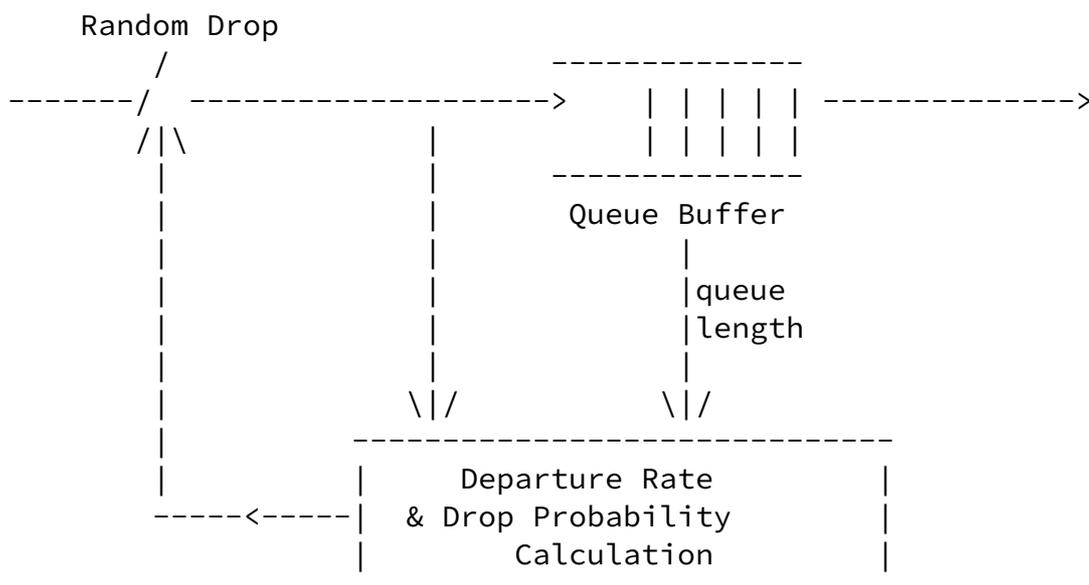            PIE->active_ = TRUE;
            PIE->burst_allowance_ = MAX_BURST;

        if PIE->burst_allowance_ > 0 enqueue packet;
        else randomly drop a packet with a probability PIE->drop_prob_.

        if (PIE->drop_prob_ == 0 and current_qdelay < QDELAY_REF/2 and
        PIE->qdelay_old_ < QDELAY_REF/2)
            PIE->active_ = FALSE;
            PIE->burst_allowance_ = MAX_BURST;

    * "Drop Probability Calculation" block, PIE MAY do the following:
        if PIE->active_ == TRUE:
            PIE->burst_allowance_ = max(0,PIE->burst_allowance_ - T_UPDATE);
```

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 value PIE->drop_prob_. 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
PIE->accu_prob_, is reset to 0 after a drop. Upon a packet arrival, PIE-
>accu_prob_ is incremented by the amount of drop probability, PIE-
>drop_prob_. If PIE->accu_prob_ is less than a low threshold, e.g. 0.85,
the arriving packet is enqueued; on the other hand, if PIE->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 PIE->accu_prob_ falls in
between the two thresholds. Since PIE->accu_prob_ is reset to 0 after a
drop, another drop will not happen until 0.85/PIE->drop_prob_ packets
later. This avoids packets being dropped too close to each other. In the
other extreme case where 8.5/PIE->drop_prob_ packets have been enqueued
without incurring a drop, PIE would force a drop in order to prevent the

Pan et al.   Expires October 21, 2016   [Page 14]

INTERNET DRAFT   PIE   April 19, 2016

drops from being spaced too far apart. Further analysis can be found in
[DOCSIS-PIE].

5.5 Cap Drop Adjustment

In the case of one single TCP flow during slow start phase in the
system, queue could quickly increase during slow start and demands high
drop probability. In some environments such as Cable Modem Speed Test,
one could not afford triggering timeout and lose throughput as
throughput is shown to customers who are testing his/her connection
speed. We MAY cap the maximum drop probability increase in each step.

    * "Drop Probability Calculation" block, PIE MAY add:

    if (PIE->drop_prob_ >= 0.1 && p > 0.02) {

      p = 0.02;

    }

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. 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 T_UPDATE 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 alpha and beta. Since
PIE's control law is robust to minor changes in alpha and beta values,
an implementation MAY choose these values to the closest multiples of 2
or 1/2 (ex: alpha=1/8, beta=1 + 1/4) 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 one variables per queue: PIE->qdelay_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
T_UPDATE, 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.

Flow queuing can also be combined with PIE to provide isolation between
flows. In this case, it is preferable to have an independent value of
drop probability per queue.   This allows each flow to receive the most
appropriate level of congestion signal, and ensures that sparse flows
are protected from experiencing packet drops. However, running the
entire PIE algorithm independently on each queue in order to calculate
the drop probability may be overkill. Furthermore, in the case that
departure rate estimation is used to predict queuing latency, it is not
possible to calculate an accurate per-queue departure rate upon which to
implement the PIE drop probability calculation. Instead, it has been
proposed ([DOCSIS_AQM]) that a single implementation of the PIE drop
probability calculation based on the overall latency estimate be used,
followed by a per-queue scaling of drop-probability based on the ratio
of queue-depth between the queue in question and the current largest
queue. This scaling is reasonably simple, and has a couple of nice
properties. One, if a packet is arriving to an empty queue, it is given
immunity from packet drops altogether, regardless of the state of the
other queues. Two, in the situation where only a single queue is in use,
the algorithm behaves exactly like the single-queue PIE algorithm.

In summary, PIE is simple enough to be implemented in both software and
hardware.

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. Security Considerations

    This document describes an active queue management algorithm based
    on implementations in Cisco products. This algorithm introduces no
    specific security exposures.

10. IANA Considerations

    There are no actions for IANA.

11. References

11.1 Normative References

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

11.2 Informative References

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

    [RFC2309]   Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering,
                S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G.,
                Patridge, C., Peterson, L., Ramakrishnan, K., Shenker, S.,
                Wroclawski, J. and Zhang, L., "Recommendations on Queue
                Management and Congestion Avoidance in the Internet",

April, 1998.

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

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

[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-02.

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

[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.
https://www.researchgate.net/publication/261134127_PIE_A_lightweight
_control_scheme_to_address_the_bufferbloat_problem?origin=mail

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

[IETF-ECN]   Briscoe, B. Kaippallimalil, J and Phaler, P.,
"Guidelines for Adding Congestion Notification to Protocols that
Encapsulate IP", draft-ietf-tsvwg-ecn-encap-guidelines.

11.3 Other References

[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.

[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

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

12. The Basic PIE pseudo Code

Configurable Parameters:
- QDELAY_REF. AQM Latency Target (default: 15ms)
- 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:15ms)

Table which stores status variables (ending with "_"):
- burst_allowance_: current burst allowance
- drop_prob_: The current packet drop probability. 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_.enque(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
      enque(Packet packet) {
            if (PIE->drop_prob_ == 0 && current_qdelay < QDELAY_REF/2
                && PIE->qdelay_old_ < QDELAY_REF/2) {
                PIE->burst_allowance_ = MAX_BURST;
            }
            if (PIE->burst_allowance_ == 0 && drop_early() == DROP) {
              drop(packet);
            } else {
              queue_.enque(packet);
            }
      }




    ===========================

      drop_early() {

            //Safeguard PIE to be work conserving
            if ( (PIE->qdelay_old_ < QDELAY_REF/2 && PIE->drop_prob_ < 0.2)
                || (queue_.byte_length() <= 2 * MEAN_PKTSIZE) ) {
                 return ENQUE;
```

```
            }

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


    ===========================
```

```
//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) {

     current_qdelay = packet.timestamp_delay();

   }

=============================
//update periodically, T_UPDATE = 15ms

   calculate_drop_prob() {

       //can be implemented using integer multiply,

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

       if (PIE->drop_prob_ < 0.000001) {
           p /= 2048;
       } else if (PIE->drop_prob_ < 0.00001) {
           p /= 512;
       } else if (PIE->drop_prob_ < 0.0001) {
           p /= 128;
       } else if (PIE->drop_prob_ < 0.001) {
           p /= 32;
       } else if (PIE->drop_prob_ < 0.01) {
           p /= 8;
       } else if (PIE->drop_prob_ < 0.1) {
           p /= 2;
       } else {
           p = p;
       }
```

```
       PIE->drop_prob_ += p;

       //Exponentially decay drop prob when congestion goes away
       if (current_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_ = 0.0
        if (PIE->drop_prob_ > 1)
                PIE->drop_prob_ = 1.0

        PIE->qdelay_old_ = current_qdelay;

        PIE->burst_allowance_ = max(0,PIE->burst_allowance_ - T_UPDATE);


    }
}
```

13. Pseudo code for PIE with optional enhancement

  Configurable Parameters:
       - QDELAY_REF. AQM Latency Target (default: 15ms)
       - MAX_BURST. AQM Max Burst Allowance (default: 150ms)
       - MAX_ECNTH. 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:15ms)
       - 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 average dequeue 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_.enque(packet). Adds packet to tail of queue_
        - queue_.deque(). 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
  enque(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_.enque(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
          && current_qdelay == 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_ < 0.2)
          || (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 = 15ms
calculate_drop_prob() {
    if ( (now - PIE->last_timestamp_) >= T_UPDATE &&
        PIE->active_ == ACTIVE) {
      //can be implemented using integer multiply,
      //DQ_THRESHOLD is power of 2 value
      current_qdelay = queue_.byte_length() * PIE-
    >avg_dq_time_/DQ_THRESHOLD;

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


      if (PIE->drop_prob_ < 0.000001) {
          p /= 2048;
      } else if (PIE->drop_prob_ < 0.00001) {
          p /= 512;
      } else if (PIE->drop_prob_ < 0.0001) {
          p /= 128;
      } else if (PIE->drop_prob_ < 0.001) {
          p /= 32;
      } else if (PIE->drop_prob_ < 0.01) {
          p /= 8;
      } else if (PIE->drop_prob_ < 0.1) {
          p /= 2;
      } else {
          p = p;
      }

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

```
        //Exponentially decay drop prob when congestion goes away
        if (current_qdelay < QDELAY_REF/2 &&  PIE->qdelay_old_ <
    QDELAY_REF/2) {
                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_ = current_qdelay;
```

```
        PIE->last_timestamp_ = now;
        PIE->burst_allowance_ = max(0,PIE->burst_allowance_ - T_UPDATE);
    }
}


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

    //deque 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 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;
        }
    }
```