            Proportional Integral Controller Enhanced (PIE):
     A Lightweight Control Scheme to Address the Bufferbloat Problem

Abstract

   Bufferbloat is a phenomenon in which excess buffers in the network
   cause high latency and latency variation. 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 latency variation degrade application performance. There
   is a pressing need to design intelligent queue management schemes
   that can control latency and latency variation, 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 queuing 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 timestamps, so it incurs very little overhead and
   is simple enough to implement in both hardware and software.

Status of This Memo

   This document is not an Internet Standards Track specification; it is
   published for examination, experimental implementation, and
   evaluation.

   This document defines an Experimental Protocol for the Internet
   community. This document is a product of the Internet Engineering
   Task Force (IETF). It represents the consensus of the IETF
   community. It has received public review and has been approved for
   publication by the Internet Engineering Steering Group (IESG). Not
   all documents approved by the IESG are a candidate for any level of
   Internet Standard; see Section 2 of RFC 7841.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc8033.

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. Because of the 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. TCP 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 queuing latency and latency variation. Designing
   a queue management scheme is a delicate balancing act: it not only
   should allow short-term bursts to smoothly pass but also should
   control the average latency in the presence of long-running greedy
   flows.

Active Queue Management (AQM) schemes could potentially solve the aforementioned problem. AQM schemes, such as Random Early Detection (RED) [RED] as suggested in [RFC2309] (which is now obsoleted by [RFC7567]), have been around for well over a decade. 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 latency 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. The more recently published RFC 7567 calls for new methods of controlling network latency.

New algorithms are beginning to emerge to control queuing latency directly to address the bufferbloat problem [CoDel]. Along these lines, Proportional Integral controller Enhanced (PIE) also aims to keep the benefits of RED, including easy implementation and scalability to high speeds. Similar to RED, PIE randomly drops an incoming packet at the onset of congestion. Congestion detection, however, is based on the queuing latency instead of the queue length (as with RED). Furthermore, PIE also uses the derivative (rate of change) of the queuing 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 latency-based AQM scheme would be applied over a Fair Queuing (FQ) structure or one of its approximate designs, Flow Queuing or Class-Based Queuing (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 or per-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 that their latency is better controlled and that 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 [RFC7806].

In October 2013, CableLabs' Data-Over-Cable Service Interface Specification 3.1 (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 [RFC8034] has improved the
original design in several areas, including derandomization of coin
tosses and enhanced burst protection.

This document describes the design of PIE and separates it into basic
elements and optional components that may be implemented to enhance
the performance of PIE.

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 latency-sensitive applications. It should follow
the general guidelines set by the AQM working group document "IETF
Recommendations Regarding Active Queue Management" [RFC7567]. More
specifically, the PIE design has the following basic criteria.

* First, queuing latency, instead of queue length, is controlled.
   Queue sizes change with queue draining rates and various flows'
   round-trip times. Latency 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
   underutilization or losing network efficiency. An early
   congestion signal could cause TCP to back off and avoid queue
   buildup. On the other hand, however, TCP's rate reduction could
   result in link underutilization. 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
   design simplicity similar to that of 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 latency, not design parameters.

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

4. The Basic PIE Scheme

As illustrated in Figure 1, PIE is comprised of three simple basic
components: a) random dropping at enqueuing, b) periodic drop
probability updates, and 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 latency is away from the target value and whether the
queuing latency is currently trending up or down. The queuing
latency 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 Appendix A of
this document ("The Basic PIE Pseudocode"). Any state variables that
PIE maintains are noted using "PIE->". For a 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    |
                    -----------------   ------------------
```
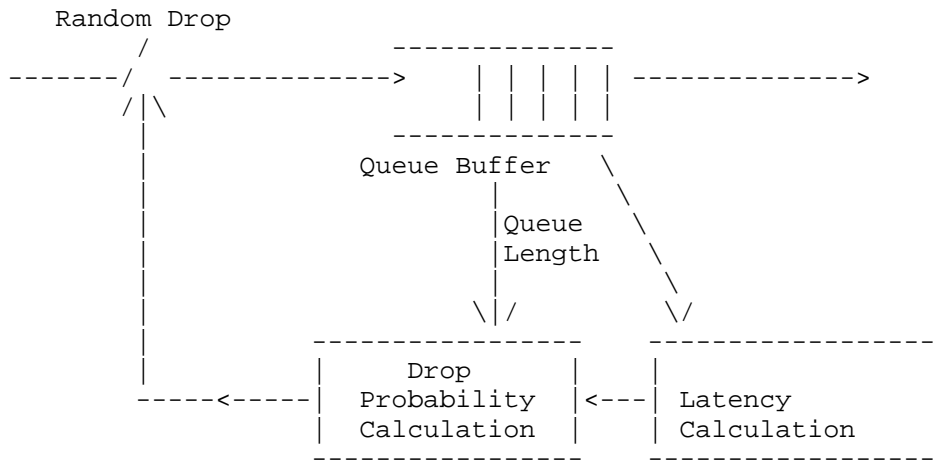
Figure 1: The PIE Structure

4.1. Random Dropping

   PIE randomly drops 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's arrival before enqueuing into a queue.

   * Upon a packet enqueue:

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

   To ensure that PIE is "work conserving", we bypass the random drop if
   the latency sample, PIE->qdelay_old_, is smaller than half of the
   target latency value (QDELAY_REF) when the drop probability is not
   too high (i.e., PIE->drop_prob_ < 0.2), or if the queue has less than
   a couple of packets.

   * Upon a packet enqueue, PIE does the following:

      //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 of
         PIE->drop_prob_.

   PIE optionally supports Explicit Congestion Notification (ECN); see
   Section 5.1.

4.2. Drop Probability Calculation

   The PIE algorithm periodically updates the drop probability based on
   the latency samples -- not only the current latency sample but also
   whether the latency is trending 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 PI controller for controlling latency. 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 the current
   congestion level; there is no need to measure the arrival rate and
   dequeue rate mismatches.

   When a congestion period ends, we might be left with a high drop
   probability with light packet arrivals. Hence, the PIE algorithm
   includes a mechanism by which the drop probability decays

exponentially (rather than linearly) when the system is not
congested. This would help the drop probability converge to 0 more
quickly, while the PI controller ensures that it would eventually
reach zero. The decay parameter of 2% gives us a time constant
around 50 * T_UPDATE.

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

*   calculate drop probability PIE->drop_prob_, and autotune it as
    follows:

```
        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;
```

*   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
        }
```

*   bound the drop probability:

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

   * store the current latency value:

          PIE->qdelay_old_ = current_qdelay.

   The update interval, T_UPDATE, is defaulted to be 15 milliseconds.
   It MAY be reduced on high-speed links in order to provide smoother
   response. The target latency value, QDELAY_REF, SHOULD be set to 15
   milliseconds. The variables current_qdelay and PIE->qdelay_old_
   represent the current and previous samples of the queuing latency,
   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 (expressed in 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 the 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 latency is reduced, e.g., for data-center
   use, the values of alpha and beta should be increased by the same
   order of magnitude by which the target latency is reduced. For
   example, if QDELAY_REF is reduced and changed from 15 milliseconds to
   150 microseconds -- 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 uses latency to calculate drop probability in one
   of two ways:

   * It estimates the current queuing latency using Little's law (see
      Section 5.2 for details):

          current_qdelay = queue_.byte_length()/dequeue_rate;

   * It may use other techniques for calculating queuing latency, e.g.,
      time-stamp the packets at enqueue, and use the timestamps to
      calculate latency during dequeue.

4.4. Burst Tolerance

   PIE does not penalize short-term packet bursts as suggested in
   [RFC7567]. PIE allows bursts of traffic that create finite-duration
   events in which current queuing latency exceeds QDELAY_REF without
   triggering packet drops. This document introduces a parameter called
   "MAX_BURST"; MAX_BURST defines the burst duration that will be
   protected. By default, the parameter SHOULD be set to 150
   milliseconds. 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 does the following:

   * In the "random dropping" block and upon packet arrival, PIE checks
      the following:

      Upon a packet enqueue:
         if PIE->burst_allowance_ > 0
            enqueue packet;
         else
            randomly drop a packet with a probability of
            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 the "drop probability calculation" block, PIE additionally
      calculates:

      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
   latency are less than half of QDELAY_REF -- PIE->burst_allowance_ is
   reset to MAX_BURST.

5. Optional Design Elements of PIE

   There are several enhancements that are added to further augment the
   performance of the basic algorithm. For purposes of clarity, they
   are included in this section.

5.1. ECN Support

   PIE MAY support ECN by marking (rather than dropping) ECN-capable
   packets [ECN]. This document introduces an additional threshold
   called "mark_ecnth", which acts as a safeguard: if the calculated
   drop probability exceeds mark_ecnth, PIE reverts to packet-dropping
   for ECN-capable packets. The variable mark_ecnth SHOULD be set to
   0.1 (10%).

   * To support ECN, the "random drop with a probability of
      PIE->drop_prob_" function in the "random dropping" block is
      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. Dequeue Rate Estimation

   Using timestamps, a latency sample can only be obtained when a packet
   reaches the head of a queue. When a quick response time is desired
   or a direct latency sample is not available, one may obtain latency
   through measuring the dequeue rate. The draining rate of a queue in
   the network often varies either because other queues are sharing the
   same link or because 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 only measures latency when there is
   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 packets. More specifically, the rate estimation
can be implemented as follows:

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

* Upon a packet dequeue:

```
    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. It is recommended
that the threshold be set to 16 KB, assuming a typical packet size of
around 1 KB or 1.5 KB. This threshold would allow sufficient data to
obtain an average draining rate but would also be fast enough (< 64
KB) to reflect sudden changes in the draining rate. If DQ_THRESHOLD
is smaller than 64 KB, 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
           |                   |          |
           |                 \ | /      \ | /
           |          ------------------------------
           |          |       Dequeue Rate         |
           -----<-----|     & Drop Probability      |
                      |        Calculation          |
                      ------------------------------
```
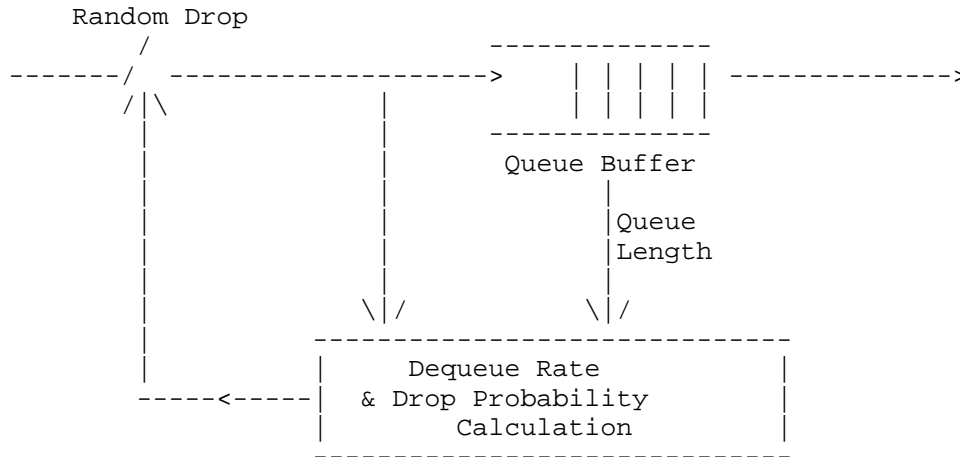
Figure 2: The Enqueue-Based PIE Structure

In some platforms, enqueuing and dequeuing functions belong to
different modules that are independent of each other. In such
situations, a pure enqueue-based design can be developed. An
enqueue-based design is depicted in Figure 2. The dequeue 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 dequeued packets = the number of
enqueued packets minus the number of remaining packets in the queue.
In this design, everything can be triggered by packet arrival,
including the background update process. The design complexity here
is similar to the original design.

5.3. Setting PIE Active and Inactive

Traffic naturally fluctuates in a network. It would be preferable
not to unnecessarily drop packets due to a spurious uptick in queuing
latency. PIE has an optional feature of automatically becoming
active/inactive. To implement this feature, PIE may choose to only
become active (from inactive) when the buffer occupancy is over a
certain threshold, which may be set to 1/3 of the tail drop
threshold. PIE becomes inactive when congestion ends; i.e., when the
drop probability reaches 0, current and previous latency samples are
all below half of QDELAY_REF.

Ideally, PIE should become active/inactive based on latency.
However, calculating latency when PIE is inactive would introduce
unnecessary packet-processing overhead. Weighing the trade-offs,
we decided to compare against the tail drop threshold to keep things
simple.

When PIE optionally becomes active/inactive, the burst protection
logic described in Section 4.4 is modified as follows:

*   "Random dropping" block: PIE adds the following:

    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 of
       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 does the following:
    if PIE->active_ == TRUE:
       PIE->burst_allowance_ =
          max(0,PIE->burst_allowance_ - T_UPDATE);

5.4. Derandomization

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 the real drop percentage to temporarily
deviate from the intended value PIE->drop_prob_. In certain
scenarios, such as a small number of simultaneous TCP flows, these
deviations can cause significant deviations in link utilization and
queuing latency. PIE may use a derandomization mechanism to avoid
such situations. A parameter called "PIE->accu_prob_" is reset to 0
after a drop. Upon 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, and the queue is congested, the arrival packet
is forced to be dropped. A packet is only randomly dropped if
PIE->accu_prob_ falls 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
drops from being spaced too far apart. Further analysis can be found
in [RFC8034].

5.5. Cap Drop Adjustment

In the case of a single TCP flow, during the slow-start phase the
queue could quickly increase, which could result in a very rapid
increase in drop probability. In order to prevent an excessive
ramp-up that could negatively impact the throughput in this scenario,
PIE can cap the maximum drop probability increase in each step.

*   "Drop probability calculation" block: PIE adds the following:

    ```
    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 enqueuing can be
readily retrofitted into existing 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 the 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 (e.g.,
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 three variables per queue:
burst_allowance_, PIE->drop_prob_, and 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 single multiply operation. Note that many
network devices keep track of an interface's departure rate. In this
case, PIE might be able to reuse this information and simply skip the
third step of the algorithm; hence, it would incur no extra cost. If
a platform already leverages packet timestamps for other purposes,
PIE can make use of these packet timestamps for latency calculation
instead of estimating the departure rate.

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 where 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:

* 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.

* 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. Scope of Experimentation

   The design of the PIE algorithm is presented in this document. The
   PIE algorithm effectively controls the average queuing latency to a
   target value. The following areas can be used for further study and
   experimentation:

   * Autotuning of target latency without losing utilization.

   * Autotuning for the average round-trip time of traffic.

   * The proper threshold to transition smoothly between ECN marking
      and dropping.

   * The enhancements described in Section 5, which can be used in
      experiments to see if they would be of more value in the real
      world. If so, they will be incorporated into the basic PIE
      algorithm.

   * The PIE design, which is separated into the data path and the
      control path. The control path can be implemented in software.
      Field tests of other control laws can be performed to experiment
      with further improvements to PIE's performance.

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

8. Incremental Deployment

   From testbed experiments and large-scale simulations of PIE so far,
   PIE has been shown to be effective across a diverse range of network
   scenarios. There is no indication that PIE would be harmful to
   deploy.

   The PIE scheme can be independently deployed and managed without a
   need for interoperability between different network devices. In
   addition, any individual buffer queue can be incrementally upgraded
   to PIE, as it can coexist with existing AQM schemes such as
   Weighted RED (WRED).

   PIE is intended to be self-configuring. Users should not need to
   configure any design parameters. Upon installation, the two
   user-configurable parameters -- QDELAY_REF and MAX_BURST -- will be
   defaulted to 15 milliseconds and 150 milliseconds for non-data-center
   network devices and to 15 microseconds and 150 microseconds for
   data-center switches, respectively.

Since the data path of the algorithm needs only a simple coin toss
and the control-path calculation happens in a much slower time scale,
we don't foresee any scaling issues associated with the algorithm as
the link speed scales up.

9. Security Considerations

This document describes PIE, an active queue management algorithm
based on implementations in different products. The PIE algorithm
introduces no specific security exposures.

10. References

10.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <http://www.rfc-editor.org/info/rfc2119>.

10.2. Informative References

   [RFC970]   Nagle, J., "On Packet Switches With Infinite Storage",
              RFC 970, DOI 10.17487/RFC0970, December 1985,
              <http://www.rfc-editor.org/info/rfc970>.

   [RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering,
              S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G.,
              Partridge, C., Peterson, L., Ramakrishnan, K., Shenker,
              S., Wroclawski, J., and L. Zhang, "Recommendations on
              Queue Management and Congestion Avoidance in the
              Internet", RFC 2309, DOI 10.17487/RFC2309, April 1998,
              <http://www.rfc-editor.org/info/rfc2309>.

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

   [RFC7806] Baker, F. and R. Pan, "On Queuing, Marking, and Dropping",
              RFC 7806, DOI 10.17487/RFC7806, April 2016,
              <http://www.rfc-editor.org/info/rfc7806>.

   [RFC8034]   White, G. and R. Pan, "Active Queue Management (AQM) Based
               on Proportional Integral Controller Enhanced (PIE) for
               Data-Over-Cable Service Interface Specifications (DOCSIS)
               Cable Modems", RFC 8034, DOI 10.17487/RFC8034,
               February 2017, <http://www.rfc-editor.org/info/rfc8034>.

   [CBQ]   Cisco, "Class-Based Weighted Fair Queueing",
               <http://www.cisco.com/en/US/docs/ios/12_0t/12_0t5/
               feature/guide/cbwfq.html>.

   [CoDel]   Nichols, K. and V. Jacobson, "Controlling Queue Delay",
               Communications of the ACM, Volume 55, Issue 7, pp. 42-50,
               DOI 10.1145/2209249.2209264, July 2012.

   [DOCSIS_3.1]
               CableLabs, "MAC and Upper Layer Protocols Interface
               Specification", DOCSIS 3.1, January 2017,
               <https://apps.cablelabs.com/specification/
               CM-SP-MULPIv3.1>.

   [DOCSIS-AQM]
               White, G., "Active Queue Management in DOCSIS 3.x Cable
               Modems", May 2014, <http://www.cablelabs.com/wp-content/
               uploads/2014/06/DOCSIS-AQM_May2014.pdf>.

   [ECN]   Briscoe, B., Kaippallimalil, J., and P. Thaler,
               "Guidelines for Adding Congestion Notification to
               Protocols that Encapsulate IP", Work in Progress,
               draft-ietf-tsvwg-ecn-encap-guidelines-07, July 2016.

   [HPSR-PIE] Pan, R., Natarajan, P., Piglione, C., Prabhu, M.S.,
               Subramanian, V., Baker, F., and B. Ver Steeg, "PIE: A
               lightweight control scheme to address the bufferbloat
               problem", IEEE HPSR, DOI 10.1109/HPSR.2013.6602305, 2013,
               <https://www.researchgate.net/publication/
               261134127_PIE_A_lightweight_control_scheme_to_address_
               the_bufferbloat_problem?origin=mail>.

   [PI]   Hollot, C.V., Misra, V., Towsley, D., and W. Gong, "On
               designing improved controllers for AQM routers supporting
               TCP flows", INFOCOM 2001, DOI 10.1109/INFCOM.2001.916670,
               April 2001.

   [QCN]   IEEE, "IEEE Standard for Local and Metropolitan Area
               Networks--Virtual Bridged Local Area Networks -
               Amendment: 10: Congestion Notification", IEEE 802.1Qau,
               <http://www.ieee802.org/1/pages/802.1au.html>.

   [RED]   Floyd, S. and V. Jacobson, "Random Early Detection (RED)
                Gateways for Congestion Avoidance", IEEE/ACM Transactions
                on Networking, Volume 1, Issue 4, DOI 10.1109/90.251892,
                August 1993.

   [TCP-Models]
                Misra, V., Gong, W., and D. Towsley, "Fluid-based analysis
                of a network of AQM routers supporting TCP flows with an
                application to RED", SIGCOMM 2000, Volume 30, Issue 4,
                pp. 151-160, DOI 10.1145/347057.347421, October 2000.

Appendix A. The Basic PIE Pseudocode

    Configurable parameters:
       - QDELAY_REF. AQM Latency Target (default: 15 milliseconds)
       - MAX_BURST. AQM Max Burst Allowance (default: 150 milliseconds)

    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: 15 milliseconds)

    Table that 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 pseudocode

   deque(Packet packet) {

     current_qdelay = packet.timestamp_delay();
   }

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

 //Update periodically, T_UPDATE = 15 milliseconds

   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);
     }
  }
```

Appendix B. Pseudocode for PIE with Optional Enhancement

    Configurable parameters:
       - QDELAY_REF. AQM Latency Target (default: 15 milliseconds)
       - MAX_BURST. AQM Max Burst Allowance (default: 150 milliseconds)
       - 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: 15 milliseconds)
       - TAIL_DROP: the tail drop threshold (max allowed queue depth)
          for the queue

    Table that 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 one needs 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 = 15 milliseconds
    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) {

        //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 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;
        }
      }
```

Contributors

    Bill Ver Steeg
    Comcast Cable
    Email: William_VerSteeg@comcast.com

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

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

    Vijay Subramanian*
    PLUMgrid, Inc.
    350 Oakmead Parkway
    Suite 250
    Sunnyvale, CA 94085
    United States of America
    Email: vns@plumgrid.com
    * Formerly at Cisco Systems

Authors' Addresses

   Rong Pan
   Cisco Systems
   3625 Cisco Way
   San Jose, CA 95134
   United States of America

   Email: ropan@cisco.com

   Preethi Natarajan
   Cisco Systems
   725 Alder Drive
   Milpitas, CA 95035
   United States of America

   Email: prenatar@cisco.com

   Fred Baker
   Santa Barbara, CA 93117
   United States of America

   Email: FredBaker.IETF@gmail.com

   Greg White
   CableLabs
   858 Coal Creek Circle
   Louisville, CO 80027
   United States of America

   Email: g.white@cablelabs.com