Internet Draft
Active Queue Management
Working Group

Expires: April 30, 2015 October 27, 2014

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

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

draft-ietf-aqm-pie-00

## 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.

We present here a lightweight design, 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

## 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. However, the above solution of large buffer fails to take into account the nature of the TCP protocol, the dominant transport protocol running in the Internet. 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 when long-term congestion persists.

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

Recently, a new trend has emerged to control queueing latency directly to address the bufferbloat problem [CoDel]. Although following the new trend, PIE also aims to keep the benefits of RED: such as easy to implement and scalable to high speeds. Similar to RED, PIE randomly drops a 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 latency moving trends: latency increasing or decreasing, to help determine congestion levels. The design parameters of PIE are chosen via 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, we assume any delay-based AQM scheme would be applied over a Fair Queueing (FQ) structure or its approximate design, 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]. These designs help flows/classes achieve max-min fairness and help mitigate bias against long flows with long round trip times(RTT). Any AQM scheme that is built on top of FQ or CBQ could benefit from these advantages. Furthermore, we believe that 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, we need to ensure their latency is better controlled and their fairness is not worse than those under the standard DropTail or RED design.

In October 2013, CableLabs' DOCSIS 3.1 specification [DOCSIS_3.1] mandates that cable modems implement a specific variant of the PIE design as the active queue management algorithm. In addition to cable specific improvements, the PIE design in DOCSIS 3.1 [DOCSIS-PIE] has improved the original design in several areas: de-randomization of coin tosses, enhanced burst protection and expanded range of auto-tuning.

The previous draft of PIE describes the overall design goals, system elements and implementation details of PIE. It also includes various design considerations: such as how auto-tuning can be done. This draft incorporates aforementioned DOCSIS-PIE improvements and integrate them into the PIE design. We also discusses a pure enque-based design where all the operations can be triggered by a packet arrival.

## 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

We explore a queue management framework where we aim to improve the performance of interactive and delay-sensitive applications. The design of our scheme follows a few basic criteria.

* First, we directly control queueing latency instead of controlling queue length. Queue sizes change with queue draining

rates and various flows' round trip times. Delay bloat is the real issue that we need to address as it impairs real time applications. If latency can be controlled, bufferbloat is not an issue. As a matter of fact, we would allow more buffers for sporadic bursts as long as the latency is under control.

* Secondly, we aim 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. The wide adoption of RED over a variety of network devices is a testament to the power of simple random early dropping/marking. We strive to maintain similar design simplicity.

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

In the following, we will elaborate on the design of PIE and its operation.

## 4. The BASIC PIE Scheme

As illustrated in Fig. 1, our scheme conceptually comprises three simple components: a) random dropping at enqueing; b) periodic drop probability update; c) dequeing rate estimation. The following sections describe these components in further detail, and explain how they interact with each other.

### 4.1 Random Dropping

Like any state-of-the-art AQM scheme, PIE would drop packets randomly according to a drop probability, p, that is obtained from the drop-probability-calculation component:

* upon a packet arrival

    randomly drop a packet with a probability p.

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

Figure 1. The PIE Structure

## 4.2 **Drop Probability Calculation**

The PIE algorithm periodically updates the drop probability as follows:

* estimate current queueing delay using Little's law:

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

* calculate drop probability p as:

```
p = p + alpha*(est_del-target_del) + beta*(est_del-est_del_old);

est_del_old = est_del.
```

Here, the current queue length is denoted by qlen. The draining rate of the queue, depart_rate, is obtained from the departure-rate-estimation block.  Variables, est_del and est_del_old, represent the current and previous estimation of the queueing delay. The target latency value is expressed in target_del.  The update interval is denoted as Tupdate.

Note that the calculation of drop probability is based not only on the current estimation of the queueing delay, but also on the direction

where the delay is moving, i.e., whether the delay is getting longer or shorter. This direction can simply be measured as the difference between est_del and est_del_old. This is the classic Proportional Integral controller design that is adopted here for controlling queueing latency. The controller parameters, in the unit of hz, are designed using feedback loop analysis where TCP's behaviors are modeled using the results from well-studied prior art[TCP-Models].

We would like to point out that this type of controller has been studied before for controlling the queue length [PI, QCN]. PIE adopts the Proportional Integral controller for controlling delay and makes the scheme auto-tuning. The theoretical analysis of PIE is under paper submission and its reference will be included in this draft once it becomes available. Nonetheless, we will discuss the intuitions for these parameters in Section 5.

## 4.3 Departure Rate Estimation

The draining rate of a queue in the network often varies either because other queues are sharing the same link, or the link capacity fluctuates. Rate fluctuation is particularly common in wireless networks. Hence, we decide to measure the departure rate directly as follows.

```
    * we are in a measurement cycle if we have enough data in the queue:

        qlen > dq_threshold

    * if in a measurement cycle:

        upon a packet departure

        dq_count = dq_count + deque_pkt_size;

    * if dq_count > dq_threshold then

        depart_rate = dq_count/(now-start);

        dq_count = 0;

        start = now;
```

We only measure the departure rate when there are sufficient data in the

buffer, i.e., when the queue length is over a certain threshold, deq_threshold. Short, non-persistent bursts of packets result in empty queues from time to time, this would make the measurement less accurate. The parameter, dq_count, represents the number of bytes departed since the last measurement. Once dq_count is over a certain threshold, deq_threshold, we obtain a measurement sample. The threshold is recommended to be set to 16KB assuming a typical packet size of around 1KB or 1.5KB. This threshold would allow us a long enough period to obtain an average draining rate but also fast enough to reflect sudden changes in the draining rate. Note that this threshold is not crucial for the system's stability.

## 5. Design Enhancement

The above three components form the basis of the PIE algorithm. There are several enhancements that we add to further augment the performance of the basic algorithm. For clarity purpose, we include them here in this section.

### 5.1 Turning PIE on and off

Traffic naturally fluctuates in a network. We would not want to unnecessarily drop packets due to a spurious uptick in queueing latency. If PIE is not active, we would only turn it on when the buffer occupancy is over a certain threshold, which we set to 1/3 of the queue buffer size. If PIE is on, we would turn it off when congestion is over, i.e. when the drop probability, queue length and estimated queue delay all reach 0.

### 5.2 Auto-tuning of PIE's control parameters

While the formal analysis can be found in [HPSR], we would like to discuss the intuitions regarding how to determine the key control parameters of PIE. Although the PIE algorithm would set them automatically, they are not meant to be magic numbers. We hope to give enough explanations here to help demystify them so that users can experiment and explore on their own.

As it is obvious from the above, the crucial equation in the PIE algorithm is

```
     p = p + alpha*(est_del-target_del) + beta*(est_del-est_del_old).
```

The value of alpha determines how the deviation of current latency from the target value affects the drop probability.  The beta term exerts additional adjustments depending on whether the latency is trending up or down. Note that the drop probability is reached incrementally, not

through a single step. To avoid big swings in adjustments which often leads to instability, we would like to tune p in small increments. Suppose that p is in the range of 1%. Then we would want the value of alpha and beta to be small enough, say 0.1%, adjustment in each step. If p is in the higher range, say above 10%, then the situation would warrant a higher single step tuning, for example 1%. There are could be several regions of these tuning, extendable all the way to 0.001% if needed. Finally, the drop probability would only be stabilized when the latency is stable, i.e. est_del equals est_del_old; and the value of the latency is equal to target_del. The relative weight between alpha and beta determines the final balance between latency offset and latency jitter.

The update interval, Tupdate, also plays a key role in stability. Given the same alpha and beta values, the faster the update is, the higher the loop gain will be. As it is not showing explicitly in the above equation, it can become an oversight. Notice also that alpha and beta have a unit of hz.

## 5.3 Handling Bursts

Although we aim to control the average latency of a congested queue, the scheme should allow short term bursts to pass through without hurting them. We would like to discuss how PIE manages bursts in this section when it is active.

Bursts are well tolerated in the basic scheme for the following reasons: first, the drop probability is updated periodically. Any short term burst that occurs within this period could pass through without incurring extra drops as it would not trigger a new drop probability calculation. Secondly, PIE's drop probability calculation is done incrementally. A single update would only lead to a small incremental change in the probability. So if it happens that a burst does occur at the exact instant that the probability is being calculated, the incremental nature of the calculation would ensure its impact is kept small.

Nonetheless, we would like to give users a precise control of the burst. We introduce a parameter, max_burst, that is similar to the burst tolerance in the token bucket design. By default, the parameter is set to be 150ms. Users can certainly modify it according to their application scenarios. The burst allowance is added into the basic PIE design as follows:

```
    * if PIE_active == FALSE

        burst_allowance = max_burst;
```

```
    * upon packet arrival

        if burst_allowance > 0 enqueue packet;

    * upon probability update when PIE_active == TRUE

        burst_allowance = burst_allowance - Tupdate;
```

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

### 5.4 De-randomization

Although PIE adopts random dropping to achieve latency control, 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 deviate from the intended drop probability p. PIE introduces a de-randomization mechanism to avoid such scenarios. We keep a parameter called accu_prob, which is reset to 0 after a drop. Upon a packet arrival, accu_prob is incremented by the amount of drop probability, p. If accu_prob is less than a low threshold, e.g. 0.85, we enque the arriving packet; on the other hand, if accu_prob is more than a high threshold, e.g. 8.5, we force a packet drop. We would only randomly drop a packet if accu_prob falls in between the two thresholds. Since accu_prob is reset to 0 after a drop, another drop will not happen until 0.85/p packets later. This avoids packets are dropped too close to each other. In the other extreme case where 8.5/p packets have been enqued without incurring a drop, PIE would force a drop that prevents much fewer drops than desired. Further analysis can be found in [AQM DOCSIS].

## 6. Implementation and Discussions

PIE can be applied to existing hardware or software solutions. In this section, we discuss the implementation cost of the PIE algorithm. There are three steps involved in PIE as discussed in Section 4. We examine their complexities as follows.

Upon packet arrival, the algorithm simply drops a packet randomly based

on the drop probability p. This step is straightforward and requires no packet header examination and manipulation. Besides, since no per packet overhead, such as a timestamp, is required, there is no extra memory requirement. Furthermore, the input side of a queue is typically under software control while the output side of a queue is hardware based. Hence, a drop at enqueueing can be readily retrofitted into existing hardware or software implementations.

The drop probability calculation is done in the background and it occurs every Tudpate interval. Given modern high speed links, this period translates into once every tens, hundreds or even thousands of packets. Hence the calculation occurs at a much slower time scale than packet processing time, at least an order of magnitude slower. The calculation of drop probability involves multiplications using alpha and beta. Since the algorithm is not sensitive to the precise values of alpha and beta, we can choose the values, e.g. alpha= 0.25 and beta= 2.5 so that multiplications can be done using simple adds and shifts. As no complicated functions are required, PIE can be easily implemented in both hardware and software. The state requirement is only two variables per queue: est_del and est_del_old. Hence the memory overhead is small.

In the departure rate estimation, PIE uses a counter to keep track of the number of bytes departed for the current interval. This counter is incremented per packet departure. Every Tupdate, PIE calculates latency using the departure rate, which can be implemented using a multiplication. Note that many network devices keep track 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. We also understand that in some software implementations, time-stamped are added for other purposes. In this case, we can also make use of the time-stamps and bypass the departure rate estimation and directly used the timestamp information in the drop probability calculation.

In some platforms, enqueueing and dequeueing functions belong to different modules that are independent to each other. In such situations, a pure enque-based design is preferred. As shown in Figure 2, we depict a enque-based design. 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 extra 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.

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

Figure 2. The Enque-based PIE Structure

In summary, the state requirement for PIE is limited and computation overheads are small. Hence, PIE is simple to be implemented. In addition, since PIE does not require any user configuration, it does not impose any new cost on existing network management system solutions. SFQ can be combined with PIE to provide further improvement of latency for various flows with different priorities. However, SFQ requires extra queueing and scheduling structures. Whether the performance gain can justify the design overhead needs to be further investigated.

## 7. Incremental Deployment

One nice property of the AQM design is that it can be independently designed and operated without the requirement of being inter-operable.

Although all network nodes can not be changed altogether to adopt latency-based AQM schemes, we envision a gradual adoption which would eventually lead to end-to-end low latency service for real time applications.

## 8. IANA Considerations

There are no actions for IANA.

## 9. References

### 9.1 Normative References

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

### 9.2 Informative References

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

### 9.3 Other References

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

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

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

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

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

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

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

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

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

## Authors' Addresses

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

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

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

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

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

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

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

Greg White
CableLabs
858 Coal Creek Circle

Louisville, CO 80027, USA
Email: g.white@cablelabs.com

* Formerly at Cisco Systems

## 10. The PIE pseudo Code

```
  Configurable Parameters:
       - QDELAY_REF. AQM Latency Target (default: 16ms)
       - BURST_ALLOWANCE. AQM Latency Target (default: 150ms)

  Internal Parameters:
       - Weights in the drop probability calculation (1/s):
         alpha (default: 1/8), beta(default: 1+1/4)
       - DQ_THRESHOLD (in bytes, default: 2^14 (in a power of 2) )
       - T_UPDATE: a period to calculate drop probability (default:16ms)
       - QUEUE_SMALL = (1/3) * Buffer limit in bytes


  Table which stores status variables (ending with "_"):
       - active_: INACTIVE/ACTIVE
       - busrt_count: current burst_count
       - 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
       - qlen_old_: The previous sample of queue length
       - dq_count_, measurement_start_, in_measurement_,
         avg_dq_time. variables for measuring avg_dq_rate_.

  Public/system functions:
       - queue_.  Holds the pending packets.
       - drop(packet).  Drops/discards a packet
       - now().  Returns the current time
       - random(). Returns a uniform r.v. in the range 0 ~ 1
       - queue_.is_full(). Returns true if queue_ is full
       - queue_.byte_length(). Returns current queue_ length in bytes
       - 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


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

  enque(Packet packet) {
       if (queue_.is_full()) {
        drop(packet);
       } else if (PIE->active_ == TRUE && drop_early() == TRUE
```

```
                  && BURST_count <= 0) {
        drop(packet);
       } else {
        queue_.enque(packet);
       }

       //If the queue is over a certain threshold, turn on PIE
       if (PIE->active_ == INACTIVE
           && queue_.byte_length() >= QUEUE_SMALL) {
            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_count = BURST_ALLOWANCE;
       }

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


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

  drop_early() {

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

      //Random drop
      accu_prob += drop_prob;
      if (accu_prob < 0.85)
          return ENQUE;
      if (accu_prob < 8.5)
          return DROP;
      double u = random();
```

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




============================
 //update periodically, T_UPDATE = 16ms
 status_update(state) {
     if ( (now - last_timestampe) >= T_UPDATE) {
       //can be implemented using integer multiply,
       //DQ_THRESHOLD is power of 2 value
       qdelay = queue_.byte_length() * avg_dqtime/DQ_THRESHOLD;
       if (PIE->drop_prob_ < 0.1%) {
            PIE->drop_prob_ += alpha*(qdelay - QDELAY_REF)/128
                               + beta*(delay-PIE->qdelay_old_)/128;
       } else if (PIE->drop_prob_ < 1%) {
            PIE->drop_prob_ += alpha*(qdelay - QDELAY_REF)/16
                               + beta*(delay-PIE->qdelay_old_)/16;
       } else if (PIE->drop_prob_ < 10%) {
            PIE->drop_prob_ += alpha*(qdelay - QDELAY_REF)/2
                               + beta*(delay-PIE->qdelay_old_)/2;
       } else {
            PIE->drop_prob_ += alpha*(qdelay - QDELAY_REF)
                             + beta*(delay-PIE->qdelay_old_);
       }
       PIE->qdelay_old_ = qdelay;
       PIE->last_timestamp_ = now;
       if (burst_count > 0) {
        burst_count = burst_count - BURST_ALLOWANCE
       }
    }
}


==========================
  deque(Packet packet) {

     //dequeue rate estimation
     if (PIE->in_measurement_ == TRUE) {
          dq_count = packet->bytelen() + dq_count;
          if ( dq_count >= DQ_THRESHOLD) {
            dq_time = now - PIE->measurement_start_;
            if(PIE->avg_dq_time_ = 0) {
```

```
              PIE->avg_dq_time_ = dq_time;
            } else {
              PIE->avg_dq_time_ = dq_time*1/4 + tmp_avg_dqtime*3/4;
            }
            PIE->in_measurement = FALSE;
          }
     }

     if (queue_.byte_length() >= DQ_THRESHILD &&
         PIE->in_measurement == FALSE) {
            PIE->in_measurement_ = TRUE;
            PIE->measurement_start_ = now;
            PIE->dq_count_ = 0;
     }
  }
```