A tutorial on the implementation of queue disciplines (qdiscs) in Linux kernel.

TABLE OF CONTENT

ABSTRACT

Packet queues are a core component of any network stack or device. They allow for asynchronous modules to communicate, increase performance and have the side effect of impacting latency. This documentation aims to explain where IP packets are queued in the Linux network stack, how interesting new latency reducing and increased throughput and fairness features such as RED, ARED , CODEL etc. and how to control buffering for reduced latency.

INTRODUCTION

Packets received by the Traffic Control layer for transmission to a netdevice can be passed to a queueing discipline (queue disc) to perform scheduling and policing. The *Linux* term “qdisc” corresponds to what NS-3 calls a “queue disc”. A netdevice can have a single (root) qdisc installed on it. Installing a qdisc on a netdevice is not mandatory. If a netdevice does not have a qdisc installed on it, the traffic control layer sends the packets directly to the netdevice. This is the case, for instance, of the loopback netdevice.

Qdisc configuration vary from qdisc to qdisc. A typical taxonomy divides qdiscs in classful (i.e., support classes) and classless (i.e., do not support classes). More recently, after the appearance of multi-queue devices (such as Wi-Fi), some multi-queue aware qdiscs have been introduced. Multi-queue aware qdiscs handle as many queues (or qdiscs – without using classes) as the number of transmission queues used by the device on which the qdisc is installed. An attempt is made, also, to enqueue each packet in the “same” queue both within the qdisc and within the device.

The traffic control layer interacts with a qdisc in a simple manner: after requesting to enqueue a packet, the traffic control layer requests the qdisc to “run”, i.e., to dequeue a set of packets, until a predefined number of packets is dequeued or the netdevice stops the qdisc. A netdevice shall stop the qdisc when its transmission queue does not have room for another packet. Also, a netdevice shall wake the qdisc when it detects that there is room for another packet in its transmission queue, but the transmission queue is stopped. Waking a qdisc is equivalent to make it run.

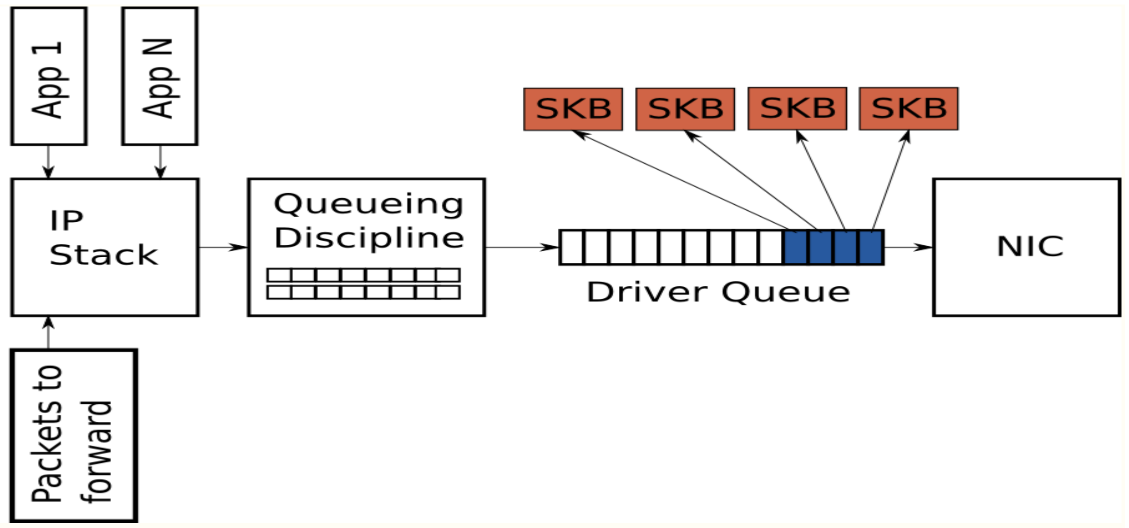


Figure:*Simplified high level overview of the queues on the transmit path of the Linux network stack*

Image Source: Articles by Dan Semion
(https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/)

❑ **The following identities hold:**

- dropped = dropped before enqueue + dropped after dequeue
- received = dropped before enqueue + enqueued
- queued = enqueued - dequeued
- sent = dequeued - dropped after dequeue (- 1 if there is a requeued packet)

As in Linux, queue discs may be simple queues or may be complicated hierarchical structures. A queue disc may contain distinct elements:

- queues, which actually store the packets waiting for transmission
- filters, which determine the queue or class which a packet is destined to

1> RED Implementation in linux kernel

❑ Introduction:

Random Early Detection (RED) algorithm was first proposed by Sally Floyd and Van Jacobson for Active Queue Management (AQM). It is used for congestion avoidance for packet switched networks. It is claimed that RED is able to avoid global synchronization of TCP flows, maintain high throughput as well as a low delay and achieve fairness over multiple TCP connections.
In contrast to traditional queue management algorithms, which drop packets only when the buffer is full i.e; tail drop algorithm, the RED algorithm drops arriving packets randomly based on probability by calculating average queue size.. The probability of drop increases as the estimated average queue size grows.
The RED router calculates the average queue size, using a low-pass filter with an exponential weighted moving average(EWMA). The average queue size is compared to two thresholds, a minimum threshold and a maximum threshold. When the average queue size is less than the minimum threshold, no packets are marked. When the average queue size is greater than the maximum threshold, every arriving packet is marked. This ensures that the average queue size does not significantly exceed the maximum threshold.
When the average queue size is between the minimum and the maximum threshold, each arriving packet is marked with probability Pa , where Pa is a function of the average queue size avg.

- ❑ **Flowchart:**

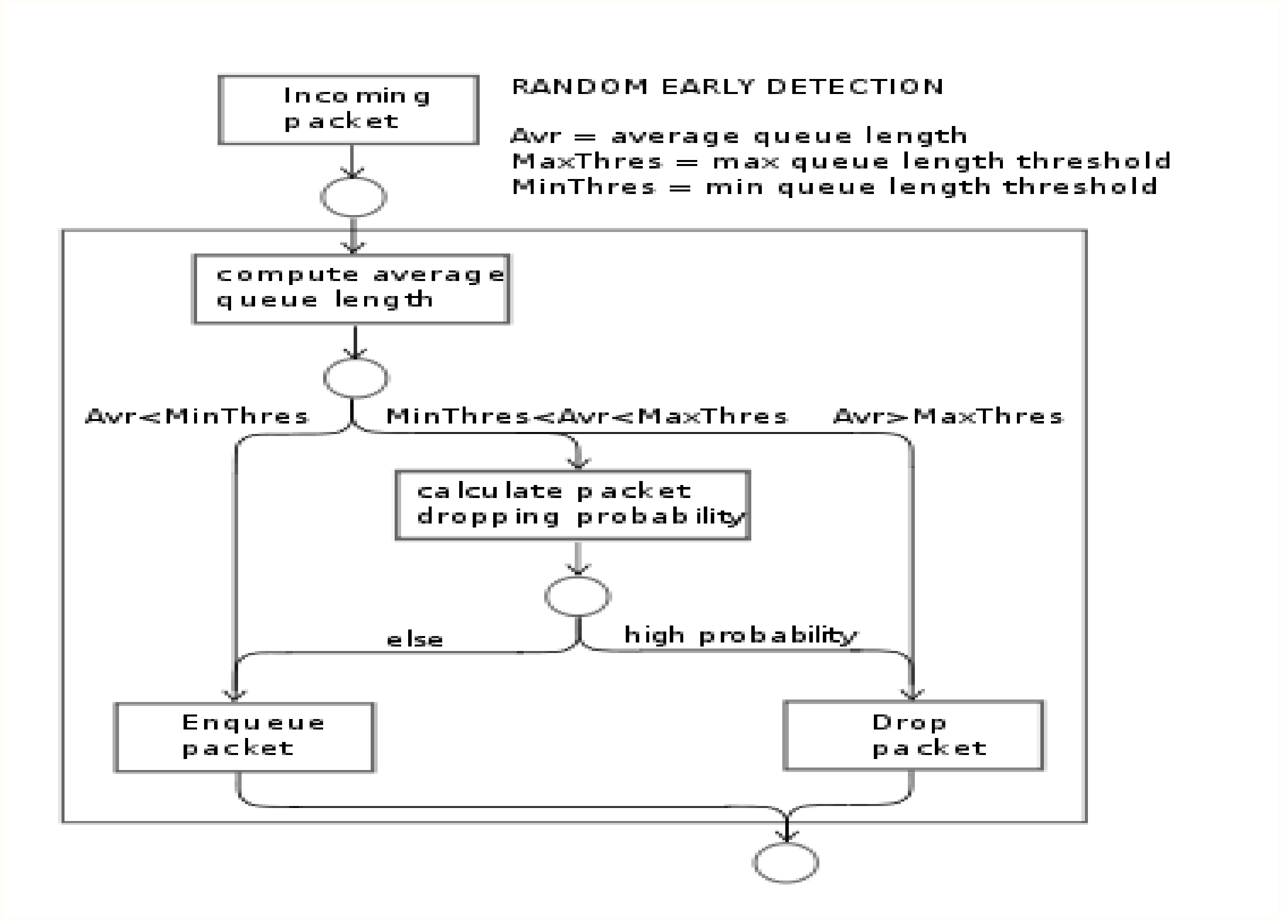

RANDOM EARLY DETECTION
Avr = average queue length
MaxThres = max queue length threshold
MinThres = min queue length threshold
Incoming packet
compute average queue length
Avr<MinThres
MinThres<Avr<MaxThres
Avr>MaxThres
calculate packet dropping probability
else
high probability
Enqueue packet
Drop packet

Image Source: wikipedia of RED (https://en.wikipedia.org/wiki/Random_early_detection)

- ❑ **RED Operation:**

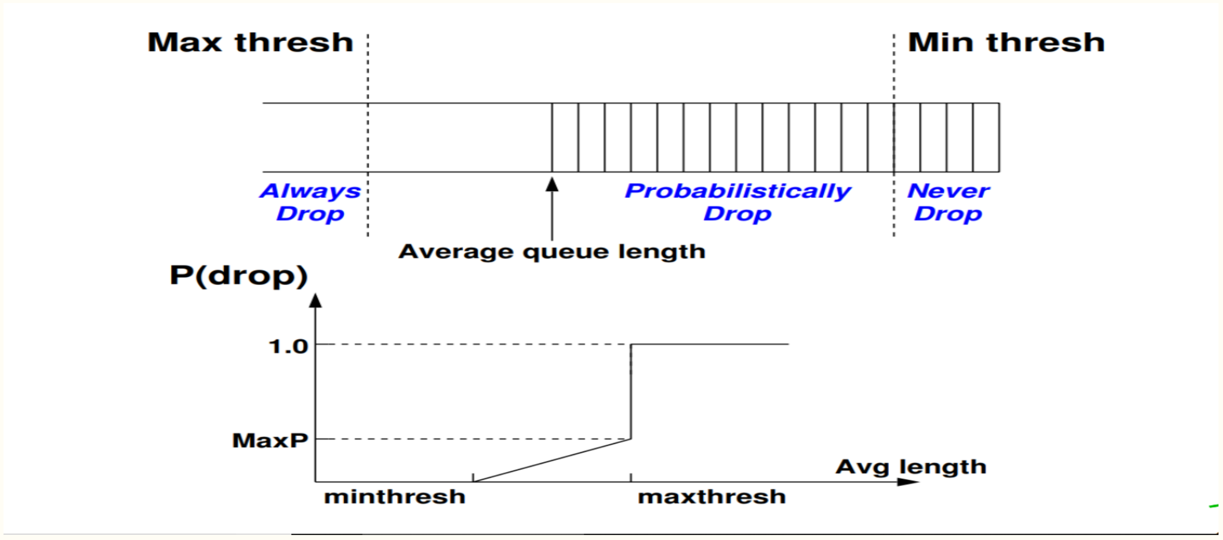


Image source: http://merlot.usc.edu/cs551-m05/lectures/tentative/12b_red.pdf

❑ The RED queue contains a number of attributes that control the RED policies:

- minth, maxth : minimum and maximum threshold of queue_disc
- Queue weight (Wq)
- Pmax : maximum value of drop probability(0.5).
- Link_Bandwidth : This rate is used for calculating the average queue size after some idle time. Should be set to the bandwidth of your interface.
- Packet drop probability

❑ Parameter values assignment:

- ### Setting Wq value

Here the Wq value is less to ensure that we are able to detect temporary congestion. Assume that the queue changes from empty to one packet, and that, as packets arrive and depart at the same rate, the queue remains at one packet. In most of our simulations we use Wq= 0.002.
Note:-> If Wq is set too low, then average responds too slowly to changes in the actual queue size. In this case,the gateway is unable to detect the initial stages of congestion.
-> If Wq is too large, then the averaging procedure will not filter out transient congestion at the gateway.

- ### Setting values of min and max

The optimal values of *min* and *max* depends on the desired average queue size. If the traffic is burst then *min* must be of enough large value to allow maximum link utilization. *max* value depends on maximum average delay that can be allowed by the gateway.
Note:
The RED gateway functions most effectively when *max* - *min* is larger than the typical increase in the calculated average queue size *avg* in one round trip time. A useful rule isto set *max* to at least twice *min*.

```
static inline void red_set_parms(struct red_parms *p,
                                 u32 qth_min, u32 qth_max, u8 Wlog, u8 Plog,
                                 u8 Scell_log, u8 *stab, u32 max_P)
{
        int delta = qth_max - qth_min;
        u32 max_p_delta;
```

```
    p->qth_min   = qth_min << Wlog;
    p->qth_max   = qth_max << Wlog;
    p->Wlog             = Wlog;
    p->Plog             = Plog;

    }
```

❑ Methods used in RED Queue

- Enqueue
- Average Queue Calculation
- Drop

❑ Enqueuing in RED:

Below is code for enqueuing of packets in buffer which will be according to the condition of min and max relative to the average queue length. Average *avg* value is used to enqueue and dequeue packets in queue buffer. The average queue size is compared to two thresholds, a minimum *min* threshold and a maximum *max* threshold.

Step 1: Calculating Average queue size

When a new packet arrives we calculate the average queue length.The router implements the low pass filter to calculate average queue size. The implemented low pass filter is an exponential weighted moving average (EWMA).

The calculation is as follows:

$avg \leftarrow (1 - Wq)*Prev\_avg + Wq*current\_q$: when queue is not empty

$avg \leftarrow ((1 - Wq)\hat{}m)*avg$: when queue is empty or idle

w_q : time constant for low pass filter
m : idle time of the queue/transmission time

```
static inline unsigned long red_calc_qavg(const struct red_parms *p,
                                const struct red_vars *v,
                                unsigned int backlog)
{
      if (!red_is_idling(v))
            return red_calc_qavg_no_idle_time(p, v, backlog);
      else
            return red_calc_qavg_from_idle_time(p, v);
}

static inline unsigned long red_calc_qavg_no_idle_time(const struct red_parms
*p,
                                           const struct red_vars *v,
                                           unsigned int backlog)
{

            return v->qavg + (backlog - (v->qavg >> p->Wlog));
}


static inline unsigned long red_calc_qavg_from_idle_time(const struct
red_parms *p,
                                           const struct red_vars *v)
{
      s64 delta = ktime_us_delta(ktime_get(), v->qidlestart);
      long us_idle = min_t(s64, delta, p->Scell_max);
      int  shift;

         shift = p->Stab[(us_idle >> p->Scell_log) & RED_STAB_MASK];

      if (shift)
            return v->qavg >> shift;
      else {
                 us_idle = (v->qavg * (u64)us_idle) >> p->Scell_log;

            if (us_idle < (v->qavg >> 1))
                  return v->qavg - us_idle;
            else
                  return v->qavg >> 1;
      }
}
```

Step 2: If *avg* is less than or equal to minimum threshold *min* of queue buffer then enqueue the packets and if avg is greater than min and less than max than enqueue or drop based on probability Pa. If avg is greater than maximum threshold then drop or mark the packet.

```
static inline int red_cmp_thresh(const struct red_parms *p, unsigned long
qavg)
{
	if (qavg < p->qth_min)
		return RED_BELOW_MIN_THRESH;
	else if (qavg >= p->qth_max)
		return RED_ABOVE_MAX_TRESH;
	else
		return RED_BETWEEN_TRESH;
}

static inline int red_action(const struct red_parms *p,
			     struct red_vars *v,
			     unsigned long qavg)
{
	switch (red_cmp_thresh(p, qavg)) {
		case RED_BELOW_MIN_THRESH:
			v->qcount = -1;
			return RED_DONT_MARK;

		case RED_BETWEEN_TRESH:
			if (++v->qcount) {
				if (red_mark_probability(p, v, qavg)) {
					v->qcount = 0;
					v->qR = red_random(p);
					return RED_PROB_MARK;
				}
			} else
				v->qR = red_random(p);

			return RED_DONT_MARK;

		case RED_ABOVE_MAX_TRESH:
			v->qcount = -1;
			return RED_HARD_MARK;
	}

	BUG();
	return RED_DONT_MARK;
}
```

Step 3: **Drop based on Pa: The packet-marking probability**, Pb is calculated as a linear function of the average queue size and varies linearly from 0 to max p :
Pb ← max p (avg − min th)/(max th − min th).
The final packet-*marking probability Pa increases slowly as the count increases since the last *marked packet.
Pa ←Pb /(1−count*Pb)
If Pa < R(random packet marking number)
Than enqueue;
Else
Drop;

```
static int red_enqueue(struct sk_buff *skb, struct Qdisc *sch,
                       struct sk_buff **to_free)
{
      struct red_sched_data *q = qdisc_priv(sch);
struct Qdisc *child = q->qdisc;
      int ret;

      q->vars.qavg = red_calc_qavg(&q->parms,
                                   &q->vars,
                                   child->qstats.backlog);

      if (red_is_idling(&q->vars))
            red_end_of_idle_period(&q->vars);
switch (red_action(&q->parms, &q->vars, q->vars.qavg)) {
      case RED_DONT_MARK:        // When the average  queue avg size is less
       break;                    // than the minimum threshold min , no packets are marked.


      case RED_PROB_MARK:
   // This is the case where marking of packets takes according to probability Pa.
qdisc_qstats_overlimit(sch);   .
            if (!red_use_ecn(q) || !INET_ECN_set_ce(skb)) {
                  q->stats.prob_drop++;
                  goto congestion_drop;
}

            q->stats.prob_mark++;
            break;

      case RED_HARD_MARK:
```

```
			qdisc_qstats_overlimit(sch);
			if (red_use_harddrop(q) || !red_use_ecn(q) ||
			    !INET_ECN_set_ce(skb)) {
				q->stats.forced_drop++;
				goto congestion_drop;
			}

			q->stats.forced_mark++;
			break;
	}

	ret = qdisc_enqueue(skb, child, to_free);
	if (likely(ret == NET_XMIT_SUCCESS)) {
		qdisc_qstats_backlog_inc(sch, skb);
		sch->q.qlen++;
	} else if (net_xmit_drop_count(ret)) {
		q->stats.pdrop++;
		qdisc_qstats_drop(sch);
	}
	return ret;
```

2> Adaptive RED implementation in Linux kernel

❏ Model description:

The motivation of ARED is that it enhances the efficiency of transfers and cooperate with TCP congestion window mechanism in adapting the flows intensity to the congestion of a network, because RED is too sensitive to parameters and it allows all packet drop after *max* that is maximum threshold.

ARED is a variant of RED with two main features:
(i) automatically sets Queue weight, *min* and *max* and (ii) adapts maximum drop probability. This model in linux kernel contains implementation of both the features, and is a port of Sally Floyd's ARED model. Note that the user is allowed to choose and explicitly configure the simulation by selecting feature (i) or feature (ii), or both.

❏ Methods of ARED

- Red adaptive timer
- Red adaptive algo

❏ Attributes of ARED

- Target delay: the factor average seeks to it (5ms for 100ms RTT)
- Interval: the time at which calculations are carried out
- Alpha: increment factor (min (0.01, Pmax/4))
- Beta: decrease factor (>0.83)

Target range: Our aim is to bring the average queue size closer to its target value more quickly. For this purpose, we compare the current average queue size with specified target queue size. It is to maximize the use of link bandwidth and to reduce packet loss.

Target range= $[min + 0.4 * (max - min), min + 0.6 * (max - min)]$

Interval: For each 0.5 seconds timing it will set the adaptive value of Pmax.

```
static inline void red_adaptative_timer(struct timer_list *t)
{
        struct red_sched_data *q = from_timer(q, t, adapt_timer);
        struct Qdisc *sch = q->sch;
        spinlock_t *root_lock = qdisc_lock(qdisc_root_sleeping(sch));

        spin_lock(root_lock);
        red_adaptative_algo(&q->parms, &q->vars);
        mod_timer(&q->adapt_timer, jiffies + HZ/2);
        spin_unlock(root_lock);
}
```

❑ Adaptive setting of parameters:

min* and *max

The value of *min* will be set accordingly requirement of desired average queue which will be maximum of target delay(5ms) and half of multiplicative term of target delay and link capacity, and the value of *max* will be thrice of min.

Pmax

- Pmax is adapted not just to keep the average queue size between *min* and *max* , but to keep the average queue size within a target range halfway between *min* and *max*.
- Pmax is adapted slowly, over time scales greater than a typical round-trip time, and in small steps.
- Pmax is constrained to remain within the range [0.01, 0.5] (or equivalently, [1%, 50%]). Instead of multiplicatively increasing and decreasing Pmax , we use an additive-increase multiplicative decrease policy.

```
static inline void red_set_parms(struct red_parms *p,
                                 u32 qth_min, u32 qth_max, u8 Wlog, u8 Plog,
                                 u8 Scell_log, u8 *stab, u32 max_P)
{
        int delta = qth_max - qth_min;
        u32 max_p_delta;

        p->qth_min  = qth_min << Wlog;
        p->qth_max  = qth_max << Wlog;
        p->Wlog           = Wlog;
        p->Plog           = Plog;
        if (delta <= 0)
                delta = 1;
        p->qth_delta= delta;
        if (!max_P) {
                max_P = red_maxp(Plog);
                max_P *= delta; /* max_P = (qth_max - qth_min)/2^Plog */
        }
        p->max_P = max_P;
        max_p_delta = max_P / delta;
        max_p_delta = max(max_p_delta, 1U);
        p->max_P_reciprocal  = reciprocal_value(max_p_delta);

        /* RED Adaptative target :
         * [min_th + 0.4*(min_th - max_th),
         *  min_th + 0.6*(min_th - max_th)].
         */
        delta /= 5;
        p->target_min = qth_min + 2*delta;
        p->target_max = qth_min + 3*delta;

        p->Scell_log= Scell_log;
        p->Scell_max= (255 << Scell_log);

        if (stab)
                memcpy(p->Stab, stab, sizeof(p->Stab));
}
```

```
static inline void red_adaptative_algo(struct red_parms *p, struct red_vars
*v)
```

```
{
        unsigned long qavg;
        u32 max_p_delta;

        qavg = v->qavg;
        if (red_is_idling(v))
                qavg = red_calc_qavg_from_idle_time(p, v);

        /* v->qavg is fixed point number with point at Wlog */
        qavg >>= p->Wlog;

        if (qavg > p->target_max && p->max_P <= MAX_P_MAX)
                p->max_P += MAX_P_ALPHA(p->max_P); /* maxp = maxp + alpha */
        else if (qavg < p->target_min && p->max_P >= MAX_P_MIN)
                p->max_P = (p->max_P/10)*9; /* maxp = maxp * Beta */

        max_p_delta = DIV_ROUND_CLOSEST(p->max_P, p->qth_delta);
        max_p_delta = max(max_p_delta, 1U);
        p->max_P_reciprocal = reciprocal_value(max_p_delta);
}
```

3> CODEL implementation in Linux kernel

❑ Introduction:

Codel operates at dequeue/drop and it uses queuing delay as a metric for queue management.

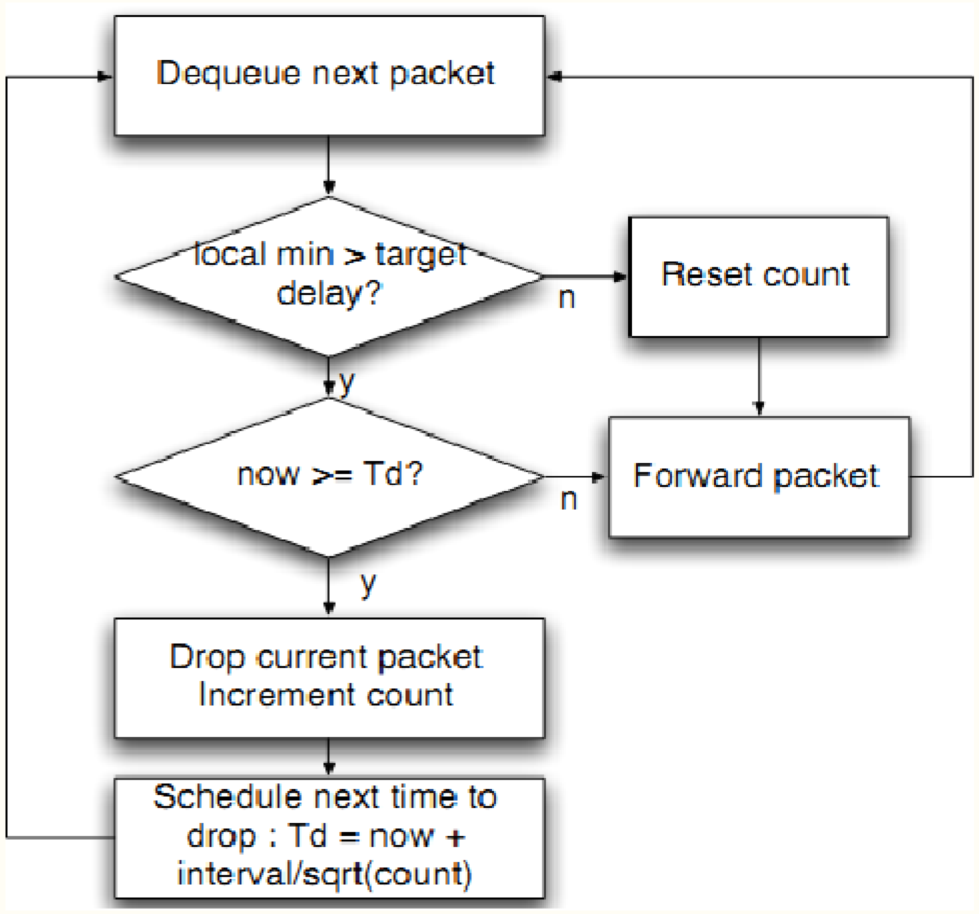
The terms which is used in documentation.

The algorithm works in two parts:

(i) at the time of enqueuing the packet and (ii) at the time of dequeuing the packet. On departure of each packet, the timestamp is removed from the header and is deducted from the current time to obtain the packet sojourn time.

Sojourn time: The amount of time packet spent in buffer, that is queuing delay.

❑ Flowchart:


Dequeue next packet
local min > target delay?
n
Reset count
y
now >= Td?
n
Forward packet
y
Drop current packet Increment count
Schedule next time to drop : Td = now + interval/sqrt(count)

Parameters in CODEL

There are two supreme important CoDel parameters to be set to reachfinestresults [7]: target and interval. These are stable parameters and their values are selected based on the explanations from numerous research. Succeeding are the values for target and interval

- target = acceptable standing queue delay (constant 5ms)
- interval = time on the order of worst case RTT through the bottleneck (constant between 10ms to 1 sec)

❑ **Methods in Codel**

- Enqueue
- Dequeue
- Drop

❑ **Enqueuing in CODEL:** On arrival of each packet, the current queue size is tested. If it is less than the queue limit, the packet is enqueued and the timestamp is attached in the header. This timestamp specifies enqueue time.

Step1: On arrival of every packet it attach a timestamp tag and on departure of every packets calculate queuing delay.

```
static int codel_qdisc_enqueue(struct sk_buff *skb, struct Qdisc *sch,
                               struct sk_buff **to_free)
{
      struct codel_sched_data *q;

      if (likely(qdisc_qlen(sch) < sch->limit)) {
            codel_set_enqueue_time(skb);
            return qdisc_enqueue_tail(skb, sch);
      }
      q = qdisc_priv(sch);
      q->drop_overlimit++;
      return qdisc_drop(skb, sch, to_free);
}
static void codel_set_enqueue_time(struct sk_buff *skb)
{
	get_codel_cb(skb)->enqueue_time = codel_get_time();
}
static inline int qdisc_enqueue_tail(struct sk_buff *skb, struct Qdisc *sch)
{
	return __qdisc_enqueue_tail(skb, sch, &sch->q);
}
```

❑ Dequeue and Drop condition in CODEL:

There are two branches based on whether the controller is in drop state:

(i) if the controller is in drop State (that is, the minimum packet sojourn time is greater than target), then the controller checks if it is time to leave drop state or schedules the next drop or

(ii) if the controller is not in drop state, it determines if it should enter drop state and do the initial drop.

Step2: if queue delay is greater than target and dropping state equal to true then drop packet.

If the packet sojourn time above the target for a listed interval of time, CoDel go into the dropping state and starts proactively dropping/marking the packets. Note that the packets are proactively dropped while dequeuing rather than during enqueuing. The time interval between the two proactive packet drops is calculated.The count shows the total number of packets dropped since the dropping state is entered

```
static struct sk_buff *codel_qdisc_dequeue(struct Qdisc *sch)
{
	struct codel_sched_data *q = qdisc_priv(sch);
	struct sk_buff *skb;
```

```
      skb = codel_dequeue(sch, &sch->qstats.backlog, &q->params, &q->vars,
                          &q->stats, qdisc_pkt_len, codel_get_enqueue_time,
                          drop_func, dequeue_func);

      if (q->stats.drop_count && sch->q.glen) {
            qdisc_tree_reduce_backlog(sch, q->stats.drop_count,
q->stats.drop_len);
            q->stats.drop_count = 0;
            q->stats.drop_len = 0;
      }
      if (skb)
            qdisc_bstats_update(sch, skb);
      return skb;
}
```

Step 3: While the algorithm is in dropping state, if the packet sojourn time come to be lesser than target or if queue does not have enough packets to fill the outgoing link, the algorithm leaves the dropping state.

Step 4: Now the controlled law will be evaluate to calculate next packet drop time and it will drop packet according to requirement to make room in buffer.

```
While dequeue_time >= nextdrop_time
do
   Drop the packets
   count= count+1
nextdrop_time + =interval / √ count
```

```
static codel_time_t codel_control_law(codel_time_t t,
                              codel_time_t interval,
                              u32 rec_inv_sqrt)
{
      return t + reciprocal_scale(interval, rec_inv_sqrt <<
REC_INV_SQRT_SHIFT);
}
```

4> PIE Implementation in Linux kernel

❑ Introduction:

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.

-> Some important points of PIE:

1> PIE has two states:

(1) Active state (2) Inactive state

Note: PIE becomes active again when queue_length >= (tail_drop/3)

When buffer occupancy over certain threshold , which may be set to ⅓ of the tail drop threshold.

2> If PIE is in measurement cycle then only dqueue_rate is calculated.

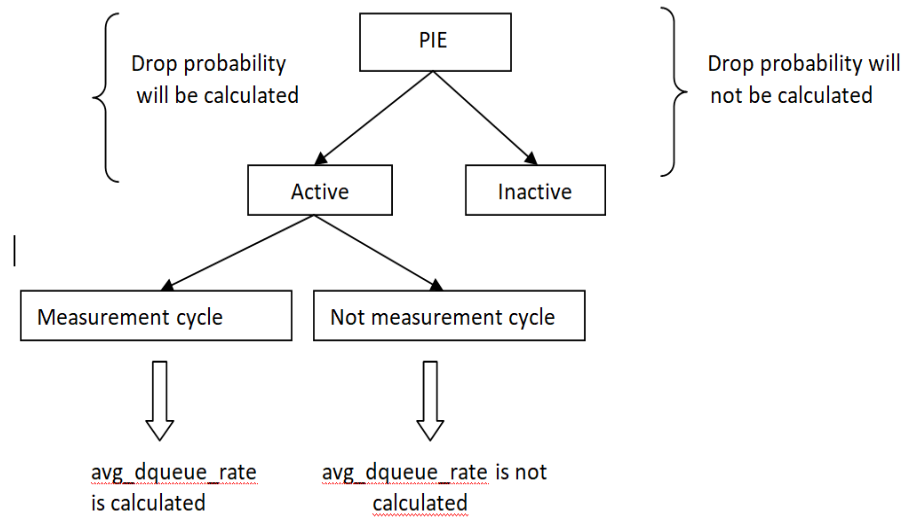
Note: if(cur_qlength >=dqueue_threshold)
Then PIE is in measurement cycle.
Else Not in measurement cycle

3> Active and inactive states are meant to handle idle periods.

4>The concept of measurement cycle is meant to handle situation , when queue occupancy is very less.

5>When queue occupancy is zero means the link is usually not utilized completely.

6>When queue occupancy is less means the link is completely utilized.


PIE
Drop probability will be calculated
Drop probability will not be calculated
Active
Inactive
Measurement cycle
Not measurement cycle
avg_dqueue_rate is calculated
avg_dqueue_rate is not calculated

❑ Model Description:

the PIE design has the following basic criteria.

- 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.
- PIE aims to attain high link utilization. The goal of low latency shall be achieved without suffering link underutilization or losing network efficiency.
- 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.
- Design parameters shall be set automatically. Users only need to set performance-related parameters such as target queue latency, not design parameters.

❑ PIE Structure

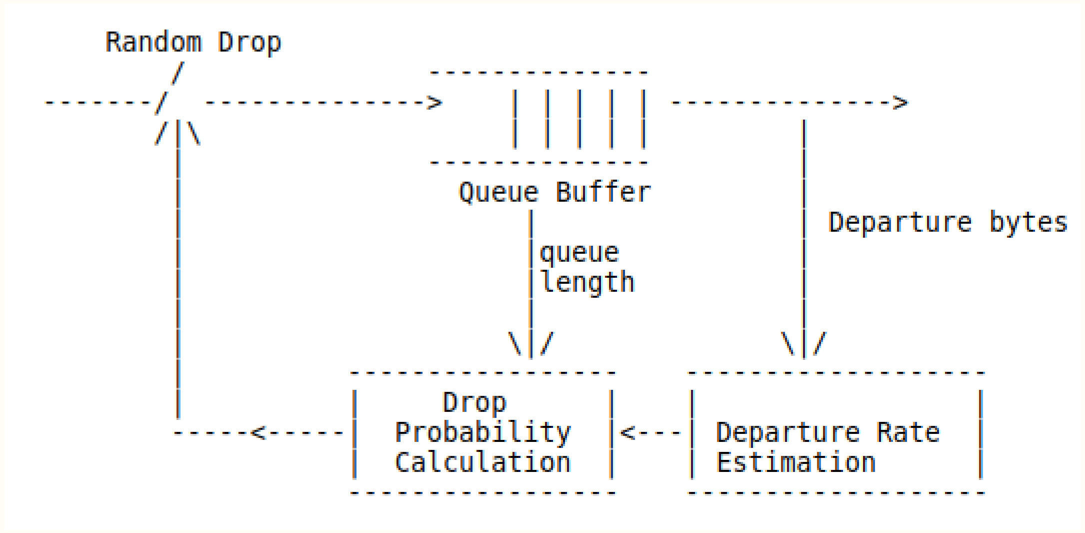


Figure 1: The PIE Structure

Image source: https://datatracker.ietf.org/doc/rfc8033/?include_text=1

❑ Parameters in PIE

- t_update (16ms)
- Count : total packet size in t_update
- Cur_dqueue_rate: current departure rate
- Avg_dqueue_rate: average departure rate
- alpha(0.125)
- Cur_del: current queue delay during current sample
- Ref_del: reference queue delay that is desired delay(15ms)
- Old_del: queuing delay during the old sample
- dqueue_threshold(10kb)
- Max_burst(150ms)

❑ Important components of PIE:

- Random dropping
- Drop probability calculation
- Average departure rate estimation
- Burst allowance calculation

❑ PIE Operation:

Step1: PIE randomly drops a packet upon its arrival to a queue according to a drop probability.The random drop is triggered by a packet's arrival before enqueuing into a queue.

Step2: It will calculate current dequeue rate first. Which is *cur_dequeue_rate=(count/t_update)*

Step3: Then it will calculate average departure rate that means average number of packets departed up to that time from queue.

avg_dqueue_rate=((1-alpha)avg_dqueue_rate)+(alpha(cur_dqueue_rate))*

Note: dequeue rate typically measured in bytes.

Step4: Now the current queue delay will be calculated.

Cur_del = cur_qlength/avg_dqueue_rate

Note: Little's law is used to estimate queue delay.

Step5: 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.

$$|P| = alpha\ (cur\_del - ref\_del)+beta\ (cur\_del - old\_del)$$

Note: If *cur_del = ref_del* then it will adopt the stable state.

In PIE alpha and beta are auto tuned.

```
static void calculate_probability(struct Qdisc *sch)
{
      struct pie_sched_data *q = qdisc_priv(sch);
      u32 glen = sch->qstats.backlog;     /* queue size in bytes */
      psched_time_t qdelay = 0;     /* in pschedtime */
      psched_time_t qdelay_old = q->vars.qdelay;/* in pschedtime */
      s32 delta = 0;          /* determines the change in probability */
      u32 oldprob;
      u32 alpha, beta;
      bool update_prob = true;

      q->vars.qdelay_old = q->vars.qdelay;

      if (q->vars.avg_dq_rate > 0)
            qdelay = (glen << PIE_SCALE) / q->vars.avg_dq_rate;
      else
            qdelay = 0;

            if (qdelay == 0 && glen != 0)
            update_prob = false;
      if (q->vars.prob < MAX_PROB / 100) {
            alpha =
                (q->params.alpha * (MAX_PROB / PSCHED_TICKS_PER_SEC)) >> 7;
            beta =
                (q->params.beta * (MAX_PROB / PSCHED_TICKS_PER_SEC)) >> 7;
      } else if (q->vars.prob < MAX_PROB / 10) {
            alpha =
                (q->params.alpha * (MAX_PROB / PSCHED_TICKS_PER_SEC)) >> 5;
            beta =
                (q->params.beta * (MAX_PROB / PSCHED_TICKS_PER_SEC)) >> 5;
      } else {
            alpha =
                (q->params.alpha * (MAX_PROB / PSCHED_TICKS_PER_SEC)) >> 4;
            beta =
                (q->params.beta * (MAX_PROB / PSCHED_TICKS_PER_SEC)) >> 4;
      }

            delta += alpha * ((qdelay - q->params.target));
      delta += beta * ((qdelay - qdelay_old));

      oldprob = q->vars.prob;

            if (delta > (s32) (MAX_PROB / (100 / 2)) &&
          q->vars.prob >= MAX_PROB / 10)
            delta = (MAX_PROB / 100) * 2;

            if (qdelay > (PSCHED_NS2TICKS(250 * NSEC_PER_MSEC)))
            delta += MAX_PROB / (100 / 2);

      q->vars.prob += delta;

      if (delta > 0) {
                        if (q->vars.prob < oldprob) {
                  q->vars.prob = MAX_PROB;

                  update_prob = false;
            }
      } else {
                        if (q->vars.prob > oldprob)
                  q->vars.prob = 0;
      }
      if ((qdelay == 0) && (qdelay_old == 0) && update_prob)
            q->vars.prob = (q->vars.prob * 98) / 100;

      q->vars.qdelay = qdelay;
      q->vars.qlen_old = qlen;

      if ((q->vars.qdelay < q->params.target / 2) &&
          (q->vars.qdelay_old < q->params.target / 2) &&
          (q->vars.prob == 0) &&
          (q->vars.avg_dq_rate > 0))
            pie_vars_init(&q->vars);
}
```

Step6: A new parameter called Max_burst is added in PIE to ensure packets are not dropped/ marked during short term burst.

Default value =150ms

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