Indian Journal of Science and Technology, *Vol 8(35), DOI: 10.17485/ijst/2015/v8i35/81476, December 2015*

ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Comparative Analysis and Comparison of Various AQM Algorithm for High Speed

Uma R. Pujeri[1], V. Palaniswamy[1], P. Ramanathan[2], Ramachandra Pujeri[3*]

[1]Anna University, Chennai - 600025, Tamil Nadu, India; umaresearch81@gmail.com, v.palaniswamy81@gmail.com
[2]Info Institute of Technology, Coimbatore - 641107, Tamil Nadu, India; pramanathan2509@)gmail.com
[3]Pune university, Pune – 411007, Maharashtra, India; sriramu.vp@gmail.com

Abstract

Congestion deteriorates the network performance. In this paper various congestion control AQM algorithm are analyzed and surveyed with their shortcomings and their advantages. The main objective of the paper is to study existing AQM algorithm and develop a new AQM algorithm that gives better result than the existing algorithm. AQM are the router based mechanism for early detection of congestion in the computer network. The basic idea of AQM is to sense and detect congestion in advance and to inform the sender to reduce its sending rate, thereby reducing the number of packets sent in the network and control the congestion. There are several AQM algorithms that controls the congestion. In this paper, we have surveyed, compared and analyzed Random Early Detection (RED), Flow Random Early Detection (FRED), Stabilized Random Early Detection (SRED), Stochastic Fair Queuing (SFQ), Random Exponential Marking (REM), BLUE, Stochastic Fair BLUE (SFB) AQM algorithms. Performance parameter are tested and evaluated in NS2 simulator. After analyzing it was found that RED AQM compared with SFQ and REM achieved best result in terms of delay. SFQ had minimum average ratio and RED had max loos ratio. REM algorithm showed the best result with respect to throughput loss ratio and link utilization. **Improvement:** After analyzing and comparing several AQM algorithm it was found that no single algorithm can solve all the problems. Hence a research is needed to develop a new AQM algorithm that has good link utilization, is fair enough, has less loss ratio, require less space and easy for configuration.

Keywords: BLUE, FRED, QoS, Queue Management, RED, REM, SFB, SFQ, SRED

1. Introduction

AQM algorithm are congestion control algorithm which increases throughput, link utilization and decreases the delay and packet loss. The first AQM algorithm called RED (Random Early Detection) provides the mechanism for congestion avoidance. This technique had several drawback and one among which was inability to deal with busty traffic. RED algorithm is studied, analyzed by many researchers and RED has been the basis for the development for new AQM algorithm. The major objective of RED algorithm are:

RED OBJECTIVES

- To monitor the queue length
- High link utilization
- Early congestion detection
- Minimize queuing delay
- Decrease packet loss
- Achieve fairness
- Avoid global synchronization

RED algorithm monitors the average queue size and drop the packet based on the statistical probability. Unlike RED FRED does not make the dropping of packet decision on queue length but FRED monitors each active flow within the buffer and usage of bandwidth of each flow and takes the dropping decision depending on the usage of bandwidth of active flow. Cost of FRED is independent from number of flow but is proportional to the buffer size. SRED algorithm is stabilized RED algorithm with additional features added to the RED algorithm.

**Author for correspondence*

The goal of SRED algorithm is to trace the active flow in queue that take more bandwidth share and allocate equal fair share of bandwidth to all the active flow in the queue was proposed by Jhon Nagle in 1987 which is fair queuing algorithm. SFQ is called "stochastic" because SFQ algorithm divides the traffic over a number of queues using hashing and round robin algorithm. REM is an AQM algorithm which achieves both high utilization and negligible loss and delay in a simple and scalable manner. REM algorithm has two specific key features and they are:

- **Match Rate Clear Buffer** - Stabilizes the queue around small target regardless of number of user that is it matches the user rate with network capacity while clearing the buffer.
- **Sum Prices** - It is sum of link prices (congestion measures), summed over all the router in the route of the packet from source to destination to estimate end-to-end marking or dropping probability.

In BLUE AQM algorithm the congestion is managed through packet loss and link utilization history. BLUE maintains only a single marking probability variable. When the queue continuously starts to drop the packet due to congestion or buffer overflow, marking probability is incremented by 1 else when the queue is empty or idle this marking probability is decremented. SFB is an AQM algorithm which identifies and rate limits non responsive flow using very small amount of state information.

The purpose of this survey is to revisit different AQM algorithm like RED, SFQ, REM, FRED, SRED, BLUE and SFB, outline different consideration in their design and also highlight their disadvantages and limitations which will help in design of new algorithm that can take maximum advantages of the existing algorithm and dilute the limitations.

The paper is organized in following manner Section 2 discusses AQM algorithms in details with step by step algorithm for each AQM algorithm considered in this paper. Section 3 compares each algorithm with respect to link utilization, fairness, space requirement, per flow state information, advantages and disadvantages. Section 4 discusses conclusion and future work.

2. AQM Algorithm

AQM algorithm senses the network congestion in advance and inform the sender to reduce its sending rate thus minimizing the number packets in the network. There are several AQM algorithms but in this paper we have considered following AQM algorithm they are:

- RED
- SFQ
- REM
- FRED
- SRED
- BLUE
- SFB

2.1 Random Early Detection (RED)

RED is an AQM algorithm which is also known as Random Early Detection or Random Early Discard or Random Early Drop that provides mechanism for congestion avoidance. Traditional drop tail algorithm drop the packets if the buffer is full. Drop tail algorithm does not fairly distribute the buffer space among the traffic flow. Drop tail algorithm can also lead to global synchronization. This problem is overcome in TCP RED.

TCP RED monitors queue size depending on the queue RED takes the decision of dropping the packet, that is if the queue is empty all the packets are accepted, as queue becomes full the probability of dropping the packet also increases. When the queue becomes full all the incoming packets are dropped.

2.1.1 Random Early Detection (RED) Algorithm

```
Step 1: Calculate the average queue size AvgQueSize
Step 2: If(AvgQueSize > MaxQueSize)
            drop the packet
Step 3: else if(AvqQueSize ==0)
            queue empty
            accepts all incoming packets
Step 4: else if(AvgQueSize=minqueuethreshold)
            calculate dropping probability pa
            drop the packet with probability pa
Step 5: else forward the packet.
```

2.2 Stochastic Fair Queuing (SFQ)

SFQ - Stochastic Fairness Queuing (SFQ) is an AQM algorithm that uses hashing and round robin algorithm. In SFQ algorithm a traffic flow is identified by four options they are source address, destination address, source port and destination port. These parameters are used by SFQ hashing algorithm to classify the packet into

1024 sub-streams. Bandwidth is distributed equally to all the sub-stream using round robin algorithm. Round robin algorithm allocates a SFQ-allot bytes of traffic on each round thus fairly distributing the available bandwidth among all the sub-stream.SFQ queue contains 1024 sub-stream and 128 packets
SFQ algorithm
Parameters
int maxqueue max queue size in packets
int buckets number of queues

Functions

1. Enque the packet enque()
2. Dequeue the packet dequeue()
3. Calculate the hash function hash()
4. Calculate the fair share initsfq()

2.2.1 Steps for Enqueue Function

```
Step 1: Initialize variables which, used, left. (which variable
        is used to find which queue from 128 queues, used
        variable is used to find number of bytes used by
        the queue, left variable is used to find left space in
        the queue)
            PacketSFQ *q
Step 2: Check if (!bucket)
            Call function initsfq
            Which = hash(pkt) % bucket
            q = &bucket[which]
            left = maxqueue-occupied
Step 3: If maxqueue is changed while running left can
        become less than 0
            check if ((used>=(left>>1)) ||
            (left<bucket_ && used > fairshare) ||
            (left <=0))
            drop the packet
            else
            enqueue the packet.
```

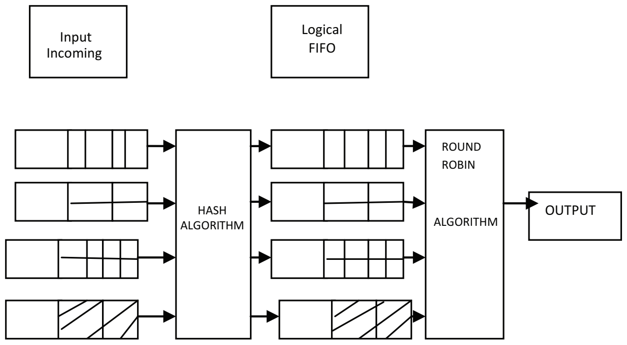

Figure 1. SFQ operation.

2.2.2 Steps for Dequeue Function

```
Step 1: Packet *pkt
        check if(!bucket)
        call function initsfq()
Step 2: check(!active)
        return 0
        dequeue the packet
Step 3: check if(active->pkts ==0)
        active=active->idle(active)
        else
        active=active->next
        return the packet
```

2.2.3 Steps for Hash function

```
Step 1: int i is pointer to source address of packet
        int j is pointer to destination address of packet.
        int k = i + j
Step 2: Calculate and return k
        (k+(k>>8)+~(k>>4)%((2<<19)–1)
```

2.2.4 Steps for Initsfq Function to Calculate Fair Share

```
Step 1: active = 0
        occupied = 0
        fairshare = maxqueue_ / buckets
```

2.3 Random Exponential Marking (REM)

Random Exponential Marking is an active queue management algorithm which decouples the congestion measures with performance measure such as delay, Packet delivery ratio, throughput packet loss etc and stabilizes the performance measure around the target independently of the number of users.

REM has two important key features.

- Match the rate clear buffer,
- Sum Prices.

Match rate clear buffer: REM algorithm stabilizes the input rate with the total capacity and the queue length with small target irrespective of the number of user sharing the link. REM output queue uses a variable called “price” which is used to estimate marking probability. The value of price variable is updated periodically or asynchronously in two cases:

- When there is difference between the input rate and link capacity.
- When there is difference between queue length and target.

The weighted sum is positive if the input rate exceeds the link capacity else the weighted sum is negative. When the weighted sum of these mismatches (that is difference between input rate and link capacity and the difference between queue length and target) is positive the value of the price variable is incremented else the value of the price variable is decremented.

Case 1: When number of users increases the input rates grows hence weighted sum is positive the price is incremented and so is the marking probability. In such case a strong congestion signal is sent to the sender to reduce their sending rate.

Case 2: When input rate is too low than the link capacity the weighted sum will be negative the price and marking probability is decremented which raises the source rate until the mismatches are driven to zero.

Thus REM explicitly control the value of price. Value of price is updated for queue length l in time period t with the following formula

$$pl(t+1)= [pl(t) + \gamma(\alpha 1(bl(t)- bl^*)+xl(t)-cl(t))]+ \quad (1)$$

Where
$\gamma> 0$
Small constants
$\alpha 1 > 0$
bl(t) is buffer occupancy at queue l in period t.
$bl*(t)>=0$ target queue length
xl(t) is input rate
cl(t) is available bandwidth of queue l in period t
xl(t)-cl(t) is rate mismatch
bl(t)-bl* is queue mismatch

α can be set by each queue depending on the bandwidth utilization and queuing delay. γ controls the responsiveness of the REM to control the network condition. When the weighted sum and queue mismatches are positive which are weighted by α the price value is increased else the price value is decreased.

Price can be stabilized when weighted sum is zero that is $\alpha 1(bl-bl*) + (xl-cl) = 0$ this can happen only when input rate equals the link capacity ($xl = cl$) and the queue length equals the target that is ($bl = bl*$)

When the target queue length b* is non zero the mismatch rate xl (t)-cl (t) can be bypassed to update the price xl (t)-cl (t) grows when queue length and buffer is nonempty. Hence approximate this term by change in backlog bl (t+1)-bl (t) it becomes:

$$Pl\ (t+1)=[pl\ (t)+\gamma(bl\ (t+1)-(1-\alpha l)bl\ (t)-\alpha lb*)]+ \quad (2)$$

Hence it is observed that REM algorithm the prices increases while the queue length is stabilized around the target bl* regardless of the increase in number of users.

Sum Prices: Sum Prices is the sum of all the link prices along the path from source to destination to estimate end-to-end marking probability.

Suppose a packet traverse links l = 1,2,3,------l that have price pl (t) in time period t then the marking probability ml (t) at queue l in time period t is calculated as:

$$Ml\ (t) = 1-\Phi-pl\ (t) \quad (3)$$

where $\Phi > 1$ is a constant

End-to-end marking probability is calculated as:

$$1- -\Sigma lpl\ (t) \quad (4)$$

End-to-end marking probability is high when congestion in its path is large .When the link marking probability ml (t) are small hence the link prices pl (t) are small, the end-to-end marking probability is approximately proportional to the sum of the link prices in the path:

end-to-end marking probability (5)

REM Algorithm
Parameters

double v_pl	link price
double v_prob	packet marking probability
double v_in	input rate
double qib	queue in bytes.
double remp_.p_gama	value of gamma
double remp_.p_phi	value of phi
double remp_.p_pktsize	mean packet size
double remp_.p_updtime	update time
double remv_.v_prob	dropping probability
double curq	current queue size
int pmark	number of packets being marked.

Four functions:

- reset function.
- to compute average input rate price and marking probability.

- dequeue.
- enqueue.

2.3.1 Function Reset

It computes the “packets time constant” if link bandwidth is known ptc is the max number of packets per second which can be placed on link

Steps for Reset Function

```
Step 1: if(link_)
        calculate packet time constant
        remp_.p_ptc=link->bandwidth()/(8.0 * remp_.p+pktsize)
Step 2: initialize variables
        remv_.v_pl=0.0
        remv_.v_prob=0.0
        remv_.v_in=0.0
        remv_.v_count=0.0
        remv.v_pl1=0.0
```

2.3.2 Function to Compute Average Input Rate Price and Marking Probability

This function compute average input rate, price and marking probability. Link price is computed by following formula:

$$Pl = remv\_.v\_pl \tag{6}$$

Where remv_.v_pl is a link price stored in variable pl.

Input rate is calculated by the formulas:

$$In = remv\_.v\_count \tag{7}$$

where remv_.v_count contains number of packets arriving at link stored in a variable in

marking probability is calculated by following formulas.

Calculate maximum number of packets sent during one update interval :

double c = remp_.p_updatetime*remp_.p_ptc pl = pl+remp_.p_gamma*(in_avg+0.1*nqueued-remp_.p_bo)-c)

where pl is link price:

if (pl<0.0)
pl = 0.0

Calculate pow 1

pow1 = pow (remp_.p_phi-pl)

$$pr = 1.0\text{-}pow1 \tag{8}$$

where pr is marking probability.

2.3.3 Steps for Computing Average Input Rate Price and Marking Probability

```
Step 1: initialize variables
        double in, in_avg, nqueued, pl, pr.
Step 2: Calculate link price pl
        Pl = remv_.v_pl
Step 3: Calculate number of bytes or packets arriving at the
        link (input rate) during one update time interval
        In = remv_.v_count
Step 4: Calculate average input rate
        in_avg = rem_.v_ave
        in_avg* = (1.0-remp_.p_inw)
        check if(qlib)
        calculate in_avg
        in_avg+ = remp_.p_inw*in/remp_.p_pktsize
        nqueued = bcount/remp_.p_pktsize
        else
        Calculate average input rate with formula
        in_avg+ = remp_.p_inq*in
        nqueued = q->length()
Step 5: Calculate maximum number of packets sent
        during one update interval
        double c = remp_.p_updatetime*remp_.p_ptc
        pl = pl+remp_.p_gamma*(in_avg+0.1*nqueued-
        remp_.p_bo)-c)

        Step 5.1: check if(pl<0.0)
        Pl = 0.0

        Step 5.2: Calculate pow 1
        pow1=pow(remp_.p_phi-pl)
        pr = 1.0-pow1

        Step 5.3: Set the value of count, average, input rate
                  link price and marking probability.
        Pr = 1.0-pow1
        remv_.v_ave = in_avg
        remv_.v_pl = pl
        remv_.v_prob = pr
```

2.3.4 Steps for Enqueue of Packets

```
Step 1: intialize variable qlen
        check number of bytes in queue calculate input
        rate if(qib_)
        rem_.v_count+=ch->size()
        else
        ++remv_.v_count
```

```
Step 2: Calculate qlimit and qlength
        check if qlimit is greater than queue length
        if (qlen> = qlim)
        drop packet
        else
        mark the packet for drop probability
```

2.3.5 *Function for Dequeue of Packet*

```
Step 1: Packet *p = q->deque()
        if(p! = 0)
        Calculate the packet size
        bcount_= hdr_cmn::access(p)->size
Step 2: Calculate the marking probability
        If (markpkts_)
        double u = Random::uniform()

        Step 2.1: check if(p! = 0)
        double pro = rem_.v_prob

        Step2.2: if (qib_)
        Calculate size
        calculate dropping probability
        pro = remv_.v_prob*size/remp_.p_pktsize

        Step 2.3: Check if(u< = pro)
        mark the current packet
        pmark++
Step 3: Calculate queue length
        double qlen = qlib_?bcount_ :q_->length()
        curq_= int qlen.
```

2.4 Stabilized RED (SRED)

SRED is called as stabilized RED AQM algorithm which is derived from RED AQM algorithm by adding some feature to it. The goal of SRED algorithm is to identify the flow that take more bandwidth and to allocate the fair share of bandwidth without performing much computation .To achieve this SRED algorithm uses Zombie list which is small list of recently seen active flows with additional information for each flow in the list "count" and timestamps.

The zombie list initially is empty whenever a new packet arrives its packet identifier (source address, destination address) is added to the list. Count is set to zero and time stamp is set to arriving time of packet.

Once the zombie list is full SRED algorithm compare the arriving packet with the random selected zombie in the zombie list. After this comparison one out of two actions can be taken from the following:

1. Whenever the arriving packet matches with the packet in the zombie list it is a hit the variable count is increased by one and timestamps is set to latest packet arrival time.
2. Whenever the new arriving packet do not match with random selected packet (zombie) in the zombie list it is no hit or miss then zombie list is replaced or over written by the new arriving packet. Count is set to zero and time stamp is set to the arrival time at the buffer with probability p.

SRED estimates p (t) for hit frequency of tth packet at the buffer.

Hit(t) = 0 if no hit
1 if hit

$$P\,(t) = (1-\alpha)p(t-1)+\alpha\_hit\,(t) \qquad (9)$$

where $0<\alpha<1$

SRED estimates p (t)-1 for effective number of active flow. Suppose there are many flows numbered 1,2,3.....n. Suppose that every time the packet arrives it belongs to the same flow flowi with the probability

Therefore every arriving packet the probability that it cause a hit is:

$$P\{Hit\,(t)=1\}= \qquad (10)$$

To reduce the overhead SRED update p (t):

$0<=p(t)<=1/256$

SRED calculates drop probability with following formula.

Let the buffer capacity be B bytes. A function PSRED (q) is defined as follows:

PSRED (q) = Pmax if 1/3 B <= q <= B
1/4 * Pmax if 1/6 B <= q <= 1/3 B
0 if 0 <= q <= 1/6B

Pmax is chosen as 0.15

q is total bytes in buffer

B is capacity of buffer in Bytes

Hence SRED calculates the drop probabilty with following equation for simple RED

$$P\,(zap) = Psed\,(q) * \min\,(1,\,1/256*P\,(t)2) \qquad (11)$$

In full SRED the drop probability is calculated as:

$$P\,(zap\,) = PSRED\,(q) * \min(\,1,1/256*p\,(t)2) * (1+(Hit\,(t)/P\,(t))) \qquad (12)$$

2.4.1 *Algorithm*

```
Parameters
int M = 1000
double p_t_       hit frequency
double p_max      maximum drop frequency
double aipha      alpha = p/M = p_overwrite/M
int bcount_       byte count
int qib           queue measure in bytes.
int count         count of pkts of flow in zombie
int fid           flow identifier
qweight = 0.002
thresh = 5
max_thresh = 15
mean_pktsize = 500
int curq_         current queue size

Step 1: Create a Zombie list check if (list size_ <M)
        if true add identifier (source address, destination
        address) to the zombie list.
        Intially count = 0
        Zombie list timestamp = arrival time of the
        packet.
        increment list size by 1
        ++listsSize
Step 2: Check the curq<qlim_      (qlim_ is qlimit)
        if true enque the packet
        else drop the packet
Step 3: Select randomly any packets from zombie list. And
        compare it with newly arrived packet
        if the flow id matches then it is hit
        if (Zombie list <-[index].fid = fid)
        hit = 1
        Zombie list [index]. count++
        set the timestamp to latest arrival time of packet
        else
        it is a miss
        hit = 0
        overwrite the zombie list with newly arrived
        packet with random probability.
        Set
        Zombie list [index].fid = fid
        Zombie list [index]. timestamp = Schedular::insta
        nce().clock()
Step 4: Update hit frequency
        p_t = (1- appha_) * p_t_+ alpha_ * hit
        (Note that value of alpha is p_overwrite/M)
Step 5: Intialize len, lim, lim_3,lim_6
        Len = curq*536
        Lim = qlim_*536
        lim_3 = lim/3
        lim_6 = lim/6
        Step 5.1: if ((len > = lim_3) && (len < lim))
        return p_max_;
        Step 5.2: else if ((len > = lim_6) && (len <
        lim_3))
        return (p_max_/4);
        Step 5.3: else if ((len > = 0) && (len < lim_6))
        return 0
Step 6 : Calculate drop probability for simple sred.
        Double factor
        factor = 256 * p_t_
        factor = factor*factor
        factor =1/factor
        if(factor>1)
        factor = 1
        return (prob_sred*factor);
Step 7 : Calculate drop probability for full Sred
        double prob_zap = calc_simple_pzap(prob_sred)
        prob_zap* = (1+hit/p_t_)
        return prob_zap
```

2.5 Flow RED (FRED)

FRED is Flow based Random Early Detection which is modified version of RED, which was developed by lin and Morris which uses per active flow accounting to make dropping decision for different active flow accounting to make dropping decision for different active flow in the queue depending on their bandwidth usage. FRED keeps a track of each flow and bandwidth usage of each flow that are inside the queue hence the cost of FRED is independent from number of flows but is proportional to the buffer size. FRED was developed as an alternative to RED algorithm to protect from number of fragile flow and to maintain high degree of fairness.

Additional parameters that are included in FRED are:

Min q: Min q represents minimum number of packet that each flow i is allowed to buffer in the queue.

Max q: Max q represents maximum number of packets that each flow i is allowed to buffer in the queue.

Avgcq - Is a global variable that estimates the per flow packets to be buffered in the queue. The flow i have less packets to be queued in buffer that avgcq is favored than the flows whose packet count to queued is greater than avgcq.

Qlen (i) – This variable maintains the count of buffered packets for each flow.

Strike (i) - For each flow i strike (i) is a count for number of times the flow has failed to respond to congestion notification.

If the strike (i) value for the flow increases FRED penalizes such flow.

Nactive - FRED estimates active scavenger service flow number by the variable nactive in FRED.

2.5.1 Algorithm

```
Constants
Wq = 0.002
Minth = MIN (buffersize/4, RTT)
Maxth = 2*minth
Maxp = 0.02
Minq = 2 for small buffer
4 for large buffer
Global variables
q - current queue size
time – current real time
avq – average queue size
count – number of packets since last drop
avgcq – average per flow queue size
maxq – maximum allowed perflow queue sizePer Flow variables:
qleni- nu ber of packets buffered
stikei: count of number of times the flow has failed to respond to congestion notification
Mapping function
Conn (P):- Connection id of packet P.
F (time): linear function of time

Step1: For each arriving packet P check if flow i = Conn
       (P) has no state table then set qleni = 0 and strikei
       = 0.
Step2: Check if queue is empty if true calculate average
       queue length avg.
Step 3: set maxq = minth
        if (avg > = maxth)
        set maxq = 2
Step 4: Identify and manage non adaptive flow check.
        If (qleni> = maxq || (avg > = maxth && qleni >
        2*avgcq) || cqleni > = avgcq && strikei >1))
        set strikei = strikei+1
        drop packet p
Step 5: Operate in random drop mode.
        check if (minth< = avg<maxth)
        set coun t= count+1
Step 6: For only random drop from robust flow do the
        following.
        check if (qleni > = MAX (minq, avgcq))
        calculate probability Pa
        pb = maxp (avg-minth)/ (maxth-minth)
        pa = pb/ (1-count*pb)
        with probability pa drop the packet p, set count 0
        else check if (avg<minth)
        no drop mode
        set count = -1
        else
        drop tail mode , set count = 0, drop packet p
        if (qleni = 0)
        set Nactive = Nactive+1
        calculate average queue length, accept the packet p
Step 7: For each departing packet p Calculate average
        queue length.
        If (qleni = 0)
        set Nactive = Nactive+1
        delete the state table for flow i
Step 8: Calculate average queue length
        check if (q || packet departed)
        calculate avg
        avg = (1-wq) * avg + wq * q
        else
        set m = f (time-q_time)
        avg = (1-wq)m*avg
        q_time = time
        check if (Nactive)
        avgcq = avg/Nactive
        else
        avgcq = avg
        acvcq = MAX(avgcq,1)
        if(q = 0 && packetdeparted)
        q_time = time
```

2.6 BLUE

BLUE is an AQM algorithm in which queue management is done base on the link utilization and number of packets dropped. BLUE maintains variable pm to estimate marking probability for either marking the packet or dropping the packet. When queue becomes full it starts dropping the packets. When queue becomes full it starts dropping the packet and pm is incremented by factor δ1. If the queue is empty pm is decremented by the factor δ2. The value of δ1 is set such a way that δ1 > δ2.

BLUE uses one more parameter called freeze time which determines the time interval between two successive updates of freeze time.

2.6.1 BLUE Algorithm

```
Step 1: Upon Packet loss or (Qlen>L) event
        if((now_last_update)>freeze_time)
        pm = pm+δ1
        last_update = now
Step 2: upon link idlee vent
        If (now lastupdate)>freeze_time)
        Pm = pm-δ2
        Last update = now
```

2.7 Stochastic Fair BLUE (SFB)

SFB is another AQM algorithm which protects the TCP flows against the non-responsive flow using BLUE AQM algorithm. SFB algorithm identifies and rate limits the non-responsive flow and mechanism used to identify this non-responsive flow is same as the accounting mechanism used in BLUE algorithm. SFB maintains N*L accounting bins where L is the number of level and N is the number of bins in each level. SFB also maintains L independent hash functions each associated with one level of accounting bin.

SFB maintains a variable called pm which keep a track of marking/dropping probability in each bin. When a new packet arrives it is mapped into one of N bins in each of the l levels. When the number of packets mapped to a bin goes above certain threshold value pm is increased. If the number of packets drops to zero the pm is decreased.

2.7.1 SFB Algorithm

```
B[l][n]: L*N arrays of bins (L levels, N bins per level)
enqueue()
Calculate hash function values h0,h1
update bin at each level
for I = 0 to L-1
if(B[i][hi]*pm+ = delta
drop packet
else if(B[i][hi]*qlen = 0)
B[i][hi]pm = delta
Pmin = min(B[0][h0].pm.... B[L][hL]*pm)
if(pmin = 1)
rate limit()
else
mark/drop with probability pmin
```

3. Comparison

Table 1. Comparison of AQM Algorithm with respect to link utilization, fairness, space requirement, per flow state information, complexity, configuration complexity

Sr No	Algorithm	Link Utilization	Fairness	Space Requirement	Per Flow State Info Mation	Complexity	Configuration Complexity
1	RED	GOOD	UNFAIR	LARGE	NO	HIGH Q Sampling frequency	HARD
2	SFQ	GOOD	FAIR	LARGE	NO	HIGH Q Sampling frequency	HARD
3	REM	GOOD	FAIR	SMALL	NO	LOW Queue Sampling frequency	EASY
4	FRED	GOOD	FAIR	SMALL	YES	HIGH Queue Sampling frequency	EASY(adaptive)
5	SRED	GOOD	FAIR	LARGE	NO	HIGH Queue Sampling frequency	EASY
6	BLUE	GOOD	UNFAIR	SMALL	NO	HIGH Queue Sampling frequency	EASY
7	SFB	GOOD	FAIR	LARGE	NO	HIGH Queue Sampling frequency	HARD

Table 2. Comparison of AQM algorithm with respect to advantages and disadvantages

Sr No	Algorithm	Advantages	Disadvantages
1	RED	1. Early congestion detection 2. RED algorithm avoids bias against busty traffic 3. TCP global synchronization that occurs in drop-tail algorithm is overcome in TCP RED AQM algorithm	1. Difficulty in parameter setting 2. Insensitive to the busty traffic.
2	SFQ	1. Simple in implementation for fair queue algorithm family 2. Fair distribution of available bandwidth among all the sub-stream 3. SFQ algorithm is useful for those network where utilization of link capacity on different source is equal. 4. SFQ have low end-to-end delay so these queue mechanism can be used in delay sensitive application	1. Loss rate of the packet is high 2. Congestion window fluctuation is more
3	REM	1. Low computational load on the system 2. Achieves both high link utilization and negligible loss and low delay.	1. Low throughput for web traffic 2. Inconsistency with TCP sender mechanism works best with ECN.
4	FRED	1. Protects from fragile flow and maintains high degree of fairness	1. Maintain per flow state 2. RED disadvantages
5	SRED	1.Stabilized queue occupancy 2. Protection from misbehaving flow 3. Detects the flow that take more bandwidth and a fair share of bandwidth without performing much computation	1. Maintains additional list called as zombie list 2. RED disadvantages.
6	BLUE	1. HIGH throughput 2. Maintain small queue	1. BLUE algorithm uses link utilization and packet loss history instead of queue length to manage congestion 2. Not Scalable
7	SFB	1. Protects TCP flow against non-responsive flow 2. Identifies and rate limits the non-responsive flow 3. Enforces the fairness among the flow 4. No additional overhead is required in packet header like SFQ AQM algorithm	1. SFB needs to reconfigured with non-responsive flow 2. Bandwidth requirement for non-responsive flow depends on the parameter Box-time 3.Box-time is a static parameter which can only be set manually and cannot be configured automatically. The suitable value of Box-time is one for one case and may be different for other case. This is a major drawback of SFB.

4. Conclusion and Future Work

In this paper we have analyzed compared and surveyed various AQM algorithm like RED, SFQ, REM, FRED, SRED, BLUE and SFB. After analyzing it was found that when RED algorithm was compared with SFQ and REM it (RED) achieved best result in terms of delay. SFQ had minimum average ratio and RED had maximum loss ratio. REM algorithm showed best result with respect to throughput, loss ratio and link utilization than RED and SFQ. RED AQM algorithm does not stabilize the queue size while SRED stabilizes the queue size. RED algorithm monitors the queue length while SRED algorithm monitors queue length and packet header. BLUE AQM algorithm greatly reduces the buffer requirement needed to support differentiate service. FRED AQM algorithm records per active flow information. SFB statistically multiplex buffer to bins, but need to be reconfigured with large number of non-responsive flows. This paper tries to compare each AQM algorithm and projects the desirable quality and short comings that exists in each algorithm of their performance. After performing a comparative analysis it was observed that no single congestion control can solve all of the problems hence more research is needed to be carried out in this area.

4.1 Future Work:

We have planned to develop a new algorithm by doing hybridization of RED, REM, SFB, BLUE AQM algorithm so that the new algorithm can take the advantages of the existing algorithms and provide a better result.

5. References

1. Floyd S, Jacobson V. Random Early Detection gateways for congestion avoidance. IEEE/ACM Transactions on Networking. 1993 Aug; 1(4):397–413.
2. Ismail AH, Elsagheer Z, Morsi IZ. Survey on Random Early Detection mechanism and its variants. IOSR Journal of Computer Engineering (IOSRJCE). 2012 July-Aug; 2(6):20–4. ISSN: 2278-0661.
3. Patel SP, Gupta K, Garg A, Mehrotra P, Chhabra M. Comparative analysis of congestion control algorithms using ns-2. IJCSI International Journal of Computer Science Issues. 2011 Sep; 8(5(1)):89–94.
4. Ahammed GFA, Banu R. Analyzing the performance of active queue management algorithm. International Journal of Computer Networks and Communications (IJCNC). 2010 Mar; 2(2):1–19.
5. Kiruthiga B, Raj EGDP. Survey on AQM congestion control algorithms. International Journal of Computer Science and Mobile Applications. 2014 Feb; 2(2):38–44.
6. Lin D, Morris R. Dynamics of Random Early Detection. Proceedings of SIGCOMM' 97. 1997 Oct; 27(4):127–37.
7. Adams R. Active Queue Management: A Survey. IEEE Communications Surveys and Tutorials. 2013; 15(3):1425–76.
8. Ratneshwer VK. A review of router based congestion control algorithms. I.J Computer Network and Information Security. 2014; 1:1–10.
9. Socrates C, Devamalar PM, Sridharan RK. Congestion control for packet switched networks: A survey. International Journal of Scientific and Research Publications. 2014 Dec; 4(12):1–6.
10. Li M, Wang H. Study of active queue management algorithms -Towards stabilize and high link utilization in communication magazine. IEEE. 2002 Jun.
11. Ott TJ, Lakshman TV, Wong LH. SRED: Stabilized RED. INFOCOM 99 Eighteenth Annual Joint Conference of the IEEE Computer and Communication Societies Proceedings IEEE. 1999 Mar 21-25; 3: p. 1346–55.
12. Santhi V, Natarajan AM. Performance analysis of active queue management algorithms. International Journal on Information Sciences and Computing. 2009 Jan; 3(1).
13. Meckenney PE. Stochastic fair queueing. Proceeding of INFCOM: San Francisco, CA; 1990 Jun 3-7. p. 733–40.
14. Nejakar SM, Sharanabasappa RR, Harshavardhan DR. Development of modified RED AQM algorithm in computer network for congestion control. International Journal of Innovative Research in Advanced Engineering (IJIRAE). 2014 Sep; 1(8):380–5.
15. Feng WC, Shin KG, Kandlur DD, Saha D. The BLUE active queue management algorithms. IEEE/ACM Transactions on Networking. 2002 Aug; 10(4):513–28.
16. Athuraliya S, Li VH, Low SH, Yin Q. REM: Active Queue Management. IEEE Network. 2001 May-Jun; 48–53.
17. Lin D, Morris R. Dynamics of Random Early Detection. Proceedings of ACM SIGCOMM 97 Cannes France. 1997 Oct; 27(4): p. 127–37.
18. Yang C, Reddy A. A taxonomy for congestion control algorithms in packet switching networks. Network. IEEE. 1995 July/Aug; 9(4):34–45.
19. Ryu S, Rump C, Qiao C. Advances in Active Queue Management (AQM) based TCP congestion control. Telecommunication Systems - Modeling, Analysis, Design and Management. 2004 Mar; 25(3):317–51.
20. Feng G, Agarwal A, Jayaraman A, Siew C. Modified RED gateways under bursty traffic. IEEE Communications Letters. 2004 May; 8(5):323–5.
21. Hollot C, Liu Y, Misra Y, Towsley D. Unresponsive flows and AQM performance. Proceedings of the Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM): San Francisco, CA, USA; 2003 Mar 30-Apr 3, p. 85–95.
22. Nabeshima M. Improving the performance of active buffer management with per-flow information. IEEE Communications Letters. 2002 Jul; 6(7):306–8.
23. Claypool M, Kinicki R, Hartling M. Active Queue Management for Web traffic. Proceedings of the IEEE International Performance, Computing and Communications Conference.: 2004 Apr p. 14–7.
24. Ziegler T. On averaging for active queue management congestion avoidance. Proceedings of the Seventh International Symposium on Computers and Communications (ISCC): Taormina-Giardini Naxos, Italy: 2002. p. 867–73.

25. Aweya J, Ouellette M, Montuno D, Chapman A. An adaptive buffer management mechanism for improving TCP behavior under heavy load. Proceedings of the IEEE International Conference on Communications: Helsinki; 2001.p. 3217–23.
26. Low S. A duality model of TCP and queue management algorithms. IEEE/ACM Transactions on Networking. 2003 Aug; 11(4):525–36.
27. Zhu C, Yang OWW, Aweya J, Ouellette M, Montuno DY. A comparison of Active Queue Management algorithm using OPNET modeler. Best paper award in OPNET 2001. 2002 Jun; 40(6):158–67.
28. Indumati P, Shanmugel S, Mahesh HC. Buffered leaky bucket algorithm for congestion control in ATM network. IETE Journal of Research. 2002; 48(1):59–67.
29. On 3026 Jul. Available from: sred.cchttps://www.cs.purdue.edu/homes/fahmy/software/aqm/sred.cc
30. 2015 May. Avaulable from: fred.cchttps://www.cs.purdue.edu/homes/fahmy/software/aqm/fred.c