COMPARISON OF ACTIVE QUEUE MANAGEMENT IN NS2

Minakshi Mehta¹, Gaurav Deep²

Dept. of Computer Science and Engineering Dept. of Computer Science and Engineering, Punjabi university, Patiala Punjabi University, Patiala

ARTICLE INFO

Corresponding Author:
Minakshi Mehta
Dept. of Computer Science and Engineering, Dept. of Computer Science and Engineering, Punjabi university, Patiala Punjabi University, Patiala
minakshimeha02@gmail.com

Key Words: Active Queue Management, Congestion control, RED, BLUE, FRED, SFB and Choke.

DOI: http://dx.doi.org/10.15520/ajcsit.v7i4.68

ABSTRACT

The internet and its applications are an important part of one's daily life. Today's internet needs to provide best effort service. The demands for quality of service have rapidly developed due to the rapid transformation of the Internet into a commercial infrastructure. Traffic is processed as quickly as possible, but there is no guarantee of timelines or actual delivery. As the masses of modern world are very much dependent on various network services like VOIP, video conferencing and file transfer. So to prevent the problem of congestion control and synchronization, various active queue management (AQM) techniques are used. AQM algorithms execute on network routers and detect initial congestion by monitoring some functions. When congestion occurs on the link the AQM algorithm detects and provides signals to the end systems. In this paper, we evaluate several queue management algorithms with respect to their abilities of maintaining high resource utilization, identifying and restricting disproportionate bandwidth usage, and their deployment complexity. We compare the performance of RED, FRED, BLUE, SFB, and CHOKE based on simulation results. The characteristics of different algorithms are also discussed and compared.

©2017, AJCSIT, All Right Reserved.

INTRODUCTION

The strength of today's web depends heavily on the transmission control protocol congestion control mechanism. Nevertheless, as a lot of UDP applications (e.g. packet audio/video applications) are deployed on the net, masses cannot rely on end users to incorporate proper congestion control. Router mechanisms should be provided to shield responsive flows from non-responsive ones, also stop "Internet meltdown". Numerous strategies are advised for the management of shared resources on the web, active queue management is one amongst the main approaches.

Traffic on the Internet tends to variate and to be greedy. Ideally, a router queue management algorithm should allow temporary bursty traffic, and penalize flows that persistently overuse bandwidth. Also, the algorithm should prevent high delay by restricting the queue length, avoid underutilization by allowing temporary queueing, and allocate resource fairly among different types of traffic [1]. In practice, largely routers being deployed use oversimplified Drop Tail algorithm that is easy to implement with nominal computation overhead, however, provides inadequate performance.

To attack this issue, tremendous queue management algorithms are proposed, like Random Early Drop (RED) [2], Flow Random Early Drop (FRED) [3], BLUE [4], Stochastic Fair BLUE (SFB) [4], and CHOKE (Choose and Keep for responsive flows, Choose and Kill for unresponsive flows) [5].

Most of the algorithms claim that they'll offer fair sharing among totally different flows without imposing an excessive amount of deployment complexity. Most of the proposals specialize in only one aspect of the matter (whether it's fairness, deployment complexity, or computational overhead), or fix the imperfections of old algorithms, and their simulations setting are completely different from one another. These all make it difficult to evaluate and to choose one to use under certain traffic load.

This paper aims at a thorough evaluation of these algorithms and demonstration of their characteristics by simulation. We compare the performance of FRED, BLUE, SFB, and CHOKE, using RED and Drop Tail as the evaluation baseline. The main three features are discussed for these all algorithms that are resource utilization, fairness among different traffic flows and implementation and deployment complexity.

I. QUEUE MANAGEMENT ALGORITHMS

A. RED (Random Early Drop)

RED [6] was made with an aim to firstly, lessen the packet loss and queuing delay. Secondly, to avoid global synchronization of sources and to maintain high link utilization, eventually togets rid of biases against burst sources. The fundamental set up behind RED queue management is to find nascent congestion early moreover to take congestion notification to the end-hosts, allowing them to scale back their rates of
transmission ahead queues within the network overflow also packets are dropped.

To perform it, RED maintains an exponentially-weighted moving average (EWMA) of the queue length that it uses to find out the congestion. Once the common queue length exceeds a minimum threshold ($min_a$), packets are dropped willy-nilly or marked with an explicit congestion notification (ECN) bit [2]. Once the typical queue length exceeds a maximum threshold ($max_a$), all packets are dropped or marked.

While RED is undoubtedly an improvement over previous drop tail queues, it has various limitations. One of the basic issues with RED is that they accept queue length as a calculator of congestion. Whilst, the presence of a persistent queue indicates congestion, its length provides packet-size information in addition to the severity of congestion. That is, the count of competitive connections sharing the link. During a busy period, merely one source transmitting at a rate greater than the bottleneck link capability can cause a queue to build up just as simple as a huge range of sources can. Since the RED algorithm depends on queue lengths, it has an inherent problem in determining the severity of congestion. Consequently, RED needs a great number of parameters to operate in a correct manner under various congestion situations. Whereas RED can attain a perfect operating point, it can solely do so when it has an enough amount of buffer space and is parameterized in an accurate manner.

RED represents a category of queue management mechanisms that doesn’t keep the state of each flow. That is, they put the data from the all the flows into one queue, and emphasize on their overall performance. It is that which originate the hurdles caused by non-responsive flows. To deal with that, a few congestion control algorithms have tried to separate a various kind of data flows, to exemplify Fair Queue [7], Weighted Fair Queue [7], etc. But their per-flow-scheduling doctrine is not same as that of RED.

B. FRED (Flow Random Early Drop) Flow Random Early Drop (FRED) [6] is a changed version of RED that utilizes per-active-flow accounting to make various dropping choices for connections with several bandwidth usages. FRED solely keeps track of flows that have packets within the buffer; hence the amount of FRED is proportional to the buffer size and not depends on the total flow numbers. FRED can be able to acquire the merits of per-flow queuing along with round-robin scheduling with considerably less complexity.

The other attention-grabbing characteristics of FRED are penalizing non-adaptive flows by imposing a maximum count of buffered packets, and exceeding their share to average per-flow buffer usage; protective fragile flows by deterministically accepted flows from low information measure connections; providing truthful sharing for huge numbers of flows by victimization “two-packet-buffer” once buffer is used up; fixing various imperfections of RED by estimate average queue length at each packet arrival and departure.

Two different parameters are brought in FRED: $min_a$ and $max_a$ which are minimum and maximum number of packets that each flow is allowed to buffer. In order to track the average per-active-flow buffer usage, FRED uses a global variable $Avg-CQ$ to estimate it. It maintains the number of active flows, and for each of them, FRED maintains a count of buffer packets, $glen$, and a count of times when the flow is not responsive ($glen-max$). FRED will penalize flows with high strike values. FRED processes arriving packets using the following algorithm:

![Fig1: FRED processing arriving packet](image)

C. BLUE

BLUE is an active queue management algorithm to manage congestion control by packet loss and link utilization history instead of queue occupancy. BLUE maintains a single probability, $P_m$, to mark (or drop) packets. If the queue is continually dropping packets due to buffer overflow, BLUE increases $P_m$, thus increasing the rate at which it sends back congestion notification or dropping packets. Conversely, if the queue becomes empty or if the link is idle, BLUE decreases its marking probability. This effectively allows BLUE to “learn” the correct rate it needs to send back congestion notification or dropping packets.

The typical parameters of BLUE are $d_1$, $d_2$, and $freeze_time$. $D1$ determines the amount by which $P_m$ is increased when the queue overflows, while $d_2$ determines the amount by which $P_m$ is decreased when the link is idle. $freeze_time$ is an important parameter that determines the minimum time interval between two successive updates of $P_m$. This allows the changes in the marking probability to take effect before the value is updated again. Based on those parameters. The basic blue algorithms can be summarized as following:

![Upon link idle event](image)

![Upon packet loss event](image)

D. SFB

Based on BLUE, Stochastic Fair Blue (SFB) is a novel technique for providing TCP flows against non-responsive flows. SFB is a FIFO queuing algorithm that identifies and rate-limits non-responsive flows based on accounting mechanisms similar to those used with BLUE. SFB maintains accounting bins. The bins are organized in $L$ levels with $N$ bins in each level. In addition, SFB maintains $L$ independent hash functions, each associated with one level of the accounting bins. Each hash function maps a flow into one of the accounting bins in that level. The accounting bins are used to keep track of queue occupancy statistics of packets belonging to a particular bin. As a packet arrives at the queue, it is hashed into one of the $N$ bins in each of the $L$ levels. If the number of packets mapped to a bin goes above a certain threshold (i.e., the size of the bin), the packet dropping probability $P_m$ for that bin is increased. If the number of packets in that bin drops to zero, $P_m$ is decreased. The observation is that a non-responsive flow quickly drives $P_m$ to 1 in all of the $L$ bins it
is hashed into. Responsive flows may share one or two bins with non-responsive flows, however, unless the number of non-responsive flows is extremely large compared to the number of bins, a responsive flow is likely to be hashed into at least one bin that is not polluted with non-responsive flows and thus has a normal value. The decision to mark a packet is based on $P_{\text{min}}$, the minimum $P_m$ value of all bins to which the flow is mapped into. If $P_{\text{min}}$ is 1, the packet is identified as belonging to a non-responsive flow and is then rate-limited.

The typical parameters of SFB algorithm are $Qlen$, $Bin\_Size$, $d_1$, $d_2$, $freeze\_time$, $N$, $L$, $Boxtime$, $Hinterval$. $Bin\_Size$ is the buffer space of each bin. $Qlen$ is the actual queue length of each bin. For each bin, $d_1$, $d_2$ and $freeze\_time$ have the same meaning as that in BLUE. Besides, $N$ and $L$ are related to the size of the accounting bins, for the bins are organized in $L$ levels with $N$ bins in each level. $Boxtime$ is used by penalty box of SFB as a time interval used to control how much bandwidth those non-responsive flows could take from bottleneck links. $Hinterval$ is the time interval used to change hashing functions in our implementation for the double buffered moving hashing. Based on those parameters, the basic SFB queue management algorithm is shown in the above table.

E. CHOKe

As a queue management algorithm, CHOKe [5] differentially penalizes non-responsive and unfriendly flows using queue buffer occupancy information of each flow. CHOKe calculates the average occupancy of the FIFO buffer using an exponential moving average, just as RED does. It also marks two thresholds on the buffer, a minimum threshold $min_{th}$ and a maximum threshold $max_{th}$. If the average queue size is less than $min_{th}$, every arriving packet is queued into the FIFO buffer. If the aggregated arrival rate is smaller than the output link capacity, the average queue size should not build up to $min_{th}$ very often and packets are not dropped frequently. If the average queue size is greater than $max_{th}$, every arriving packet is dropped. This moves the queue occupancy back to below $max_{th}$. When the average queue size is bigger than $min_{th}$ each arriving packet is compared with a randomly selected packet, called drop candidate packet, from the FIFO buffer. If they have the same flow ID, they are both dropped. Otherwise, the randomly chosen packet is kept in the buffer (in the same position as before) and the arriving packet is dropped with a probability that depends on the average queue size. The drop probability is computed exactly as in RED. In particular, this means that packets are dropped with probability 1 if they arrive when the average queue size exceeds $max_{th}$. A flow chart of the algorithm is given in Figure 2. In order to bring the queue occupancy back to below $max_{th}$ as fast as possible, we still compare and drop packets from the queue when the queue size is above the $max_{th}$.

CHOKe has three variants:

1) **Basic CHOKe (CHOKe):** It behaves exactly as described in the above, that is, choose one packet each time to compare with the incoming packet.

2) **Multi-drop CHOKe (M-CHOKe):** In M-CHOKe, $m$ packets are chosen from the buffer to compare with the incoming packet, and drop the packets that have the same flow ID as the incoming packet. Easy to understand that choosing more than one candidate packet improves CHOKe’s performance. This is especially true when there are multiple non-responsive flows; indeed, as the number of non-responsive flows increases, it is necessary to choose more drop candidate packets. Basic CHOKe is a special case of M-CHOKe with $m=1$.

3) **Adaptive CHOKe (A-CHOKe):** A more sophisticated way to do M-CHOKe is to let algorithm automatically choose the proper number of packets chosen from buffer. In A-CHOKe, it is to partition the interval between $min_{th}$ and $max_{th}$ into $k$ regions, $R_1, R_2, ... , R_k$. When the average buffer occupancy is in $R_k$, $m$ is automatically set as $2i (i = 1, 2, ... , k)$.

![Fig2](image-url) Flowchart for Basic Choke

### III. SIMULATION AND COMPARISON

In this section, we are going to compare the performances of FRED, BLUE, SFB, and CHOke. We use RED and Drop Tail as the evaluation baseline. Our simulation relies on NS-2 [8]. Both RED and FRED have an implementation for NS-2. BLUE as well as SFB are purely carried out in a former version of NS, NS-1.1, and are re-implemented in NS-2. On the basis of Choke paper [5], we enforced Choke in NS-2. In our simulation, ECN support is not abled, and “marking a packet” indicates “dropping a packet”.

#### A. Simulation Settings

As various algorithms have various preferences or assumptions for the network configuration and traffic pattern, one of the demanding or stimulating situations in designing or planning our simulation is to pick up a typical set of network topology and parameters (link bandwidth, RTT, and gateway buffer size), also load parameters (count of Transmission Control Protocol and UDP flow, packet size, Transmission Control Protocol window size, traffic patterns) as the basis for analysis.

![Fig3](image-url) Simulation topology
The network topology we employed is a classic dumbbell configuration as shown in Figure 3. This is often a typical scenario that tremendous types of traffic share a bottleneck router. TCP (FTP application in particular) and UDP flows (CBR application in particular) are taken as typical traffic patterns.

In our simulation, we use ten Transmission Control Protocol flows and one UDP flow. The bottleneck link in this scenario is the link between two gateways. We set TCP window size as fifty packets and the router queue buffer size in the simulation as 150 packets (the packet size for both TCP and UDP are 1000 bytes). For RED, we conjointly ought to opt for values for min_0 and max_0, which are typically set as 20% and 80% queue buffer size. In the following, we set them as 50 and 100 packets.

B. Metrics
Throughput and queue size are the two metrics having greater importance in our simulations. The throughput of each flow is used to demonstrate the fairness among numerous flows, while the total throughput can be compared with the bottleneck bandwidth as an indicator of resource utilization. Queue size is a directly proportional resource utilization. The average queue sizes of each flow overt the fairness of router resource allocation, which also depicts the tremendous characteristics of various algorithms. We calculate the average queue size using the exponentially weighted average (EWMA), and the aging weight is set to 0.002.

C. Comparison
Figure 4 and Figure 5 show the most important outcomes of the simulation. The entire throughput values of all TCP and UDP flows don’t seem to be overt here. For all the simulations, the entire throughputs are moderately high (about 90-96% of accessible bandwidth), indicating that every one of these algorithms provide high link utilization. Figure 4.1 elucidates the UDP throughput and queue length beneath simulations using ten TCP flows, one UDP flow, when UDP sending rate fluctuates from 0.1Mbps to 8Mbps. On the basis of this diagram, Drop Tail is the worst in terms of unfairness that provides no protection for adaptive flows and yields the dominant UDP throughput. RED as well as BLUE does not work considerably under high UDP sending rate. When UDP sending rate is above the bottleneck bandwidth, UDP flow quickly dominates the transmission on the bottleneck link, and TCP flows could solely share the remaining bandwidth. On the flip side, FRED, SFB and CHO Ke properly penalize UDP flow, and TCP could accomplish their reasonable share.

One fascinating point in Figure 4.1 is the behaviour of Choke. UDP throughput plunge with upsurge of UDP rate from 2Mbps to 8Mbps. This is because, with the raise of UDP rate, the total count of packets selected to compare uplift, which will surging the dropping probability for UDP packets, and plummet UDP flow throughput as a result. Figure 4.2 demonstrates the dimension of queue buffer attained by UDP flow. It appears that buffer usage is a good indicator of link bandwidth utilization. Likewise Figure 4.1, Drop Tail is the worst in fairness. Although RED and BLUE are alike in not preventive to non-responsive flows, BLUE uses much less buffer. FRED and SFB are also the fairest.

Figure 5 illustrates the common queue size for UDP and TCP flows additionally the total buffer usage. The distinction of algorithms is apparently captured in the buffer usage plots. We can see, for Drop Tail, RED, and BLUE, mostly packets in the queue are UDP flow packets, whereas solely a small ratio belongs to TCP flows. FRED, SFB and CHO Ke effectively penalize UDP flow and permit TCP flows to occupy a maximum throughput. It is conjointly fascinating to notice the difference among the full queue sizes. Since Drop Tail barely drops packets once the queue buffer is full, at most time, its total queue size is the maximum queue buffer size. For RED, although it initiates to provide congestion notification when the queue size reaches min_0, it solely affects TCP flows, whilst UDP flow will keep the same sending rate, which drives the total queue size to max_0 quickly, after which all the incoming packets will be dropped and the entire queue size will be kept at max_0. In Choke, nevertheless, the random packet selection mechanism effectively penalizes UDP flow after the average queue size reaches min_0.

What’s more, UDP dropping rate is proportional to its incoming rate, which will effectively keep the total queue size around min_0 as illustrated in Figure 5f. FRED, BLUE, and SFB are not directly affected by min_0 and max_0 settings, so their total queue sizes have no obvious relation to these two parameters in Figure 5.

In some of the figures in Figure 5 where TCP flow queue size is very extremely tiny, UDP flow queue size is constant as that of the total queue size, but the corresponding queue size for TCP flows aren’t zero, which seems to be a contradiction. The cause is that we draw these figures using the EWMA value of the queue size. Although we compute the queue size every time we get a new packet, solely EWMA value (weight = 0.002) is plotted\(^2\). It is EWMA that eliminates the difference between UDP flow queue size and total queue size when TCP flow queue size is very small.

---

\(^1\) Due to the method for changing the UDP rate in ns-2, the sample intervals we choose are not uniform, but they will not affect our analysis.

\(^2\) The figures of the actual queue size has a lot of jitters and difficult to read.
V. CONCLUSION

This paper compared several queue management algorithms (RED, FRED, BLUE, SFB, CHOKe) based on simulation results. We have demonstrated our comparison results in simulation setting, along with algorithm characteristics. It’s still hard to summarize that which algorithm is better in all aspects than another, especially considering the deployment complexity. However, the main trends are all these algorithms give high link utilization and RED and BLUE do not identify and penalize non-responsive flow, whereas the remaining algorithms (three) maintain fair sharing among numerous traffic flows, the fairness is achieved using completely different methodologies, FRED record per-active-flow information, SFB statistically multiplex buffers to bins, whilst has to be reconfigured with huge number of non-responsive flows, CHOKe correlates dropping rate with corresponding flow’s incoming rate, and is in a position to penalize sizable amount of non-responsive flows adaptively, all of the algorithms has computation overhead per incoming packet, their space necessities are different. The following tabular chart recapitulates our evaluation outcomes:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>RED</th>
<th>FRED</th>
<th>BLUE</th>
<th>SFB</th>
<th>CHOKe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal of Algorithm</td>
<td>Optimized for cell-based architecture in ATM networks.</td>
<td>Make RED fair.</td>
<td>Low loss rates and low queue length oscillation.</td>
<td>To detect the non-responsive flows</td>
<td>To maximize the system capacity</td>
</tr>
<tr>
<td>Link Utilization</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Fairness</td>
<td>Unfair</td>
<td>Fair</td>
<td>Unfair</td>
<td>Fair</td>
<td>Fair</td>
</tr>
<tr>
<td>Complexity</td>
<td>HIGH Q Sampling frequency</td>
<td>HIGH Q Sampling frequency</td>
<td>HIGH Q Sampling frequency</td>
<td>HIGH Q Sampling frequency</td>
<td>HIGH Q Sampling frequency</td>
</tr>
<tr>
<td>Computational Overhead</td>
<td>Arrival</td>
<td>Arrival-departure</td>
<td>Freeze-time</td>
<td>Freeze-time</td>
<td>Arrival</td>
</tr>
<tr>
<td>Space requirement</td>
<td>Large</td>
<td>Small</td>
<td>Small</td>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td>Malicious-aware</td>
<td>No, $P_{\text{drop}}$ increases linearly with the utilization of bandwidth.</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Pre-flow state information</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 5: Queue size in different algorithms (Notice that the total queue sizes of different algorithms are different)
| Special characteristic | Takes the characteristics of ATM networks into account. | RED combined with per-flow state. | Hash-bin based detection of greedy flows | Enforce fairness among a large number of flows. | Foster reciprocation and is robust to free riders. | Configuration complexity | Hard | Easy | Easy | Hard | Easy |

**ACKNOWLEDGMENT**

The authors are thankful to the Punjabi University, Patiala for its kind support to carry out this research work. The authors also acknowledge the expert opinions of their colleagues to complete this research work.

**REFERENCES**


[15] A. Legout, "Rarest First and Choke Algorithms Are Enough".


