Volume 2, Issue 5, May 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Classification of Active Queue Management Algorithms for Application Specific Traffics

Saurabh Sarkar, Mrs. Geeta Sikka, Ashish Kumar
Department of CSE
Dr. B.R.Ambedkar, NIT Jalandhar, India
saurabhsarkar85@gmail.com

Abstract— ***Due to enormous growth of internet, demands for multiple access and request for services have also increased significantly. In a result there is a rise in packet loss rates and drop in network efficiency. In addition, the inability to support new services has severely hindered the widespread deployment of bandwidth-sensitive applications. Active Queue Management (AQM) algorithms have been designed to be able to actively control the average queue length in routers supporting TCP traffic, and thus to be able to prevent congestion and resulting packet loss as much as possible. This paper focuses on how congestion control and queue management techniques have evolved in due course of time and being modified to minimize the rate of packet loss. This paper specifically classifies the wide range of available AQM algorithms.***

Keywords— *Active Queue Management; TCP; RED; Drop Probability; BLUE; AVQ; GREEN.*

I. INTRODUCTION

In the last fifteen to twenty years, the use of the Internet has taken off rapidly. The public Internet is a world-wide computer network, which interconnects millions of computing devices around the world. Most of these devices are so-called hosts or end systems, e.g., traditional desktop PCs, Unix-based workstations or servers that store and transmit information such as web pages and email messages. Each of these end systems, as well as most of the other pieces forming the Internet, run so-called protocols that control the sending and the receiving of information within the Internet. Two of the most important of these protocols are the Transmission Control Protocol (TCP) and the Internet Protocol (IP).

End systems are connected by communication links, made up of different types of physical media, such as coaxial cable, copper wire, fibre optics and the radio spectrum. Usually, these end systems are not directly connected to each other via a single communication link, but are indirectly connected via intermediate switching devices, which are called routers or gateways. A router collects incoming information, determines the destination of this information, and forwards it again. The path travelled by the information from the source host to the destination host, via one or more routers, is called a route or path through the network.

Each of the routers in a network has a finite amount of buffer space, in which it can store information before forwarding it to the network. This storage space is necessary for prevention of information loss whenever data is coming in at a rate higher than the maximum processing rate of the router. Information that cannot be processed and forwarded directly is stored in the router buffer until the router processor is available. In order to prevent buffering delay at the router from getting too high, and also to keep the amount of information loss as low as possible, this buffer needs to be managed intelligently. Therefore, a variety of buffer management algorithms, which are commonly known as active queue management algorithms have been designed. This paper contains an overview and a classification of a wide range of active queue management algorithms.

II. THE GENERAL PURPOSE OF AQM ALGORITHMS

The TCP protocol, detects congestion only after a packet has been dropped from the queue (according to the drop tail-algorithm). However, it is clearly undesirable to have large queues that are full most of the time, since this will significantly increase the delays. Therefore and also keeping in mind the ever increasing speed of networks, it is ever more important to have a mechanism that keeps the overall throughput high, but at the same time keeps average queue size as low as possible. Note that in order to maximize the network throughput, queues should not necessarily be kept completely empty all the time, since this will result in under-utilization of the link, but in order to have a small queuing delay, the queue length should be kept sufficiently small.

In order to fulfil the above needs, a wide range of AQM algorithms have been proposed. The purpose of these algorithms is to provide a mechanism to detect network congestion early and to start dropping packets from the router queue before this congestion will affect the network throughput too much. The definition of too much depends on the Quality of Service (QoS) to be delivered by the network.

AQM algorithms have been designed for implementation at network routers, as opposed to implementation at end nodes, such as TCP sender or receiver entities. This choice is advocated by the fact that detection of congestion can be carried out most effectively in the router itself. A network router can reliably distinguish between propagation delay and persistent queuing delay. Only the router has a unified view of the queuing behaviour over time; decisions about the duration and magnitude of congestion to be allowed at the router are therefore best made by the router itself.

III. A CLASSIFICATION OF AQM ALGORITHMS

The AQM algorithms are classified according to the criteria on which the decision whether or not to drop packets from the queue (when the link suffers from congestion) is being made. Four different strategies in this can be identified:

- Average queue length-based queue management (QM)
- Packet loss & link utilization-based QM
- Class-based QM
- Control theory-based QM

Alternatively, algorithms can be classified as being either reactive or proactive:

- A reactive AQM algorithm focuses on congestion avoidance, i.e., active early detection of and reaction to congestion. Congestion can occur in this case, but it will be detected early. Decisions on actions to be taken are based on current congestion.
- A proactive AQM algorithm focuses on congestion prevention, i.e., intelligent and proactive dropping of packets, resulting in prevention of congestion from ever occurring and ensuring a higher degree of fairness between flows. Decisions on actions to be taken are based on expected congestion.

Figure 1 gives an overview of the classification structure and the algorithms that will be addressed in the following sections. This classification is a slightly adapted version of the one presented in [1].

A. Average queue length-based algorithms

This class of reactive AQM algorithms bases the decision on whether or not to drop packets from the queue on the observed average queue length. The algorithms in this class can be divided further based on whether or not the algorithm pays special attention to fairness as it comes to the distribution of available bandwidth over the active data flows.

1) Random Early Detection (RED)

RED is one of the first AQM algorithms ever developed, proposed in [2] in 1993. It has been widely used with TCP and has been recommended by IETF. The algorithm detects incipient congestion by computing the average queue size at the router. When this average queue size exceeds a certain preset threshold, the router drops or marks each arriving packet with a certain probability p_a, which is a linear function of the average queue size.

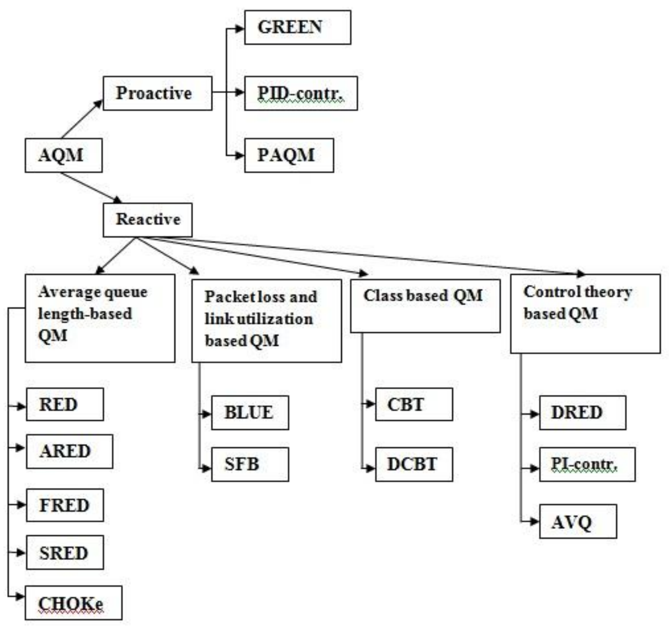

Figure 1 : Classification of AQM Algorithms

Connections are notified of congestion either by dropping packets arriving at the router, or rather by setting a bit in packet headers (which is referred to as 'marking a packet'). RED can be specified in pseudo-code as in *Figure 2*, where min_{th} and max_{th} are minimum and maximum thresholds, which can be set by the algorithm user, and which are both smaller than the maximum queue size allowed by the router.

```
for each packet arrival:
  calculate the average queue size $q_{avg}$
  if $min_{th} \le q_{avg} < max_{th}$
    calculate probability $p_a$
    with probability $p_a$:  mark/drop the arriving packet
  else if $max_{th} \le q_{avg}$
    mark/drop the arriving packet
  else
    do not mark/drop packet
```

Figure 2 : Pseudo-code of the RED algorithm

Whenever a packet arrival is detected by a router implementing RED, the average queue size q_{avg} is calculated using a low-pass filter to get rid of the potentially significant influence that short-term traffic bursts could have on the average queue size. Calculation of q_{avg} is carried out using

$$q_{avg} = (1 - w_q) q_{avg} + w_q q \qquad (1)$$

where q is the instantaneous queue length as observed at the router, and w_q is the weight applied by the low-pass filter to the 'old' average queue size. By increasing this value w_q, the influence of the current queue size on q_{avg} is also increased. This also increases the influence of traffic bursts, or periods of relatively little traffic on the average queue length. After q_{avg} has been calculated, it is compared to two threshold values, min_{th} and maxth. When q_{avg} is smaller than min_{th}, no packets are marked, and each arriving packet is appended at the end of the queue. When q_{avg} lies between min_{th} and max_{th}, each arriving packet is marked with probability p_a. When q_{avg} is greater than max_{th}, all incoming packets are marked with probability 1. The marking probability p_a is calculated as follows:

$$p_a = p_{\max} * (q_{avg} - \min_{th})/(\max_{th} - \min_{th}) \qquad (2)$$

In [2], p_a is increased even further as a function of the number of packets that have arrived since the last packet was marked, resulting in a new marking probability p_b. *Figure 3* shows a graphical representation of the possible values for the marking probability p_a.

Despite being widely used in combination with TCP for several years, RED has not found acceptance in the Internet research community. The reasons for this are several drawbacks of the algorithm, of which the most important are:

- The packet loss fraction is equal for all flows, regardless of the bandwidth used by each flow;
- No restriction exists against aggressive, non-adaptive flows, which has a negative impact on well-behaving, adaptive flows;
- It is difficult to parameterize RED queues to give good performance under different congestion scenarios;
- The equilibrium queue length strongly depends on the number of active TCP connections;
- Simulations have shown that RED does not provide clear advantages over the TCP drop tail-algorithm in realistic scenarios.

These drawbacks have been the main reason for the development of a vast collection of improved AQM algorithms.

2) *Average Rate Early Detection (ARED)*

ARED, proposed in [3], uses the average packet enqueue rate as a congestion indicator, and signals end hosts of incipient congestion with the objective of reducing the packet loss ratio and improving link utilization. Unlike most other AQM algorithms based on average queue length, ARED attempts to control the rate of queue occupancy change rather than controlling the queue occupancy itself. It intends to keep the queue length stabilized at an operational point at which the aggregate packet enqueue rate is approximately equal to or slightly below the link capacity. An incoming packet is dropped with a probability p, which is calculated according to the pseudo-code as seen in *Figure 4*. This pseudo-code is executed at every packet arrival event. From this pseudo-code, it can be derived that there are two situations in which packet loss might occur in ARED:

- The estimated packet enqueue rate R is higher than the link capacity μ (the congestion condition), or
- the instantaneous queue length is larger than a predetermined value L and the ratio of R to μ is larger than a predetermined value ρ (the congestion alert-condition).

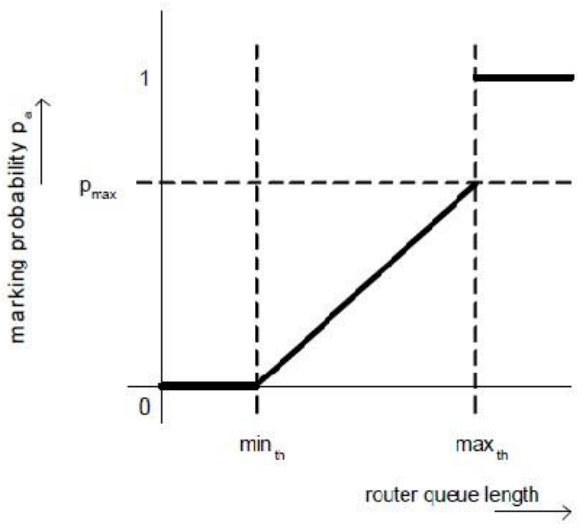

Figure 3 : The RED marking probability p_a

Here, L is the average queue length to which the ARED algorithm automatically converges, and ρ is usually set to 0.95. A packet is dropped when one of the two above conditions holds, and p is larger than a uniformly distributed random variable. In addition, the probability p is increased with an amount d_1 whenever either the congestion condition or the congestion alert-condition holds for a time span longer than a preset freeze time. When the link has been idle for a freeze time amount of time, p is decreased by an amount d_2, which is typically an order of magnitude smaller than d_1.

3) *Flow Random Early Drop (FRED)*

A problem with the algorithms described in RED and ARED is that they do not explicitly protect fragile, low-bandwidth flows from aggressive flows that tend to consume all of the available bandwidth. Aggressive flows typically have small RTTs, which allows them to increase their congestion window size faster after loss. This loss is caused by these aggressive flows with a reasonably high probability, and has a strongly negative influence on available bandwidth for fragile flows. FRED has been proposed in [4] as an alternative to RED in order to protect these fragile flows and thus to maintain a higher degree of fairness. It attempts to do so by using per-active-flow accounting, to impose on each flow a loss rate that depends on that flow’s buffer usage.

```
Upon every packet arrival:
    if ($link\_idle$)
        reset link idle timer;
        $link\_idle = false$;
    if ($R > \mu \parallel (q_{len} > L$ && $R > \rho \times \mu)$)
        drop packet with probability $p$;
        if ($current\_time - last\_update > freeze\_time$)
            $p = p + d_1$;
            $last\_update = current\_time$;

When link becomes idle:
    set link idle timer to $freeze\_time$;
    $link\_idle = true$;

When link idle timer expires:
    $p = p - d_2$;
    set link idle timer to $freeze\_time$;
```

Figure 4 : Pseudo-code of the ARED algorithm

FRED acts just like RED, but with the following additions: it introduces the parameters $min_q(i)$ and $max_q(i)$, which represent goals for the minimum and maximum number of packets that each flow i is allowed to buffer. FRED also has a global variable avg_{cq}, which is an estimate of the average per-flow buffer count; flows with less than avg_{cq} packets queued are favoured over flows with more packets in the queue. FRED also maintains a variable $q_{len}(i)$, a count of buffered packets for each flow that currently has any packets buffered. Finally, FRED maintains a variable strike(i) for each flow, which is a count of the number of times the flow has failed to respond to congestion notification; FRED penalizes flows with high strike(i) values.

Despite the additional fairness provisions taken by FRED, the algorithm has a known drawback concerning this fairness. FRED has a potential problem: its TCP-favoured per-flow punishment could unnecessarily discourage responsive UDP flows. Under FRED, incoming packets for a well-behaved TCP flow consuming more than their fair share are randomly dropped applying RED's drop rate. However, once a flow, although flow-controlled, is marked as a non-TCP friendly flow, it is regarded as an unresponsive flow and all incoming packets of the flow are dropped when it is using more than its fair bandwidth share. As a result, a responsive UDP flow, which may have a higher chance to be marked, will experience more packet loss than a TCP flow and will thus be forced to do with less than its fair share of bandwidth.

Another reasonably important drawback of the FRED algorithm is the fact that the peractive-flow accounting strategy requires a certain amount of overhead data to be stored for each active flow. This could result in significant additional processing delay at a router, especially when there are many flows active at the same time.

4) Stabilized RED (SRED)

The SRED algorithm has been presented in [5] as an attempt to stabilize the TCP router buffer occupation at a level independent of the number of active connections, by estimating the number of active connections or flows statistically. The goal of the algorithm is to identify flows that are taking more than their fair share of bandwidth, and to allocate a fair share of bandwidth to all flows, without requiring too many computations. For this purpose, it uses a finite zombie list, in which it stores information about recently active flows, together with a count variable and a time stamp, which is used for administrative purposes, for each flow in the zombie list. The list starts out empty, and whenever a packet arrives, its packet flow identifier (source address, destination address, etc.) is added to the list. Once the zombie list is full, every packet arriving at the router is compared to a randomly drawn flow from the list (the 'zombie'). After this comparison, one out of two possible actions will be taken:

- When the arriving packet's flow matches the zombie (a 'hit'), that zombie's count variable is increased by one, and the time stamp is set to the latest packet arrival time, or,
- when the arriving packet's flow does not match the zombie (a 'no hit'), with a preset probability p the flow of the new packet overwrites the zombie chosen for comparison. Its count variable is initialized to 0.

The number of active flows, which is used by SRED to stabilize buffer occupation, can then be estimated using the average 'hit' rate. How this is done is presented in [5]. The count variable is used in the identification of misbehaving flows: a flow with a high count value is more likely to be a misbehaving flow, and will therefore experience a higher loss probability.

5) CHOKe

The basic idea behind the CHOKe algorithm [6] is that the contents of a buffer form a sufficient statistic about the incoming traffic, and that these contents can be used in a simple fashion to penalize misbehaving flows. CHOKe attempts to provide a fair queueing policy by discriminating against these misbehaving flows. When a packet arrives at a congested router, CHOKe draws a packet at random from the buffer and compares it with the arriving packet. If they both belong to the same flow, then they are both dropped, else the randomly chosen packet is left intact and the arriving packet is admitted into the buffer with a probability p that depends on the level of congestion (where p is computed in exactly the same way as in RED). The reason for doing this is that the router buffer is assumed to be more likely to contain packets belonging to a misbehaving flow, and, hence, these packets are more likely to be chosen for comparison. Furthermore, packets belonging to a misbehaving flow are assumed to arrive more numerously and are more likely to trigger comparisons.

B. Packet loss & link utilization-based algorithms

The main idea behind this class of algorithms is to perform active queue management based on packet loss and link utilization rather than on the instantaneous or average queue lengths. Algorithms in this class maintain a single probability p, which is used to mark (or drop) packets when they are queued. If the queue is continually dropping packets due to buffer overflow, p is incremented, thus increasing the rate at which congestion notifications are sent back. Conversely, if the queue becomes empty or if the link is idle, p is decremented. Note that, in contrast to the class of AQM algorithms discussed in the previous section, no queue occupancy information is used.

1) BLUE

The BLUE algorithm [7] maintains a single probability, p_m, which it uses to mark (or drop) packets when they are enqueued. If the queue is continuously dropping packets due to buffer overflow, BLUE increments p_m, thus increasing the rate at which it sends back congestion notification. As has been said above, if the queue becomes empty or if the link is idle, p_m will be decremented. This allows BLUE to"learn" the correct rate at which it has to send back congestion notification. The algorithm can be specified in pseudo-code as in *Figure 5*.

```
Upon packet loss event:
    if $((now - last\_update) > freeze\_time)$
        $p_m = p_m + d_1$
        $last\_update = now$

Upon link idle event:
    if $((now - last\_update) > freeze\_time)$
        $p_m = p_m - d_2$
        $last\_update = now$
```

Figure 5 : Pseudo-code of the BLUE algorithm

The freeze time parameter should be a fixed variable, which is random for each router, in order to avoid global synchronization (the undesired phenomenon where each flow updates its window size at the same time). Note that the BLUE algorithm looks much like the ARED algorithm.

A slightly different variant of the BLUE algorithm, in which the marking probability is updated when the queue length exceeds a certain threshold value, has also been proposed. This modification allows room to be left in the queue for transient bursts, and allows the queue to control queuing delay when the size of the queue being used is large.

2) Stochastically Fair BLUE (SFB)

Since the original BLUE algorithm does not ensure fairness among flows, and more specifically does not provide protection for fragile flows against aggressive, non-responsive flows, SFB has been proposed in [7]. SFB is an algorithm that identifies and rate-limits non-responsive flows based on accounting mechanisms similar to those used with BLUE. It maintains N ×L accounting bins, which are organized in L levels containing N bins each. Additionally, SFB maintains L independent hash functions, each associated with one level of accounting bins. Each hash function maps a packet flow into one of the N accounting bins in that level. The accounting bins are used to keep track of queue occupancy statistics of packets belonging to a particular bin. Each bin in SFB keeps a marking (dropping) probability p_m, which is the same as in BLUE, and which is updated based on bin occupancy. When 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 reaches a certain threshold (i.e., the size of the bin), the marking probability p_m for that bin is increased. If the number of packets in a bin drops to zero, p_m for that bin is decreased. The pseudo-code specification of SFB is given in *Figure 6*.

```
define $B[l][n] = L \times N$ array of bins ($L$ levels, $N$ bins per level)

at packet arrival:
    calculate hash functions $h_0, h_1, \ldots, h_{L-1}$;
    update $q_{len}$ for each bin at each level;
    for $i = 0$ to $L-1$
        if $(B[i][h_i].q_{len} > bin\_size)$
            $B[i][h_i].p_m += delta$;
            drop packet;
        else if $(B[i][h_i].q_{len} == 0)$
            $B[i][h_i].p_m -= delta$;
    $p_{min} = \min(B[0][h_0].p_m, \ldots, B[L][h_L].p_m)$;
    if $(p_{min} == 1)$
        limit flow's sending rate;
    else
        mark / drop packet with probability $p_{min}$;
```

Figure 6 : Pseudo-code of the SFB algorithm

The main idea behind SFB 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. This bin thus has a normal p_m value. The decision to mark a packet is based on p_{min}, the minimum p_m value of all bins to which the flow is mapped into. If p_{min} is equal to 1, the packet is identified as belonging to a non-responsive flow, and the available bandwidth for that flow is then limited.

C. CLASS-BASED ALGORITHMS

For an algorithm belonging to this class, the treatment of an incoming packet (under congestion circumstances) depends on the class this packet belongs to. Theoretically, there are many possible class definitions, but in practice, it is most common to categorize incoming packets based on the transport protocol (i.e., TCP or UDP) that has been used to send the packet. With this type of algorithms, for every non-TCP class, a threshold is defined, setting a limit to the maximum amount of packets that a certain class can have in the queue.

1) Class-Based Threshold (CBT)

The CBT algorithm, proposed in [8], divides incoming packets into two classes: TCP packets and UDP packets. The goal of the algorithm is to reduce congestion in routers and to protect TCP flows from UDP flows, while also ensuring acceptable throughput and latency for well-behaved UDP flows. This is done by constraining the average number of non-TCP packets that may reside simultaneously in the queue. CBT attempts to realize a"better than best effort"-service for well-behaved multimedia flows that is comparable to that achieved by a packet or link scheduling discipline. However, CBT does this by active queue management rather than by scheduling. The algorithm is an attempt to construct an AQM scheme that will maintain the positive features of RED, limit the impact of unresponsive flows, but still allow UDP flows to access a configurable share of the link bandwidth. Moreover, it tries to do this without having to maintain per-flow state information in the router. Two different CBT variants can be identified:

CBT with RED for all: When an UDP packet arrives; the weighted average number of packets enqueued for the appropriate class is updated and compared against the threshold for the class, to decide whether to drop the packet before passing it to the RED algorithm. For the TCP class, CBT does not apply a threshold test, but directly passes incoming packets to the RED test unit.

CBT with RED for TCP: only TCP packets are subjected to RED's early drop test, and UDP packets that survive a threshold test are directly enqueued to the outbound queue. Another difference from CBT with RED for all is that RED's average queue size is calculated only using the number of enqueued TCP packets. CBT with RED for TCP is based on the assumption that tagged (multimedia) UDP flows as well as untagged (other) UDP flows are mostly unresponsive, and it is of no use to notify these traffic sources of congestion earlier.

2) Dynamic CBT (DCBT)

DCBT [9] is an extension of"CBT with RED for all". This algorithm fairly allocates the bandwidth of a congested link to the traffic classes by dynamically assigning the UDP thresholds such that the sum of the fair share of flows in each class is assigned to the class at any given time. The fair class shares are calculated based on the ratio between the numbers of active flows in each class. The key differences between CBT and DCBT are:

- Dynamically moving fairness thresholds, and
- the UDP class threshold test that actively monitors and responds to RED indicated congestion.

In addition to this, DCBT adopts a slightly different class definition compared to CBT: unlike the class categorization of CBT in which responsive multimedia flows are not distinguished from unresponsive multimedia flows (they are all tagged), DCBT classifies UDP flows into responsive multimedia (tagged) UDP flows and other (untagged) UDP flows. In order to calculate the per-class thresholds, DCBT needs to keep track of the number of active flows in each class.

D. Control theory-based algorithms

This category of algorithms has been developed using classical control theory techniques. The queue length at the router is regulated to agree with a desired value, by eliminating the "error", that is, the difference between the queue length and this desired value (see [10] for more details). A typical feedback control system for an AQM algorithm consists of

- a desired queue length at the router q_{ref} (i.e., a reference input),
- the queue length at a router as a system variable (i.e., a controlled variable),
- a system, which represents a combination of subsystems (such as TCP sources, routers and receivers),
- an AQM controller, which controls the packet arrival rate to the router queue by generating the packet drop probability as a control signal, and
- a feedback signal, which is a sampled system output used to obtain the control error term.

AQM algorithms based on control theory attempt to maintain the instantaneous queue length as close as possible to q_{ref}. The packet drop probability p is adjusted periodically, based on a combination of both the current queue length deviation (the error signal) and the sum of previous queue length deviations from q_{ref}. As a result of these periodic updates, an AQM control system can be modelled in a discrete-time fashion. *Figure 7* presents a graphical representation of a typical control theory-based AQM system. Apart from the typical control system parts mentioned above, this system also includes a low-pass filter and influences from external disturbances, such as noise, which have an impact on the controlled output.

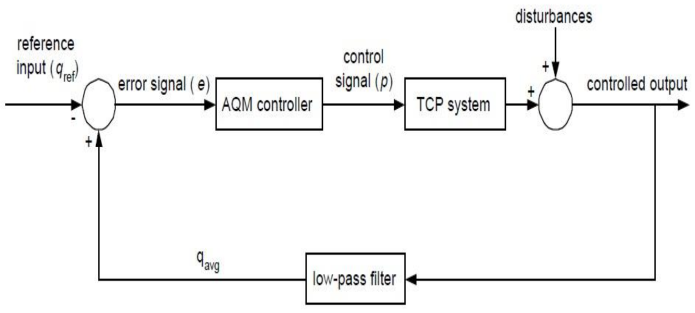

Figure 7 : TCP and AQM as a closed-loop control system

1) *PI-controller*

A linearized dynamic model of TCP, which has been presented in [11], has been chosen as the basis for the proportional-integral (PI-) controller algorithm [12]. PI-controller is basically proposed as a combination of two controller units, a proportional controller and an integral controller. In the proportional controller, the feedback signal is simply the regulated output (i.e., the queue length) multiplied by a proportional gain factor α:

$$p(n) = \alpha e(n) \tag{3}$$

where e(n) is the error signal at time n, i.e., the difference between the actual queue length at time n and the reference queue length q_{ref}. This error signal is usually normalized by the router buffer size B, since queue length fluctuations (and hence the error signal) grow linearly with buffer size. The α factor 'steers' the queue length to its desired length (the reference input). One of the drawbacks of the use of a purely proportional controller is the fact that its resulting steady state output will never be equal to the reference output, due to the multiplicative nature of this mechanism. Therefore, an integral controller has been added to the algorithm. The purpose of this integral controller is to remove the steady-state regulation error of an AQM algorithm, which is defined as the difference between the steady-state output and the desired reference value. The generic structure of a PI-controller is given by

$$p(n) = \alpha e(n) + \beta \int_0^n e(\tau) d\tau \tag{4}$$

where β is the integral gain constant. The proportional and the integral part of this equation can be identified easily. Note that when discrete time systems are being considered, β is often replaced by β / T_I, where T_I is the integral time. In [15], the structure of the PI-controller under attention has been put differently, namely as

$$p(n) = p(n-1) + \kappa e(n) + \lambda(e(n) - e(n-1)) \tag{5}$$

In [15], it is claimed that this is a representation of a PD-controller, although with some mathematical transformations, this equation can be rewritten as (4). The pseudo-code of the digital implementation of PI-controller, which should be functionally equal to (4), is given as

$$p(n) = p(n-1) + \alpha e(n) + \beta e(n-1) \tag{6}$$

When we suppose α = κ + λ, and β = λ, (5) and (6) are equivalent, implying (5) is equivalent to (4). The integral part of (5) is not as clearly visible as in (4), but it is present in the p(n − 1) element. The use of this element implies a summation over previous values of p(n), and implicitly over previous values of e(n), which corresponds to the integral term of (4). The fact that the system described in [13] is really a PI-controller in nature, instead of the PD-controller that it claims to be, is confirmed by the fact that this system is based on a PI-controller mentioned in [14]. The authors of [13] claim that the system from [14] is a simple P-controller, which is the reason for erroneously claiming their own system is a PD-controller.

A considerable difference between the systems from [12] and from [13] is the fact that the latter uses a low-pass filter in the calculation of the average queue length, whereas the former deliberately refrains from doing this. The first step in the application of the low-pass filter in [13] is the sampling of the current queue length q(n) from the system. This q(n) is then used to compute a filtered queue length $q_{avg}(n)$ using

$$q_{avg}(n) = (1-\chi) q_{avg}(n-1) + \chi q(n); \tag{7}$$

where χ is a preset filter gain, which affects the response speed of the queue length filter. The filtered queue length $q_{avg}(n)$ is then used in the calculation of the error signal e(n), which is given by

$$e(n) = q_{avg}(n) - q_{ref} \tag{8}$$

The intention of the application of a low-pass filter is to remove undesired effects of traffic bursts on the average queue length. The system described in [12] refrains from using the filter because:

"Although one of the design goals of the low-pass filter was to let transient bursts pass through, from a control standpoint the averaging can lead to instability and low frequency oscillations in the regulated output. In fact, the averaging mechanism is built into the queue dynamics, and the queue essentially acts like a low-pass filter."

2) *Dynamic RED (DRED)*

DRED [14] also uses simple control theory techniques to randomly discard packets with a load-dependent probability whenever a router becomes congested. The main goal of this algorithm, which is very similar to RED but uses a different (i.e. control theoretic) approach, is to stabilize the router buffer occupancy at a level independent of the number of active TCP connections. The operation of DRED is quite similar to the operation of a PI-controller (see the previous section), and can also be specified as a sequence of steps, carried out at time n. First, the current queue length q(n) is sampled. Then, the current error signal e(n) is computed as e(n) = q(n) − q_{ref}, which is very similar to (8), with the difference that no low-pass filter has been applied yet to the observed queue length, meaning q(n) represents the instantaneous queue length. To this error signal e(n), a low-pass filter is then applied using

$$\hat{e}(n) = (1-\chi)\hat{e}(n-1) + \chi e(n) \tag{9}$$

where χ is again the preset low-pass filter gain. The dropping probability p(n) can now be computed as

$$p(n) = \min\{\max[\,p(n-1) + \alpha\hat{e}(n), 0],\, p_{\max}\} \qquad (10)$$

where p_{max} is again the preset maximum dropping probability, and α is again the proportional gain. The dropping probability p(n) is then stored for use at time n+1, when a new probability p(n + 1) will be computed using the steps defined above. For this purpose, ˆe(n) needs to be stored as well. Simulations have shown that DRED is indeed able to stabilize the router queue length close to a predefined reference value, and accommodates well to traffic bursts without dropping too many incoming packets.

Note that when it comes to the application of the low-pass filter, DRED only differs from PI-controller in the moment at which the filter is applied. In DRED, it is applied to the error signal e(n), whereas in PI-controller it is applied to the average queue length q(n), which is then used to derive the error signal. Since the application of a low-pass filter is a linear operation, this does not make a difference when comparing DRED and PI-controller.

3) *Adaptive Virtual Queue (AVQ)*

The AVQ algorithm, proposed in [15], maintains a virtual queue, whose capacity (referred to as the virtual capacity) is less than the actual capacity of the router queue implementing AVQ. When a packet arrives at the router, the virtual queue is also updated to reflect a new arrival. Packets in the real queue are marked or dropped when the virtual buffer overflows. The virtual capacity at each link is then modified such that the total flow entering each link achieves a desired utilization of that link. Thus, instead of directly controlling the real queue length using a dropping probability, AVQ controls the virtual queue capacity, which implicitly imposes a dropping probability on packets in the real queue. No dropping probability is calculated directly.

Figure 8 shows an AVQ specification in pseudo-code, where VQ is the number of bytes currently in the virtual queue, b is the number of bytes of the arriving packet, B is the total (real) buffer size, and the last arrival variable is used to store the time of the most recent packet arrival. The ’update VQ’ event consists of updating the variable holding the current virtual queue length, since this one might have changed since the previous packet arrival event, e.g., because of packets being served that have left the queue. Note that this is different from updating the virtual capacity [15].

E. *Proactive algorithms*

One important drawback of the AQM algorithms presented in the previous sections is that their congestion detection and control functions depend only on either the current queue status, or the history of the queue status (e.g., the average queue length). Hence, the congestion detection and control in these algorithms are reactive to current or the past congestion, but do not respond proactively to incipient congestion. For example, the congestion detection method in RED can detect and respond to long-term traffic patterns using the exponentially weighted moving average queue lengths. However, it is unable to detect incipient congestion caused by short-term traffic load changes. In this case, the implicit congestion notification sent back to end hosts by a packet drop may be an inappropriate control signal, and can possibly make the congestion situation worse.

```
at each packet arrival do:
    update $V_Q$;
    if $(V_Q + b > B)$
        mark packet in real queue;
    else
        $V_Q \leftarrow V_Q + b$;
    endif
    update $virtual\_capacity$;
    update $last\_arrival$;
```

Figure 8 : Pseudo-code of the AVQ algorithm

This section describes a few proactive QM algorithms, which are designed to react on incipient congestion, rather than react on current congestion, as the reactive AQM algorithms presented in the previous sections do. Basically, this means that the packet drop probability at time t is not only a function of the current system parameters (thus, the parameter values at time t) and/or previous values (at time t − 1, t − 2, etc.), but also on (estimations of) future parameter values (e.g., at time t + 1).

1) *GREEN*

The GREEN (Generalized Random Early Evasion Network) algorithm [16] applies knowledge of the steady state behaviour of TCP connections to intelligently and proactively drop packets, and thus attempts to prevent congestion from ever occurring and to ensure a higher degree of fairness between flows. The algorithm operates as follows. The bandwidth BW of a TCP connection can be approximated by

$$BW = \frac{(MSS \times c)}{(RTT \times \sqrt{p})} \qquad (11)$$

where p is the packet loss probability and c is a constant depending on the acknowledgement strategy being used, as well as on whether packets are assumed to be lost periodically or randomly. Now consider a scenario with N active flows on a link of capacity L. The fair share throughput of each flow is then L/N. The loss probability p can now be defined as

$$p = \left(\frac{N \times MSS \times c}{L \times RTT}\right)^2 \qquad (12)$$

When this value is used as the dropping probability for congestion notification, GREEN forces flows to send at their fair-share rate. Since p depends on the number of flows and the RTT of each flow, congestion notification is more aggressive for large N and small RTT. By including the RTT as an inverse parameter in (12), GREEN also eliminates the bias of favouring TCP connections with smaller RTT with respect to throughput (flows with smaller RTT can increase their window size faster due to this smaller RTT and are therefore more aggressive. Hence, these flows grab more than their fair share of bandwidth, which leads to this bias). GREEN does not require any information about the congestion window size, which is usually hard to calculate and varies with the topology of the network. No end-to-end modifications are required either. The implementation of GREEN relies on the knowledge of flow RTTs and the total number of active flows N. In [16], it is assumed that routers have knowledge of these parameters, but at the end of the paper, some hints on how the GREEN algorithm can estimate these parameters itself (thereby removing the need for routers to have knowledge about these parameters) are briefly presented.

2) PID-controller and PAQM

The PID (proportional-integral-derivative) controller algorithm [10] is constructed by combining a PI-controller unit with a PD-controller unit. It thereby combines the advantages of the PI-controller (removal of the steady-state error, at the cost of increased response time) with the most important advantage of a PD-controller unit, which is the improving of damping and of speed of response. The design of PID-controller is based on a control theoretic plant model of TCP.

A generic PID control equation, which, in combination with the plant model of TCP, serves as the basis for the PID-controller algorithm, can be given as

$$p(n) = \alpha e(n) + \beta \int_0^n e(\tau)d\tau + \gamma \frac{de(n)}{dn} \qquad (13)$$

The proportional, integral and derivative parts of the equation can be clearly identified here. A variant of PID-controller, the Pro-Active Queue Management or PAQM algorithm, does not rely on assumptions on the plant dynamic model, i.e. on the TCP flow dynamic model. Instead, it has been designed using the direct digital design method, based on a discrete representation of the PID control equation in (13) and on certain properties of network traffic. A detailed specification of the PAQM algorithm can also be found in [10]. Simulations presented in [10] show that both algorithms outperform reactive AQM algorithms such as RED in terms of queue length dynamics, packet loss rates, and link utilization, which in turn justifies the fact that a proactive queue management approach has some considerable benefits over ”traditional” reactive queue management algorithms.

IV. CONCLUSIONS

There has been a need for structuring the wide variety of AQM algorithms that have been developed since the introduction of RED in 1993. Organizing the existing algorithms into a decent structure makes it easier to see which ones are more or less alike, and therefore provides a strong basis for the comparison of different algorithms. A slightly adapted version of the classification first presented in [1] has proven to be a useful tool in this structuring. When categorizing an algorithm according to the criteria on which the decision whether or not to drop an incoming packet is being made in that algorithm, a well-structured classification tree can be set up, as can be seen in Figure 1.

REFERENCES

[1] A.J. Warrier, ”Active queue management: A survey”, North Carolina State University, December 2003.
[2] S. Floyd and V. Jacobson, ”Random Early Detection gateways for congestion avoidance”, IEEE/ACM Transactions on Networking, vol. 1 No. 4, pp. 397-413, August 1993.
[3] L. Su and J.C. Hou, ”An active queue management scheme for Internet congestion control and its application to differentiated services”, The Ohio State University, Columbus, OH, January 2000.
[4] D. Lin and R. Morris, ”Dynamics of Random Early Detection”, Computer Communication Review, Proceedings of ACM SIGCOMM 1997, vol.27 No.4, pp. 127-137, 1997.
[5] T.J. Ott, T.V. Lakshman and L. Wong, ”SRED: Stabilized RED”, Proceedings of IEEE INFOCOM 1999, pp. 1346-1355, March 1999.
[6] R. Pan, B. Prabhakar and K. Psounis, ”CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation”, Proceedings of IEEE INFOCOM 2000, pp. 942-951, April 2000.
[7] W.C. Feng, D.D. Kandlur, D. Saha and K.G. Shin, ”BLUE: A new class of active queue management algorithms”, Tech. report CSE-TR-387-99, University of Michigan, April 1999.
[8] M. Parris, K. Jeffay and F.D. Smith, ”Lightweight active router-queue management for multimedia networking”, Multimedia Computing and Networking, SPIE Proceedings Series, vol. 3020, January 1999.
[9] M. Claypool and J. Chung, ”Dynamic CBT and ChIPS Router support for improved multimedia performance on the Internet”, Proceedings of ACM Network and Operating System Support for Digital Audio and Video (NOSSDAV), June 2000.
[10] S. Ryu, C. Rump and C. Qiao, ”Advances in active queue management (AQM) based TCP congestion control”, Telecommunication Systems, vol.25 No.4, pages 317-351, March 2004.
[11] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, ”A control-theoretic analysis of RED”, Proceedings of IEEE INFOCOM 2001, pp. 1510-1519, April 2001.
[12] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, ”On designing improved controllers for AQM routers supporting TCP flows”, Proceedings of IEEE INFOCOM 2001, pp. 1726-1734, April 2001.

[13] J. Sun, G. Chen, K.T. Ko, S. Chan and M. Zukerman, "PD-Controller: A new active queue management scheme", IEEE Globecom 2003.

[14] J. Aweya, M. Ouelette and D.Y. Montuno, "A control theoretic approach to active queue management", Computer Networks, vol.36, pp. 203-235, 2001.

[15] S. Kunniyur and R. Srikant, "Analysis and design of an Adaptive Virtual Queue (AVQ) algorithm for Active Queue Management", Proceedings of ACM SIGCOMM, pp. 123-134, August 2001.

[16] W. Feng, A. Kapadia and S. Thulasidasan, "GREEN: Proactive queue management over a best-effort network", IEEE GLOBECOM, November 2002.