CDRED: ACTIVE QUEUE MANAGEMENT FOR FAIRNESS

CDRED: Active Queue Management for Fairness

By
Shahram Siavash, B.Sc./Software

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Applied Science (M.A.Sc.)

McMaster University

© Copyright by Shahram Siavash, January 2005

MASTER OF APPLIED SCIENCE (2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: CDRED: Active Queue Management for Fairness

AUTHOR: Shahram Siavash, B.Sc./Software (Sharif University)

SUPERVISOR: Dr. D.G. Down

NUMBER OF PAGES: x, 92

Abstract

Active Queue Management (AQM) algorithms, which are used in intermediate routers, try to prevent network congestion by randomly dropping packets. One problem with most AQM schemes is that they cannot enforce fairness. In other words, they do not have the ability to control how to divide the available bandwidth amongst flows. This may cause an unresponsive or aggressive flow to monopolize the bandwidth usage.

In this thesis, we have studied current AQM schemes and their approach in congestion management. We improved the existing DRED algorithm and used it in our own AQM algorithm called Class-based DRED (CDRED). CDRED is able to protect responsive flows from unresponsive ones. Moreover, it has the ability to divide the available bandwidth based on weights that the user supplies.

Acknowledgements

I would like to thank my supervisor, Dr. D.G. Down, for his time, guidance, and support he gave me during this thesis. Without his help and constructive suggestions, this thesis would have not been finished.

Contents

List of Figures

List of Tables

Chapter 1

Congestion Control and Active Queue Management

1.1 Introduction

Since the Internet was in its early stages, protocol designers have been thinking about the problem of congestion and how to deal with it. The Transmission Control Protocol (TCP) is a good example of that. TCP provides different methods to control the flow between two hosts and how to slow them down if any congestion occurs. As a packet starts its journey from its source to its destination, it passes through different routers. These routers each have a buffer with a fixed capacity to store the arriving packets before dispatching them. The arriving packets form a queue in this buffer. What we mean by congestion is the situation where one or more routers along the path hold packets in their queue close to their buffer capacity. In this case there is a high probability that arriving packets will be discarded simply because there is no room left in the buffer. The rate control that TCP provides is at flow level (only the transmitters/receivers are involved). In other words, TCP looks at routers and network resources as passive objects that cannot play a role in congestion control.

As the Internet expanded, the methods used by these protocols were no longer enough. The reasons why TCP can no longer handle congestion by itself will be discussed in the next section. Because of these problems, new means of congestion control were suggested. In these methods, collectively called Active Queue Management (AQM), the intermediate routers play an active role in controlling congestion. Their main goal is to slow down sources before any congestion happens. This seems like a good idea as sources are not aware of the queue condition inside routers. So by preventing congestion, the number of lost packets will be smaller, which will be beneficial for both hosts and network resources, simply because for each lost packet TCP retransmits the packet, which takes time and also uses network bandwidth.

1.2 TCP Flow Control

In this section, we are going to describe briefly how TCP detects and deals with congestion and how it controls the flow between two hosts. For more information about this topic, the reader is referred to [1] and [2].

TCP is one of the most popular protocols in the Internet for reliable packet delivery. In order to make sure that every byte reaches its destination, TCP assigns a sequence number for each byte and expects the receiver to acknowledge the receipt of the byte with that sequence number. Amongst the fields in the TCP header, there are two fields for holding the sequence number and the acknowledgement number. The acknowledgement number is cumulative. So for example if the receiver sends a packet with acknowledgement number 10, that means it has received from bytes 1 to 9 and is expecting to receive the 10th byte.[1]

Data supplied to the TCP protocol for transmission will be divided into pieces called segments. Each segment includes parts of the user data along with the TCP header. TCP will decide on the segment size after establishing

[1] Assuming that the sequence number started from 1

a connection with the receiver. During early phases of establishing the connection, the receiver can send (as an option in the TCP header) the size of the largest segment which it is willing to accept. Based on that and other factors such as the maximum transmission unit of the network, the sender will decide on its maximum segment size (MSS).

In order to use network bandwidth efficiently, instead of sending one packet and then waiting for its acknowledgment, TCP sends groups of packets, called the *window*. The sender then stops transmitting and hopes that all of the packets in the window arrive at the destination and expects the receiver to acknowledge the reception of the last packet (actually the last byte of the last packet). The window size is not constant. The basic idea behind TCP is that the window size (and thus the transmission rate) can increase if there is no congestion and should decrease if there is congestion. As a result, when the sender starts transmitting, its window size will be one segment and based on network congestion it will be changed during the transmission . TCP gives an algorithm for the rate at which to increase and decrease the window size. The value for the window size is based on two other variables: the receiver's advertised window and the congestion window. These values are explained later in this section.

The optimum value for the window size is the product of the average bandwidth along the path and the round trip time. In this case the channel between two hosts will have 100% utilization and there will not be a packet queue. TCP tries to estimate the average round trip time by using its timers and keeping track of the time it took for previous segments to be acknowledged (RTT of previous segments). This estimate is updated constantly.

TCP detects congestion along the path when a segment is lost. If during a specified time, the source does not get an acknowledgment for a segment, it will assume that there is congestion in the network that has caused either the segment itself or its acknowledgment to get discarded.

In recent years, a new approach was suggested to inform sources about network congestion. Instead of dropping a packet, routers can set a bit in

the packet header called the Explicit Congestion Notification (ECN) bit and forward the packet. An ECN-capable receiver will report this bit back to the source so it can slow down. The advantage of this approach is that the packet will not be dropped and it will reach its destination while at the same time the source will be notified about the congestion. For more information about ECN, the reader is referred to [3]. Most AQM schemes can be configured to either set the ECN bit or drop the packet.

1.2.1 Congestion Control

TCP has different ways in controlling the flow between two hosts:

1. **Receiver's advertised window**
   The Receiver can set up a maximum number of bytes that it can accept at any time by advertising a window size in the TCP header. This window size represents local buffer space available at the receiver for that specific connection. The application receiving those bytes should remove them from the TCP buffer, so for example if the application goes to sleep, the buffer will get filled as the bytes arrive and sooner or later there will be no space left in the buffer. In that case the receiver will advertise a window of size zero, which will make the sender stop sending any more data.

2. **Congestion window**
   When TCP detects congestion in the network (when its retransmission timer expires), by retransmitting the same number of bytes over and over, the congestion in the network will get more severe and the receiver may not get the retransmitted bytes again. If sources do not slow down during congestion, the network may reach a situation which is called congestion collapse.

   While the receiver's advertised window is a receiver-side limit on the amount of data that the sender can send, the congestion window (*cwnd*)

is a sender-side limit. At any time, the amount of data that TCP can send beyond the last acknowledged byte is the minimum of the receiver's advertised window and the congestion window.

Based on the Request For Comments (RFC) 2581 [4], "the initial value for congestion window MUST be less than or equal to 2*SMSS bytes (SMSS: Sender's MSS) and MUST NOT be more than 2 segments." Usually in implementations, the initial value for the congestion window will be set to one full segment size (MSS).

TCP also holds another local variable, *ssthresh*, which stands for slow start threshold. The value for this variable and its relation with the congestion window determines how TCP should increase the congestion window.

- **Slow start phase** This is when *cwnd* < *ssthresh*. During slow start, TCP increments *cwnd* by at most MSS bytes for each acknowledgement (ACK) received that acknowledges new data. So *cwnd* will be set to 1 and TCP will send one segment and wait for its acknowledgment. After receiving its ACK, *cwnd* will be 2 and TCP will send two segments in its window. By receiving the acknowledgment for those segments, *cwnd* will get 4, then 8, 16, 32, etc. Even though it is called the slow start phase, the *cwnd* will grow exponentially during this time.
- **Congestion avoidance phase** In this phase, we have *cwnd* >= *ssthresh*. This is when *cwnd* finally reaches *ssthresh* during the slow start phase or there has been a segment loss. "During congestion avoidance, *cwnd* is incremented by 1 full-sized segment per round-trip time (RTT)" [4]. In order to implement this, TCP will try to increase the congestion window by MSS bytes when all data in the current window have been acknowledged. Instead of waiting for the acknowledgment of the whole window of data, TCP increases *cwnd* by a fraction for each ACK it receives. The following formula

is usually used to update *cwnd* during congestion avoidance

$$cwnd_{new} = cwnd_{old} + (MSS * MSS)/cwnd_{old} \qquad (1.1)$$

For more info on how formula 1.1 achieves what is expected by the RFC, the reader is referred to [1].

When TCP detects a segment loss through its retransmission timer, it will set *cwnd* to one and *ssthresh* will be updated to

$$ssthresh = max(current\ window\ size/2, 2 * MSS). \qquad (1.2)$$

The above formula ensures that *ssthresh* will not get a value less than twice the size of MSS. Based on the new values for *cwnd* and *ssthresh*, TCP will enter its slow start phase.

1.2.2 Fast Retransmit and Fast Recovery

When the receiver gets an out of order segment, it should immediately send a duplicate ACK. The receiver will buffer the out of order segment and hope that the missed segment will finally arrive to fill the gap. In TCP with the Fast Retransmit feature, after receiving 3 duplicate ACKs (that is 4 ACKs in a row for the same segment), the sender will no longer wait for its retransmission timer to expire for that specific segment. It will assume that the segment is lost and will retransmit only the lost segment. After this, the Fast Recovery algorithm will take over until a new ACK arrives. The Fast Recovery algorithm will set *ssthresh* based on (1.2). The congestion window will get *ssthresh* + $3 * MSS$. The term $3 * MSS$ is to account for those three segments that have left the network and have been received by the receiver, even though out of order. By receiving each duplicate ACK after that, TCP will increase the congestion window by MSS. TCP can also transmit a segment if allowed by the new value of *cwnd* and the receiver's advertised window. After receiving a new ACK, TCP will set *cwnd* to *ssthresh* (the value resulting from (1.2)).

For more information on how TCP reacts during Fast Retransmit, the reader is referred to [4].

What is different in Fast Retransmit is that TCP will not switch to the slow start phase when receiving 3 duplicate ACKs. That is because receiving those ACKs shows that the receiver has got 3 segments, but that they are simply out of order. Since those segments have already left the network, there is less congestion.

The next section discusses AQM algorithms and how they control and manage congestion.

1.3 Active Queue Management

In this section we will briefly introduce some of the current AQM algorithms suggested in different articles to be used in routers. For each of these schemes, the notation used is the same as that used by their developers.

1.3.1 Drop Tail

This is actually the simplest way a router can act during network congestion. In Drop Tail, as soon as the router's internal buffer is full, it will drop all of the incoming packets until there is again some space available in the buffer. This is actually how routers behave right now when there is no AQM involved.

The advantage of this method is its ease of implementation but there are a couple of disadvantages.

The first problem is that it can cause *global synchronization*. That is when the router drops packets from all sources when its buffer becomes full. This in turn causes all sources to slow down and thus decrease their window size. After sometime, all of them will speed up almost simultaneously. In this way the queue size will oscillate. In the worst case the queue may oscillate from empty to full and empty again. It is obvious that in this case, bandwidth will not be utilized efficiently.

The second problem with drop tail is that it will notify the sources too late about congestion, especially in networks with large round trip times. A source will send additional packets before it finds out about network congestion. One purpose of AQM is to let sources slow down earlier than the time that there is actually real congestion in the network.

The other problem is that in networks with heavy load when the queue size is close to its maximum. In this case, the router cannot handle burst traffic which may be caused from one or more sources transmitting simultaneously.

There have been some suggested alternatives to drop tail such as *Random Drop*[5] and *Early Random Drop*. In Random Drop, instead of dropping the last packet all of the time, when the buffer gets full, the router will randomly drop one of the packets in the queue, including the one just arrived. This helps in avoiding global synchronization. In this case the drop probability for each flow to get its packet discarded depends on the share that flow is using from the queue at that specific time.

"In the implementation of Early Random Drop gateways in [6], if the queue length exceeds a certain *drop level*, then the gateway drops each packet arriving at the gateway with a fixed *drop probability*"[7].

"For the version of Early Random Drop gateways used in the simulations in [8], if the queue is more than half full then the gateway drops each arriving packet with probability 0.02. Zhang [8] shows that this version of Early Random Drop gateways was not successful in controlling misbehaving users. In these simulations, with both Random Drop and Early Random Drop gateways, the misbehaving users received roughly 75% higher throughput than the users implementing standard 4.3 BSD TCP"[7]. That is because by dropping packets from responsive flows like TCP, those sources will slow down but unresponsive flows, such as those using the User Datagram Protocol (UDP), will keep sending with their previous rates.

1.3.2 Random Early Detection (RED)[7]

In this algorithm, the router detects incipient congestion by computing the average queue size. Its goal is to avoid congestion by controlling the queue size. Additional goals are avoiding bias against burst traffic and avoiding global synchronization by randomly dropping packets. Keeping the average queue size low will also help dealing with burst traffic. In order to prevent unresponsive sources filling the queue, the algorithm will drop every arriving packet when the average queue size exceeds some maximum threshold.

RED calculates weighted average queue size. The weight is given as a parameter, w_q. Other parameters are max_{th} for maximum threshold, min_{th} for minimum threshold and max_p for maximum drop probability. Basically RED calculates average queue size, avg, on each packet arrival and compares it with given thresholds. Based on that:

If $avg < min_{th}$

- Enqueue the arriving packet

else If $min_{th} \leq avg < max_{th}$

- Calculate the mark probability p_a
- Mark the arriving packet with probability p_a

else // $avg \geq max_{th}$

- Mark the arriving packet

The detailed algorithm for RED is given in Figure 1.1. As we can see, there is another variable involved in the algorithm, *count*, which is initialized to -1. This variable holds the number of unmarked packets when $min_{th} \leq avg < math_{th}$ and when the algorithm marks a packet, its value is set to zero. Let X be the number of packets that arrive, after a marked packet, until the next packet is marked. It is shown in [7] that by calculating the final drop

probability as $p_a = p_b/(1 - count \cdot p_b)$. X will be a uniform random variable on $\{1, 2, ..., 1/p_b\}$ (assuming for simplicity that $1/p_b$ is an integer). Figure 1.2 shows the diagram for p_b.

In calculating the average queue size, the algorithm also considers the time the queue was empty. It estimates the number of small packets, m, that could have been transmitted during the router idle time and then based on that, updates avg.

There is an option in RED to consider queue size in bytes rather than number of packets. In this case, a large FTP packet is more likely to be marked than a small TELNET packet. In order to achieve this, the algorithm should slightly change in the following manner:

$$
\begin{aligned}
p_b &\leftarrow max_p(avg - min_{th})/(max_{th} - min_{th}) \\
p_b &\leftarrow p_b \cdot PacketSize/MaximumPacketSize \\
p_a &\leftarrow p_b/(1 - count \cdot p_b)
\end{aligned}
$$

Figure 1.3 shows simulation results for the network shown in Figure 1.4. The parameters are given in Table 1.1. As can be seen in Figure 1.4, the average

Parameter	Value
min_{th}	5 packets
max_{th}	15 packets
max_p	1/50
w_q	0.002

Table 1.1: RED parameters used in simulation

queue size is between the maximum and minimum thresholds, but the actual queue size exceeds the maximum threshold during peak times.

```
Initialization:
    avg ← 0
    count ← −1
for each packet arrival
    calculate the new average queue size avg:
        if the queue is nonempty
            avg ← (1 − w_q)avg + w_q q
        else
            m ← f(time − q_time)
            avg ← (1 − w_q)^m avg
    if min_th ≤ avg < max_th
        increment count
        calculate probability p_a:
            p_b ← max_p(avg − min_th)/(max_th − min_th)
            p_a ← p_b/(1 − count · p_b)
        with probability p_a:
            mark the arriving packet
            count ← 0
    else if max_th ≤ avg
        mark the arriving packet
        count ← 0
    else count ← −1
when queue becomes empty
    q_time ← time
```

Saved Variables:

- avg: average queue size
- q_time: start of the queue idle time
- $count$: packets since last marked packet

Fixed parameters:

- w_q: queue weight
- min_{th}: minimum threshold for queue
- max_{th}: maximum threshold for queue
- max_p: maximum value for p_b

Other:

- p_a: current packet-marking probability
- q: current queue size
- $time$: current time
- $f(t)$: a linear function of the time t

Figure 1.1: RED Algorithm in detail

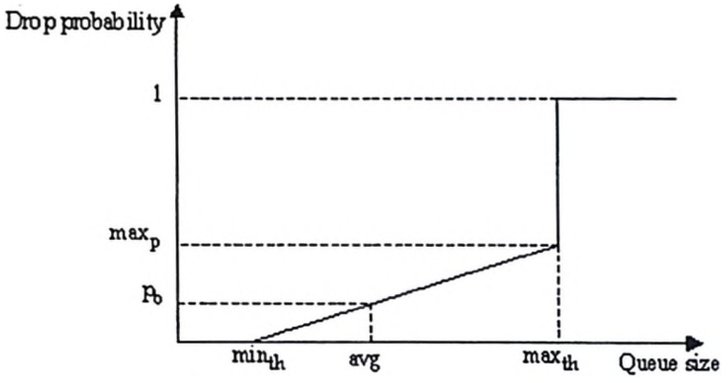


Figure 1.2: The diagram for p_b vs average queue size

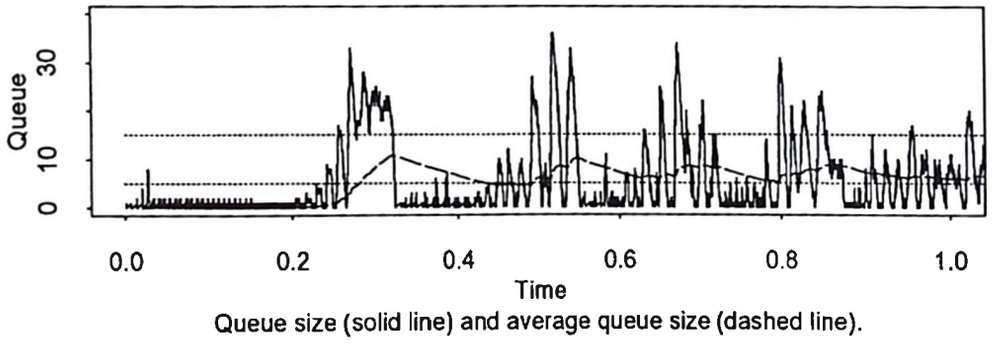


Figure 1.3: RED Algorithm: Queue size and average queue size vs. Time

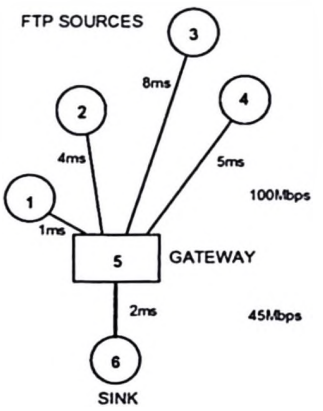


Figure 1.4: RED Simulation Network

1.3.3 BLUE [9]

Unlike RED, which acts based on average queue size, BLUE uses packet loss and link idle events to manage congestion. When the link becomes idle, it decrements the marking probability, which in turn will make sources more aggressive. On the other hand, when the queue overflows or passes a certain limit, the marking probability will be increased.

In order to let the sources adjust their sending rates based on the new value of marking probability, BLUE makes sure that a certain amount of time, *freeze_time*, has passed between two consecutive updates. There are two more parameters used in the algorithm: $d1$ and $d2$. They determine the amount to increment or decrement the marking probability on the events of packet loss or link idleness, respectively. The algorithm for BLUE is given in Figure 1.5.

The idea to have two different rates to increase or decrease the marking probability sounds reasonable, since one gets the ability to penalize or reward sources with different rates. For the experiments in [9], $d1$ is set significantly larger than $d2$. The logic behind this decision is claimed to be that an under-utilized link is caused by either an overly conservative algorithm (one that

Upon packet loss (or $Q_{len} > L$) event:

if ((now - *last_update*) > *freeze_time*) then

$p_m = p_m + d1$

last_update = now

Upon link idle event:

if ((now - *last_update*) > *freeze_time*) then

$p_m = p_m - d2$

last_update = now

Figure 1.5: BLUE Algorithm

tends to enqueue most packets) or an overly aggressive algorithm (one that tends to mark most packets) while queue overflow is only because of an overly conservative algorithm. So by weighting against packet loss, BLUE can manage to keep the queue size below its capacity even under heavy loads.

One challenge for most AQM schemes is how to be fair amongst all flows. Usually when there are one or more unresponsive flows, the algorithm will let the unresponsive flows take the major share of the bandwidth. To address this problem, Stochastic Fair BLUE (SFB) has been introduced in [9].

The idea behind SFB is to use different independent hash functions to map each packet passing through the router. A two dimensional array is kept where each element is actually a marking probability. In this way, SFB tries to detect unresponsive flows and use a different marking probability for them. For more information on SFB, the reader is referred to [9].

1.3.4 Random Exponential Marking (REM) [10]

REM tries to match the aggregate input rate to the link capacity and stabilize the queue around a small target, regardless of the number of sources competing for the bandwidth. The basic idea behind REM is to decouple congestion

measurements from performance measurements such as packet loss or queue size. It is argued in [10] that one can have good performance measures, like low packet loss or low delay, while the network is congested. While RED uses average queue size as a measurement for congestion, REM calculates its own expression called *price*. The price is defined for each outgoing queue and can be updated either periodically or asynchronously. Its value will be used to calculate the marking probability. The price at time $t+1$ for queue l, $p_l(t+1)$, is calculated via the recursion

$$p_l(t+1) = \Big[p_l(t) + \gamma\Big(\alpha_l(b_l(t) - b_l^*) + x_l(t) - c_l(t)\Big)\Big]^+ \qquad (1.3)$$

where

$$\begin{aligned} z^+ &= max\{z, 0\} \\ \gamma > 0 &: \text{user-defined constant} \\ \alpha_l > 0 &: \text{user-defined constant} \\ b_l(t) &: \text{queue size in period } t \\ b_l^* > 0 &: \text{target queue size} \\ x_l(t) &: \text{aggregate input rate in period } t \\ c_l(t) &: \text{available bandwidth to queue } l \text{ in period } t \end{aligned}$$

As we can see in (1.3), REM calculates how far the queue size is from the desired queue size and measures the difference between the aggregate input rate and the available capacity. The constant α_l determines the trade off between link utilization and queueing delay while γ controls how fast the algorithm reacts to network changes.

Since in practice it is easier to measure the queue size than the aggregate input rate, one can replace $x_l(t) - c_l(t)$ by $b_l(t+1) - b_l(t)$. Thus, (1.3) becomes

$$p_l(t+1) = \Big[p_l(t) + \gamma\Big(b_l(t+1) - (1-\alpha)b_l(t) - \alpha_l b_l^*\Big)\Big]^+ . \qquad (1.4)$$

Based on the link price, the marking probability $m_l(t)$ is calculated as follows:

$$m_l(t) = 1 - \phi^{-p_l(t)} \qquad (1.5)$$

where $\phi > 1$. Using (1.5), the probability of a packet being marked along a path is:

$$1 - \prod_{l=1}^{L}(1 - m_l(t)) = 1 - \phi^{-\sum_l p_l(t)}. \tag{1.6}$$

From (1.6), the end-to-end marking probability of a packet depends on the sum of the link prices it is passing through.

1.3.5 GREEN[11]

Like REM, GREEN tries to estimate the packet arrival rate and correspondingly adjust its congestion notification. However, all that is important to GREEN is whether or not the estimated arrival rate is larger than the target link capacity. In other words, GREEN does not consider how far the estimated arrival rate is from the target link capacity. If the estimated arrival rate is larger than the target link capacity, the marking probability will be increased by ΔP every ΔT seconds, otherwise, it will be decreased by the same amount at the same rate. GREEN calculates its marking probability in the following manner:

$$P = P + \Delta P \cdot U(x_{est} - c_t)$$

where

$$U(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases},$$

and x_{est} is the estimated arrival rate which is calculated based on exponential averaging, (given explicitly in (1.7)), and c_t is the target link capacity (typically 97% of total capacity). The estimated arrival rate, x_{est}, is updated according to (5) in [11]:

$$x_{est} = (1 - exp(-Del/K)) \cdot (B/Del) + exp(-Del/K) \cdot x_{est}, \tag{1.7}$$

where Del is the inter-packet delay, B is the packet size, and K is a time constant. As shown in [11] and [12], x_{est} converges to the actual arrival rate under a wide range of conditions.

It is discussed in [11] that by considering the difference in arrival rate and link capacity, in some cases REM misinterprets the decrease in rate. For example, when the queue overflows, the sources will slow down, but REM assumes that the demand for the link has dropped so it decreases the price. But since GREEN uses a step function, it will not make a big change to its marking probability for this temporary situation.

1.3.6 Class-Based Thresholds (CBT) [13]

So far, all of the AQM schemes introduced assume that the flows passing through the router are responsive. A source of a responsive flow will reduce its rate upon receiving a marked packet or detecting a packet loss, while unresponsive or misbehaving flows are those that do not act exactly as TCP in the case of congestion. Examples of misbehaving flows are non-standard implementations of TCP, a UDP source that does not respond to congestion notification, or a UDP source that responds in a different way to network congestion.

Some applications, like real time or multimedia applications, choose unresponsive flows, since they are more concerned about the network throughput and delay, rather than its reliability. In the presence of one or more unresponsive flows, the AQM algorithm should protect the responsive flows, otherwise there is a possibility that the high bandwidth unresponsive flows monopolize the queue and take most of the bandwidth capacity. Figure 1.6 shows the aggregate TCP throughput under RED in an experiment done in [13], where a high bandwidth UDP flow is introduced between time 60 and 110. As shown in Figure 1.6, the aggregate TCP throughput drops significantly during that time.

CBT puts flows in two general groups: responsive flows and unresponsive flows, such as UDP. Later it divides the unresponsive flows into continuous media flows, and all other flows. The packets from continuous media flows are tagged either by the end-systems or by network administrators.

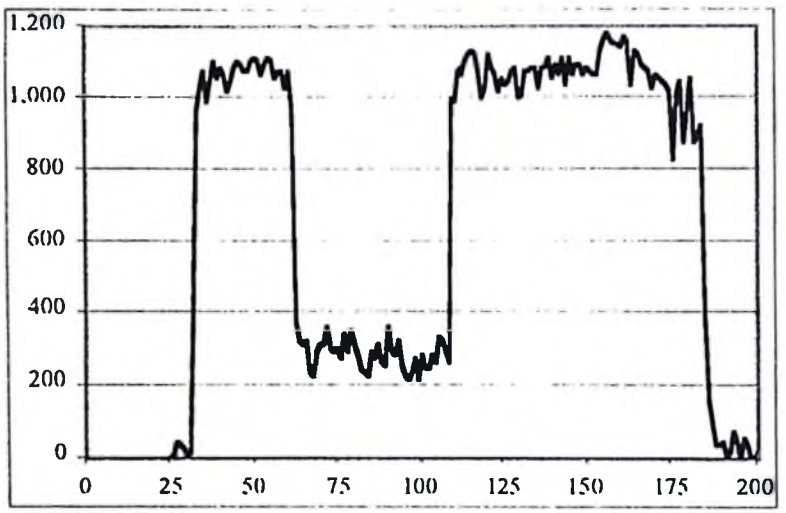


Figure 1.6: The aggregate TCP throughput under RED in kilobytes/second versus elapsed time in seconds

The main goal of CBT is to protect responsive flows from unresponsive ones. It is not trying to provide a fair share of bandwidth amongst all flows. It achieves its goal by using RED and limiting the number of packets unresponsive flows can have in the queue.

All TCP flows are subject to RED. For unresponsive flows, CBT defines two thresholds, one for tagged flows and one for all others. It counts the packets each of these groups have in the queue. When they pass their respective thresholds, the arriving packet belonging to that group will be dropped. If they are still below their threshold, the packet will be subject to the RED algorithm. Thus a non-TCP packet may be dropped either because there is already a lot of packets from its group in the queue or because RED decides to drop it. Note that there is still one first-in-first-out (FIFO) queue for the outgoing packets in the router. Figure 1.7 shows the throughput of aggregate TCP under the same experiment as Figure 1.6, with CBT controlling the

queue.

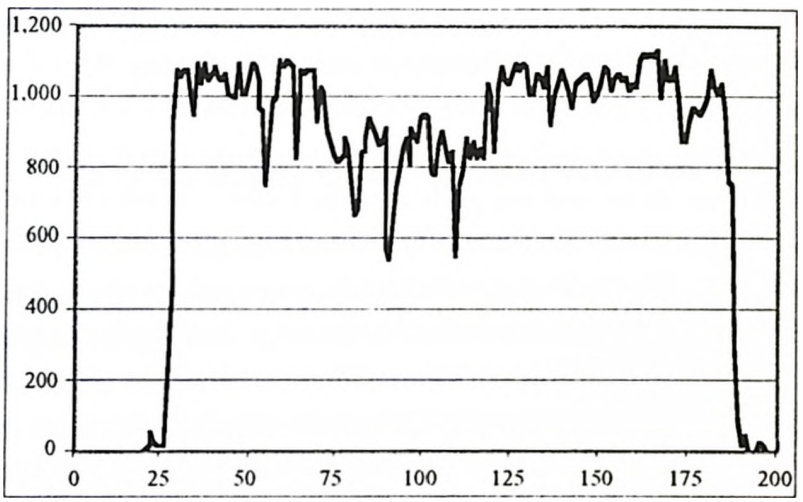


Figure 1.7: The aggregate TCP throughput under CBT in kilobytes/second versus elapsed time in seconds when there is an unresponsive source

In the above algorithm, even though non-TCP flows have their own thresholds, they will still be counted toward total queue size. This version of CBT is called *CBT with RED for all flows*. CBT has another variant, *CBT with RED only for TCP*, where the non-TCP packets do not count toward queue size.

The idea behind "CBT with RED only for TCP" comes from the point that unresponsive flows, in the worst case, do not respond to drops made by RED. Therefore as long as they have not passed their threshold, there is no need to drop more packets from these groups. On the other hand, by considering the non-TCP packets in the average queue size, TCP flows may be punished more than what is necessary.

One issue regarding "CBT with RED only for TCP" is how to set the minimum and maximum thresholds for RED. One option is to deduct the number of allocated slots for unresponsive flows from the total queue capacity

and then determine the thresholds. Another possibility is to increase the size of the queue by the number of allocated slots for unresponsive flows. As discussed in [13], the TCP throughput in both cases is equal but in the second alternative, the latency and drop rate are marginally higher.

1.3.7 Flow Random Early Drop (FRED) [14]

FRED tries to provide a fair share of bandwidth to all "active flows" passing through a router. Active flows are those who have at least one packet in the queue. In RED, all flows are marked with the same marking probability. As a result, flows with higher rates see more marked packets than those with lower rates. In the presence of an unresponsive flow, this is not enough to ensure fairness (see Figure 1.6). Since TCP reduces its sending rate significantly when it detects a packet loss, any source that does not respond to packet loss in the same way will end up using more than its fair share of bandwidth.

The bandwidth share of a flow is proportional to the flow's buffer size. FRED counts packets and keeps statistics for each active flow so that it can impose a separate loss rate on each flow to achieve fairness.

In [14], flows are categorized into three different groups: non-adaptive (same as unresponsive), robust, and fragile.

A source of a non-adaptive flow sends its data with a certain rate and does not slow down under congestion. Some audio and video applications with constant bit rate (CBR) fall in this group.

Robust connections always have some packets to send and act aggressively toward taking bandwidth. They respond to packet loss and are capable of retransmitting the lost packets quickly. They have enough packets in the queue so that they can obtain at least their fair share of bandwidth. An FTP connection with small RTT falls in this group.

Fragile connections are either sensitive to packet loss or slow in taking the available bandwidth. Like robust connections, fragile connections also respond to packet loss but they have less packets buffered in queue. A telnet application

or a TCP connection with small congestion window falls in this group.

FRED acts similar to RED but it introduces more parameters and variables. Beside average queue length (avg), FRED defines average per connection queue length ($avecq$), which is calculated by dividing avg by the number of active connections. Other than RED's min_{th} and max_{th}, FRED also defines min_q and max_q. When the average queue size is less than max_{th}, an arriving packet from a connection with less than min_q packets in queue will always be accepted. This helps fragile connections to take their fair share of bandwidth.

FRED does not let any connection have more than max_q packets in the buffer. It counts the times a connection tries to exceed that limit and stores that in a per flow *strike* variable. Flows with high strike values are restricted to have at most $avecq$ packets in queue. This will protect responsive flows from unresponsive ones.

The other difference between RED and FRED is in the way they update avg. While RED updates avg only on packet arrival, FRED updates it both on packet arrival and departure. This will make avg to be a better estimation for the average queue size. For a detailed description of FRED, the reader is referred to [14].

1.3.8 Dynamic-RED (DRED) [15]

One of the problems of RED is that its equilibrium queue length depends on the number of active TCP connections (as shown later in this section), therefore it cannot stabilize the queue size. DRED, on the other hand, tries to bring the queue length to a given target, independent of the number of active TCP connections. It uses a control-theoretic approach to adjust drop probability and manage congestion. Unlike some of the AQM schemes discussed before, it does not try to estimate the arrival rate or the number of active connections and it does not keep any state information for current flows.

DRED is basically a closed-loop feedback control system. As shown in Figure 1.8 (Figure 2 in [15]), TCP senders and receivers along with all routers

and network devices are considered as a process (or plant). Queue size is a monitored variable that is compared to the desired target queue size. Based on the resulting error signal, DRED updates the value for drop probability in attempt to drive the error to zero.

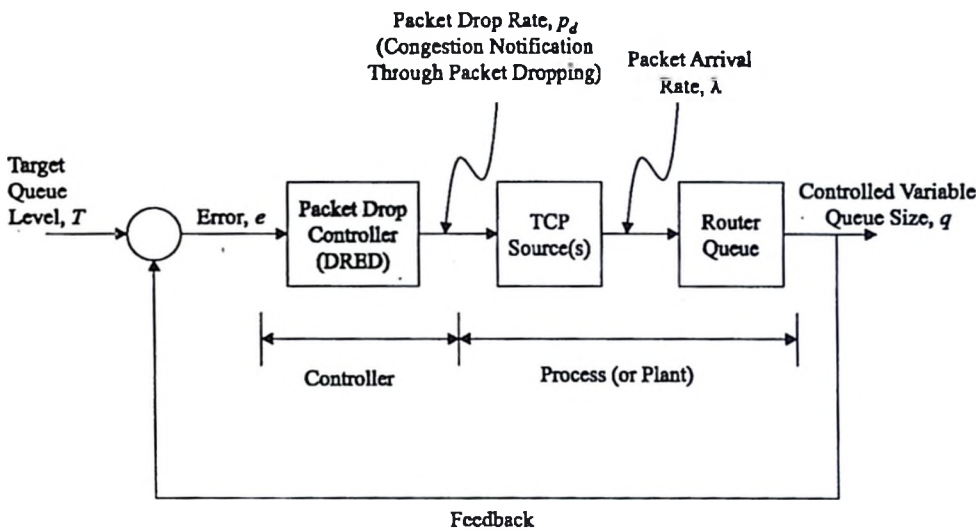


Figure 1.8: DRED acting as a controller in a closed-loop feedback control system

DRED is a discrete integral controller that tries to bring the difference between current queue size, $q(n)$, and target queue size, T, to zero. The queue size is sampled every Δt seconds and the error signal is defined as

$$c(n) = q(n) - T$$

To accommodate burstiness and rate fluctuation, DRED calculates the filtered error, $\tilde{c}(n)$, by applying a low pass filter with gain $0 < \beta < 1$ on the error signal:

$$\tilde{c}(n) = (1 - \beta)\tilde{c}(n-1) + \beta c(n).$$

The filtered error is normalized when it is used to calculate the drop probability. If the target queue size dynamically changes over time, the normalization factor will be $2 * T(n)$, otherwise it will be the buffer size B. As an integral controller, DRED calculates the drop probability based on

$$p_d(n) = p_d(n-1) + \alpha \frac{\hat{e}(n)}{B},$$

where α is the control gain.

DRED also introduces an upper bound θ for the drop probability . Since p_d cannot be negative, the final drop probability of DRED is calculated in the following manner:

$$p_d(n) = min\left\{max\left[p_d(n-1) + \alpha \frac{\hat{e}(n)}{2T(n)}, 0\right], \theta\right\}. \tag{1.8}$$

In order to have a high link utilization, DRED introduces an additional no-drop threshold parameter L. When the queue size is smaller than L, DRED does not drop any packets but continues to update the drop probability using (1.8). This way there is no need to re-initialize the drop probability. The suggested value for L in [15] is $L = 0.9T$.

The sampling interval, Δt, should be much smaller than the time it takes for the router to transmit T packets. This way, the controller can react effectively when the queue size approaches or exceeds T. This time depends on the output link speed and average packet size. For example, if the target queue size is 500 packets, we may choose Δt to be the time it takes to transmit 10 packets of average size.

Figure 1.9 (Figure 5 in [15]) shows the network used for simulation of DRED and Figure 1.10 (Figure 6 in [15]) shows the resulting instantaneous queue size and drop probability for 1000 TCP connections. Figure 1.11 shows the queue size when the target is dynamically changed from $B/2$ to $2B/3$ to $B/3$ and finally back to $B/2$. As shown in these diagrams, DRED is successful in keeping the queue size around the target and can quickly shift the queue size to a new target. Amongst the benefits of a stabilized queue size are bounded

delay and high link utilization. Many audio or video streaming applications may be more interested in bounded delay than in a high average bandwidth with highly variable delay (high jitter).

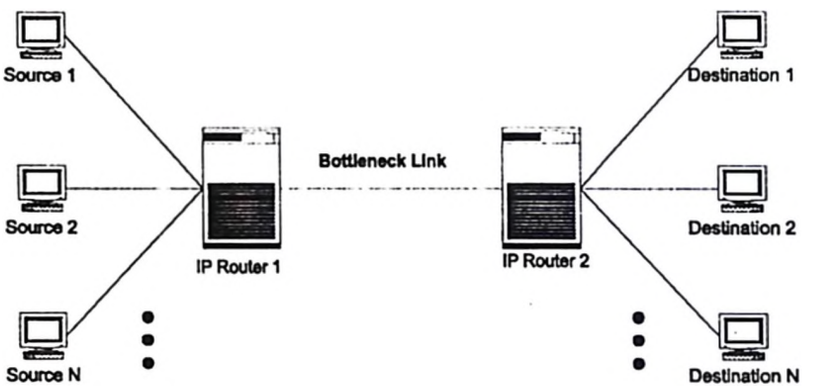


Figure 1.9: The simulation network for DRED

In [15] there is also a simulation for TCP connections with different round trip times. Even though DRED keeps the total queue size near the target, this does not mean fair bandwidth sharing between connections. The rate at which each source is sending its data is equal to its window size divided by its round trip time. Even with equal window sizes, those connections with smaller round trip times get more bandwidth.

RED and DRED are compared in [15] in terms of drop probability. The drop probability for RED in its simple form (where $min_{th} = 0$) is given by

$$p_d = \frac{q_a}{max_{th}} max_p. \tag{1.9}$$

With the drop probability given in (1.9), the average TCP window size can be approximated by ([16])

$$w = \frac{0.87}{\sqrt{p_d}},$$

or

$$p_d = \frac{0.76}{w^2}. \tag{1.10}$$

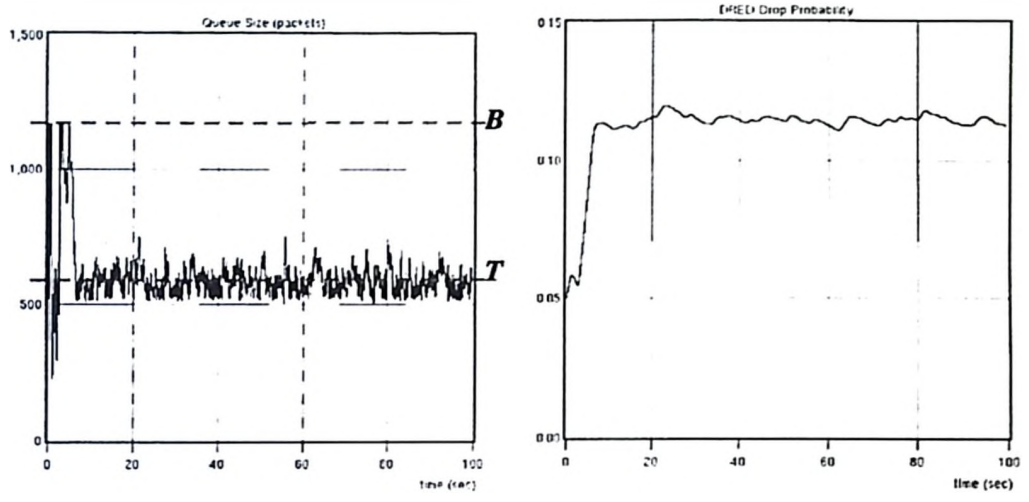


Figure 1.10: DRED queue size and drop probability under 1000 TCP connections

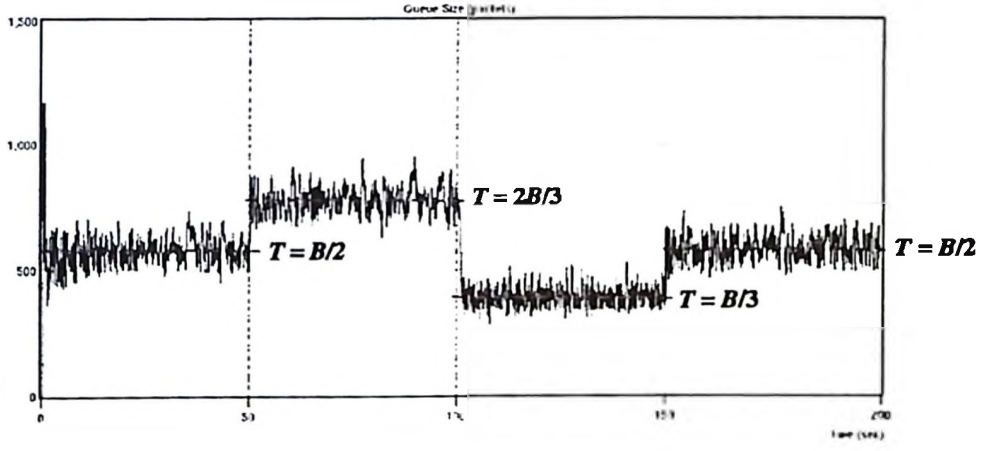


Figure 1.11: DRED queue size while dynamically changing the target queue size

If we assume there are N active connections in the queue, the average window size for each connection will be $w = q_a/N$. By substituting this value in (1.10) we get

$$p_d = 0.76\frac{N^2}{q_a^2}. \tag{1.11}$$

Equating (1.9) and (1.11) and solving for q_a results in

$$q_a = 0.91N^{2/3}\left(\frac{max_{th}}{max_p}\right)^{1/3}. \tag{1.12}$$

Equation (1.12) tells us that the equilibrium average queue size in RED depends on the number of active connections. Also, RED acts the same as DropTail after the average queue size reaches max_{th}. By substituting max_{th} for q_a in (1.12) and solving for N we get

$$N = (1.15)max_{th}\sqrt{max_p}. \tag{1.13}$$

When the number of active connections exceeds N, RED will act the same as DropTail. On the other hand, for DRED in its equilibrium state, if the drop probability has converged to a fixed value $p_{d,e}$ and the queue size has converged to a value q_e we have

$$p_{d,e} = p_{d,e} + \alpha\frac{q_e - T}{B}, \tag{1.14}$$

which results in $q_e = T$. This shows that DRED can bring the queue size to its desired target independent of the number of active connections.

1.4 Comparing AQM Algorithms

There are a few references that have compared different AQM algorithms via simulation. This section briefly summarizes the results of two papers ([17] and [18]). In [17], RED, BLUE, DRED, and SRED are simulated using the OPNET simulator (a commercial network simulation package). The performance metrics used are queue size, drop probability, and packet loss rate. The simulation

network is similar to Figure 1.9 with 1000 TCP connections. Parameter values used for algorithms are those recommended in their corresponding papers.

As observed in [17], RED and BLUE are incapable of stabilizing the queue size. They suffer from periods of queue underflow and overflow. On the other hand, DRED is successful in stabilizing the queue size around the given target.

Regarding drop probability, RED has the highest drop probability. BLUE does not react fast enough since it uses buffer overflow and underflow as signals of traffic load change. Compared to BLUE, DRED seems to adapt faster.

Comparing the packet loss rate, RED has the highest loss rate. Both BLUE and DRED seem to have almost the same loss rate but BLUE appears to show less variation.

In general, since all of these AQM schemes use probabilistic packet drop, they have an advantage over drop tail by avoiding a bias against bursty traffic. At the same time, dropping randomly can avoid global synchronization.

Based on the original papers for each of these AQM schemes and the comparisons made in [17] and [18] the combined observation can be summarized as follows:

- In RED the queue size depends on the number of active connections and thus cannot be stabilized. It is difficult to set the parameters and in particular they should be adjusted based on network congestion.

- BLUE has a simple algorithm and it is easy to implement, but it reacts slowly to load changes. In particular when the RTT is large, it takes a long time for sources to adjust their sending rates. This will result in alternating periods of queue overflow and underflow.

- FRED protects TCP flows from misbehaving flows but it keeps a per-flow state. In the presence of encryption, packet header information may not be accessible to recognize the flow.

- DRED can keep the queue size around its desired target, if the buffer size is sufficiently large (typically twice the bandwidth delay product of

the network)

- Other than CBT and FRED, other AQM schemes cannot act fairly when there are one or more misbehaving sources competing with TCP sources. Consequently, most of the outlink bandwidth will be assigned to the unresponsive flow.

As we have seen in the previous sections, there are different factors and parameters an algorithm can pick to measure network congestion. There are two main categories ([18]): queue-based algorithms and load-based algorithms. Load-based AQM schemes, such as GREEN, try to estimate arrival and departure rates while on the other hand, queue-based schemes, such as RED, DRED, and FRED, look at queue size as a measure for network congestion. Some algorithms like REM try to include both factors in their design. Table 1.2 summarizes different congestion measurement factors and algorithms using them.

Factor	**Algorithm**
Instantaneous queue size	DropTail, BLUE, REM
Average queue size	RED, CBT, FRED, DRED
Link idleness	BLUE
Aggregate input rate	REM
Aggregate output rate (link capacity)	REM, GREEN
Estimated arrival rate	GREEN

Table 1.2: Congestion measurement factors and algorithms using them

1.5 Factors Affecting the Performance of an AQM Scheme

No matter which AQM scheme is chosen for a router, there are several factors that may affect the performance. These factors include:

- values chosen for the algorithm-specific parameters
- number of active connections
- round trip time of the sources
- presence of one or more unresponsive flows
- bursty traffic

A number of the AQM schemes are designed to explicitly deal with some subset of the above problems. For example, FRED can protect responsive flows from unresponsive ones and DRED brings the queue size to the given target, independent of the number of sources.

1.6 Performance Measures of Interest

In designing a new AQM algorithm, there are two groups of performance measurements one should consider. The first group is to the interest of end-users and the second group is to the interest of the link or router owner.

End-users may be interested in the following items:

- low delay
- low delay jitter
- high throughput
- low throughput jitter
- low packet loss

A link or router owner may be interested in

- high link utilization
- short queue length (smaller buffer size)

- an algorithm that is not very sensitive to the values of its parameters and thus there is no need to change its parameters based on different congestion situations

- an algorithm that gives the administrator the power to decide how to divide the bandwidth amongst flows or sources (for example, to provide a fair allocation of bandwidth to flows)

Usually AQM schemes try to have benefits for both groups. For example, by keeping the queue size around a desired target, DRED can provide both fixed delay (low delay jitter) and high link utilization.

Chapter 2

The Network Simulator: NS-2

NS [19] is a discrete event simulator designed for simulating networks. It first appeared in 1989 and since then it has evolved substantially [20]. NS is an on-going project, with many network protocols and routing algorithms for both wired and wireless networks having been implemented. It has been used in many network related articles to simulate a new approach or idea and is pretty much the standard package in the academic environment (along with OPNET, but OPNET is commercial). NS has been ported to different computer architectures and the program itself and its source code are available for free.

NS is an object oriented program which makes it easy to write new modules in order to expand it. It is written in two languages: C++ and OTcl. C++ is used when a programmer needs low level functionalities or when a large set of data will pass through a function and run-time speed is important. Most protocols, for example, are written in C++. On the other hand, those parts of the simulator that will run only once or need slight adjustments to the existing C++ classes are written in OTcl. Writing and changing code in C++ takes more time, but after compilation will run faster than OTcl. On the other hand, writing and manipulating code in OTcl is easier and faster than C++ but the resulting code will be slower.

NS acts like an OTcl interpreter that recognizes some extra network-related classes. A user can supply the network topology and simulation scenario within a .tcl file which makes it convenient if one wishes to quickly change network parameters or modify simulation scenarios.

The code in Figure 2.1 is a simple Tcl script which shows how one can define network topologies and simulation scenarios in NS. Comments (lines starting with #) are inserted into the code to make it more readable. The code simulates an FTP connection between two hosts that are connected through two routers. The DropTail algorithm is used for all queues in the network. The FTP application starts sending data at time 0.5 seconds and stops at time 3.5 seconds. The simulation ends at time 4.0 seconds. Classes like `Simulator`, `Agent/TCP`, and `Application/FTP` are network related classes that NS recognizes, in addition to the usual OTcl classes.

NS supports two class hierarchies: one in C++ which is called the compiled hierarchy and the other one in OTcl which is called the interpreted hierarchy. These two hierarchies are similar to each other. From the user's view, there is a one-to-one relation between classes in these two hierarchies. There are certain classes in C++ and OTcl that map between these two hierarchies and build corresponding objects. There are also classes that enable code written in C++ to access the interpreter and run OTcl code (and vice versa). The next sections will describe these classes. Note that in order to simply use the simulator, one does not need to know about these classes but it is necessary for expanding or modifying the current version of NS.

2.1 Accessing the Interpreter: class Tcl

The class Tcl is used to get an instance of the interpreter in the C++ code. By having access to the interpreter, the programmer can run OTcl commands from C++ environment and retrieve their results. The statement required to get access to the interpreter is

```
# creating an instance of the NS simulator
set ns [new Simulator]

# creating source and destination nodes (or hosts)
# and two routers
set s1 [$ns node]
set d1 [$ns node]
set r1 [$ns node]
set r2 [$ns node]

# building the network topology
$ns duplex-link $s1 $r1 1Mb 10ms DropTail
$ns duplex-link $r1 $r2 1Mb 10ms DropTail
$ns duplex-link $r2 $d1 1Mb 10ms DropTail

# objects required to simulate a TCP connection
set tcpAgent [new Agent/TCP]
set tcpSink [new Agent/TCPSink]

# attaching TCP related agents to their corresponding nodes and
# establishing the connection
$ns attach-agent $s1 $tcpAgent
$ns attach-agent $d1 $tcpSink
$ns connect $tcpAgent $tcpSink

# creating a FTP application to generate data for the TCP agent
set ftpApp [new Application/FTP]
$ftpApp attach-agent $tcpAgent

# the simulation scenario
$ns at 0.5 "$ftpApp start"
$ns at 3.5 "$ftpApp stop"
$ns at 4.0 "exit 0"

# running the simulation
$ns run
```

Figure 2.1: An example of a NS simulation script

```
Tcl& tcl = Tcl::instance()
```

The class Tcl has four methods to invoke an OTcl command. These methods differ only in the way they receive their arguments. One of these methods is `tcl.evalc(const char *s)` where the programmer passes an OTcl command as a constant string to the interpreter. The interpreter will then run the command and put back its result as a string in a private member variable (`tcl_->result`). To access this variable, the programmer should call `tcl.result(void)`. Figure 2.2 gives an example on how to use these methods.

```
Tcl& tcl = Tcl::instance()

# find out number of interfaces the simulator has
tcl.evalc("Simulator set NumberInterfaces_");

# get back the result
char* ni = tcl.result();

# if the number of interfaces is other than one, set it to one
if (atoi(ni) != 1)
    tcl.evalc("Simulator set NumberInterfaces_ 1");
```

Figure 2.2: An example of running an OTcl command from C++ environment

Similarly, when the interpreter calls a C++ method, it expects the result to be in the same private member variable. To set this variable, the programmer can use `tcl.result(const char *s)`. In case there is an error, the programmer can instead use `tcl.error(const char *s)`, which will print its argument to the standard output, print `tcl_->result`, and finally exit with error code 1.

2.2 Class TclObject

The class TclObject is the base class for most interpreted and compiled classes in NS. It is also responsible for calling the appropriate constructor of the shadow class in the compiled hierarchy.

By convention, "/" in NS script represents sub-classing. So for example, a class named Agent/UDP is a subclass of Agent which itself is a subclass of TclObject. So a command like

```
set UDPAgent [new Agent/UDP]
```

causes NS to call the constructor of the class Agent/UDP which in turn will call its parent class constructor until the constructor for TclObject is called. Later, TclObject will create the shadow class in the compiled hierarchy. Also, when an object is destroyed in the interpreter, the destructor of its parent class will be called until it reaches the TclObject destructor which in turn will destroy the shadow object in the compiled hierarchy.

2.2.1 Variable Binding: the bind method

TclObject provides a couple of methods that make C++ variables accessible in Tcl script (and vice versa). In this case, a change in a member variable in the compiled or interpreted hierarchy will change the value of the bound variable in the interpreted or compiled hierarchy, respectively. Binding is done in the constructor of the class in the compiled hierarchy.

Since NS supports five different data types (integer, real, bandwidth, time, and boolean), for each of these data types, there is a different bind method. So for example, for integers, the programmer should use the following method

```
void bind(const char* var, int* val),
```

where the first argument is the name of the variable in the interpreted hierarchy and the second argument is the address of the variable in the compiled hierarchy. As an example, if we define a new class in C++ for DRED and

define an integer member variable T for the target, the following code will make T accessible in its corresponding class in the interpreted hierarchy with the same name:

```
bind("T", &T)
```

Later in the NS script code, the user can set the value for queue target to its new value (630 for example) by executing

```
$object set T 630
```

where object is the variable holding the DRED object. If a new value is not supplied, NS will return the current value of the variable.

When a new object is created in NS, all of its bound variables are assigned their default values. The default values should be specified as interpreted class variables. Usually these class variables are defined in a file called ns-default.tcl. If the programmer has not specified a default value for a bound variable, NS will generate a warning message when creating an object of that class. In our examples, the DRED class is a sub-class of the Queue class. Therefore, by putting the following line in the file ns-default.tcl, all DRED classes created will get 630 as their default value for T

```
Queue/DRED set T 630
```

Besides C++ variables, the programmer can also make methods of an object in the compiled hierarchy accessible in a Tcl script. The next section describes the mechanism and the procedure required for this purpose.

2.2.2 Running C++ methods from Tcl: the command method

When a user places an identifier in front of an object in a Tcl script, NS first checks to see if there is a procedure with the same name defined in the interpreted class of the corresponding object and if so, it will call that procedure.

If there is no procedure defined with the given name, NS will pass the identifier along with all other arguments to a special method in the shadow-object in the compiled hierarchy, called `command`. The `command` method is originally defined in the TclObject class and is inherited to all of its subclasses. The user can override the default behavior of this method and write his own version to handle special cases. Here is the prototype of this method

```
int TclObject::command(int argc, const char*const* argv)
```

where `argc` holds the number of arguments and `argv` is an array holding the arguments. The elements in `argv` are as follows

- `argv[0]` holds the fixed string "cmd" which is actually the name of the procedure in the interpreter hierarchy which calls the `command` method of the shadow-object in the compiled hierarchy
- `argv[1]` is the name of the identifier
- `argv[2...agrc -1]` are parameters passed along with that identifier (if any)

Since the identifier and all other arguments are passed to the `command` method, the programmer can parse the given string and based on that decide on the next action which can be as simple as executing a single statement, or as complex as calling several methods. In case the `command` method does not know how to handle a special case, it should call the `command` method of its parent class. The return value of the `command` method should be either `TCL_OK` if the operation was successful or `TCL_ERROR` if there was an error.

The following code shows the `command` method of the Queue class, which has been modified so that a user can find out the number of packets in the queue from the Tcl script.

```
int Queue::command(int argc, const char * const * argv) {
    Tcl& tcl = Tcl::instance();
```

```
        if (argc == 2) {
            if (strcmp("length", argv[1]) == 0) {
                tcl.resultf("%d", pq_ -> length());
                return(TCL_OK);
            }
        }
        return (Connector::command(argc, argv));
    }
```

By compiling the above code, "length" appears as a procedure for class Queue or any of its subclasses in the interpreted hierarchy. The following NS script shows how a user can learn about the current queue size of an object of type DRED by calling the virtual `length` procedure of the object.

```
set queueObj [new Queue/DRED]
puts "Current queue size:" [$queueObj length]
```

2.3 Class TclClass

TclClass is a virtual class that provides two main functionalities to its subclasses. First, classes derived from TclClass specify their interpreted hierarchy through the TclClass constructor and second, they indicate which class in the compiled hierarchy should be instantiated through their inherited `create` methods. This way, NS can build the interpreted hierarchy and associate each Tcl object with its mirror in the compiled hierarchy.

As an example, the following code shows how the DRED class is implemented under NS. To make DRED a subclass of the Queue class in the interpreted hierarchy, we have to call the constructor of the TclClass with the argument `Queue/DRED`. Later in the `create` method, an object of type DRED, a subclass of TclObject, is returned.

```
static class DREDClass : public TclClass {
```

```
    public:
        DREDClass() : TclClass("Queue/DRED") {}
        TclObject *create(int argc, const char * const *argv) {
            return (new DRED);
        }
} class_DRED;
```

The arguments passed to the `create` method in `argv` are as follows:

1. `argv[0]`: name of the object.
2. `argv[1]`: pointer to the object (`$self`).
3. `argv[2]`: name of the interpreted class (`Queue/DRED` in our example).
4. `argv[3]`: name of the procedure that called the `create` method (`create-shadow`).
5. `argv[4..argc-1]`: any additional argument(s) that the user may have supplied.

Note that the above code defines a static class: `class_DRED`. As a result, when NS first starts running, it will call the constructor of the above class and all other static classes and will build the interpreted hierarchy based on them. It registers the class `Queue/DRED` with `DREDClass`, so that later when a user enters a command like `set DREDObj [new Queue/DRED]` in a Tcl script, NS creates an object of type `Queue/DRED` in the interpreted hierarchy and its mirror in the compiled hierarchy by calling the `create` method of the class `DREDClass`.

Chapter 3

Improving DRED

This chapter examines DRED in detail and tries to give some suggestions to improve its performance based on the experiments and observations made during simulations.

3.1 Drop Tail Effect

During different simulations, especially when the number of sources was significant, it was noticed that at the beginning, when all sources start at the same time, the queue size reaches its maximum limit which will consequently cause several packet drops. This continuous drop does not let TCP sources employ fast retransmit and thus will make them fall into time-out. As a result, after reaching the maximum queue size, the queue then empties and stays empty for a short period of time. Although this can be considered only as a transient effect, if the buffer size is not large enough or the target is close to the maximum buffer size, this situation can happen more frequently. In the latter case, when the queue reaches its maximum, the error (from the target) may not be large, but the price TCP sources pay because of tail-dropping will be considerable.

The problem is that tail-dropping can significantly affect the sending rates

of the sources, but DRED does not consider this when calculating its drop probability, as it only has the queue size as its congestion measure. So as long as DRED does not bring the effect of tail-drop into its calculation, it is better to avoid small buffers (compared to the bandwidth-delay product of the network) and have the target far from the maximum limit. As in the original paper of DRED and in the simulations done during this thesis, the target was half the maximum queue size and the maximum queue size was twice the bandwidth-delay product of the network.

3.2 DRED as a Controller

Suppose the problem of regulating the queue length were treated as a control problem. In other words, we want the queue length, q_t (the monitored variable), brought to a target value of T by using the drop probability as a control signal. Equivalent to bringing q_t to T is to bring the error variable $e_t = q_t - T$ to zero. Feedback control based on e_t is used to generate the appropriate control signal.

Even though different kinds of controllers can be found nowadays, the classic one used in practice is the PID controller which consists of three components, called proportional, integral, and differential corresponding to the initials P, I, and D, respectively. The difference between these components is the way their control signal is generated based on the error signal.

As was mentioned earlier in Chapter 1, DRED acts as an integral controller. An integral controller calculates the area under e_t and tries to force the convergence of this area to a constant. In other words, its control signal, I_t, is calculated as follows:

$$I_t = \alpha_i' \int_0^t e_\tau d\tau,$$

where α_i' is the integral gain. Differentiating both sides with respect to t gives us

$$\frac{dI_t}{dt} = \alpha_i' e_t.$$

which in discrete time (with sampling interval Δt) can be written as

$$\frac{I_{n\Delta t} - I_{(n\Delta t - \Delta t)}}{\Delta t} = \alpha_i' e_{n\Delta t},$$

or

$$I_{n\Delta t} = I_{(n\Delta t - \Delta t)} + \alpha_i' e_{n\Delta t} \Delta t,$$

By defining $I(n) = I_{n\Delta t}$, $e(n) = e_{n\Delta t}$, and $\alpha_i = \alpha_i' \Delta t$ we have

$$I(n) = I(n-1) + \alpha_i e(n). \tag{3.1}$$

Being a pure integral controller, the drop probability of DRED, $p_d(n)$, is equal to $I(n)$. As shown in Section 1.3.8, when $I(n)$ converges, $e(n)$ converges to zero which in turn means $q(n)$ converges to T.

For a proportional controller, the control signal, P_t, is directly proportional to the error signal. More precisely

$$P_t = \alpha_p e_t,$$

which becomes

$$P(n) = \alpha_p e(n) \tag{3.2}$$

in discrete time, where α_p is the proportional gain. If we normalize $e(n)$ by $2T$ in (3.2) and solve for $q(n)$ we will get

$$q(n) = T + \frac{2TP(n)}{\alpha_p}, \tag{3.3}$$

which means that if $P(n)$ converges to any value other than zero, the queue size will differ from its target by an offset of $\frac{2TP(n)}{\alpha_p}$. Since we do not know in advance the value that $P(n)$ will converge to, we cannot predict the offset value.

A differential controller with a gain of α_d' generates its control signal D_t as

$$D_t = \alpha_d' \frac{de_t}{dt}.$$

which in discrete time becomes

$$D_{n\Delta t} = \alpha'_d \frac{e_{n\Delta t} - e_{(n\Delta t - \Delta t)}}{\Delta t},$$

or

$$D(n) = \alpha_d(e(n) - e(n-1)) \tag{3.4}$$

by setting $\alpha_d = \alpha'_d/\Delta t$. The differential controller is suitable for variables that have smooth transitions. If the dynamics of the monitored variable are very fast or there is a significant amount of noise in the system, then the use of a differential controller can be problematic.

The control signal for a PID controller, $U(n)$, is simply the sum of the control signals of its elements. In other words

$$U(n) = P(n) + I(n) + D(n)$$

which by using (3.1), (3.2), and (3.4) becomes

$$\begin{aligned} U(n) &= I(n-1) + (\alpha_p + \alpha_i + \alpha_d)e(n) - \alpha_d e(n-1), \\ I(n) &= I(n-1) + \alpha_i e(n). \end{aligned}$$

To generalize DRED, we can choose $U(n)$ as the drop probability. In fact $U(n)$ will be equal to $p_d(n)$ if we set $\alpha_p = \alpha_d = 0$ (as $U(n)$ is to be interpreted as a drop probability, it must be constrained to be between zero and one, an issue which is dealt with in the next section).

3.2.1 Constraining the control signal

Since the drop probability should lie between zero and one, if $U(n) < 0$ or $U(n) > 1$, we can simply replace it by zero or one, respectively. However, the problem is how we should update the value of $I(n)$, after changing $U(n)$. This must be considered since the current value of $I(n)$ will be used to compute $U(n+1)$ while the outputs from the proportional and differential components at time n will not be used to compute the control at time $n+1$. In other

words, the proportional and differential components do not have memory of their previous states and their values are just based on $e(n)$ and $e(n-1)$, while the integral component uses its last state value, $I(n-1)$, to calculate the value of its current state $I(n)$.

There are different ways to change the value of $I(n)$ in these special cases. One way is set its new value, $I_{new}(n)$, as

$$\begin{aligned} I_{new}(n) &= 1 - (P(n) + D(n)) \qquad \text{when } U(n) > 1 \\ I_{new}(n) &= 0 - (P(n) + D(n)) \qquad \text{when } U(n) < 0. \end{aligned} \tag{3.5}$$

In some cases, the above formula can change the value of $I(n)$ drastically. For example, if all three components are positive and $P(n) + D(n) > 1$, $I_{new}(n)$ will be negative.

The other method we can use when $U(n) > 1$ is to scale down the value of $I(n)$ by the same factor that $U(n)$ is scaled down. In other words, if the new value for $U(n)$ is now 1, the new value for $I(n)$ will be:

$$\begin{array}{ccc} \text{Old value} & & \text{New value} \\ U(n) & \rightsquigarrow & 1 \\ I(n) & \rightsquigarrow & I_{new}(n) \end{array}$$

$$I_{new}(n) = \frac{I(n)}{U(n)}.$$

We cannot use this method when $U(n) < 0$ since it will make $I_{new}(n)$ zero.

Maybe the best way to deal with this problem is to change the value of $I(n)$ proportional to its contribution in making $U(n)$ greater than one or less than zero. To implement this idea, we first have to find the absolute sum of the three components

$$absolute\ sum = |I(n)| + |P(n)| + |D(n)|,$$

and then in the case $U(n) > 1$, $I_{new}(n)$ will be

$$I_{new}(n) = I(n) - \frac{(U(n) - 1)|I(n)|}{absolute\ sum}$$

and if $U(n) < 0$,

$$I_{new}(n) = I(n) + \frac{(0 - U(n))|I(n)|}{absolute\ sum}.$$

We chose the last method and implemented it in DRED. Simulation results for the modified algorithm are discussed in the next section.

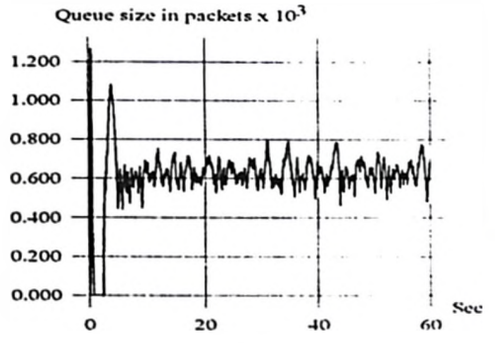
3.3 Simulation Results for DRED with PID Controller

We added the two other components of a PID controller (proportional and differential) to DRED and simulated the network in Figure 1.9 for 60 seconds with 100 TCP connections and queue target set to 630 packets.

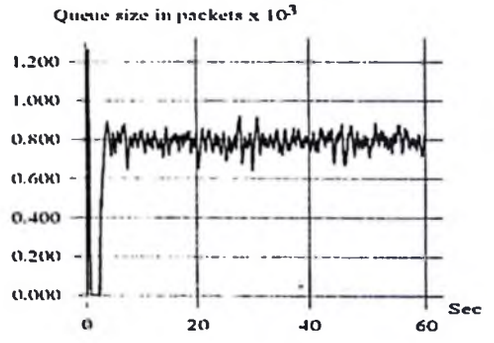
At first we tried to stabilize the queue with only one component involved. As we know, the integral controller by itself is the same as DRED. For proportional and differential components, different simulations were done to find proper values for α_p and α_d. The values chosen were $\alpha_p = 0.05$ and $\alpha_d = 5$. For simplicity and to be able to match $c(n)$ with $p_d(n)$, β was set to 1 for all simulations. Figures 3.1 and 3.2 show the queue size, drop probability, and normalized error, $\frac{c(n)}{2T}$, for each of the controllers.

As shown in Figure 3.1*b*, a pure proportional controller can stabilize the queue size, but as predicted, there is an offset. The queue size is stabilized around approximately 800 packets instead of around 630 packets. If we take the average value of $P(n)$ from Figure 3.1*d* to be 0.007, (3.3) gives us the value of 806.4 for $q(n)$ when stabilized. On the other hand, even though the drop probability for a pure differential controller has almost the same magnitude as the integral controller(10.0×10^{-3}), it is not successful in stabilizing the queue size. That is because the normalized error appears to have high frequency components and differentiating such a signal does not produce a smooth drop probability.

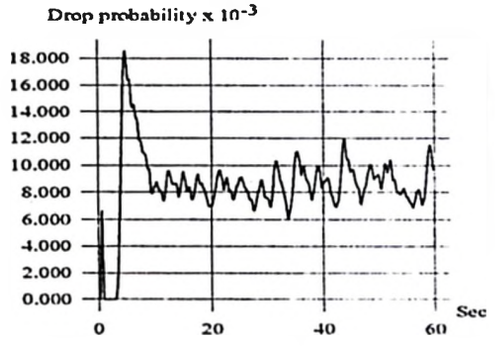
Based on the results we got from the first set of simulations, we decided to include both proportional and integral components in DRED and run the



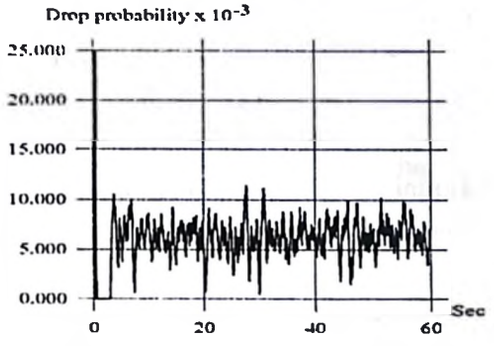
a) Queue size for integral comp.



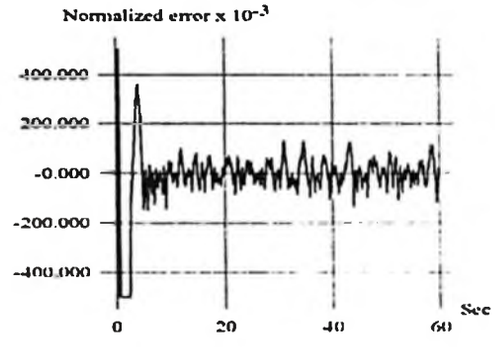
b) Queue size for proportional comp.



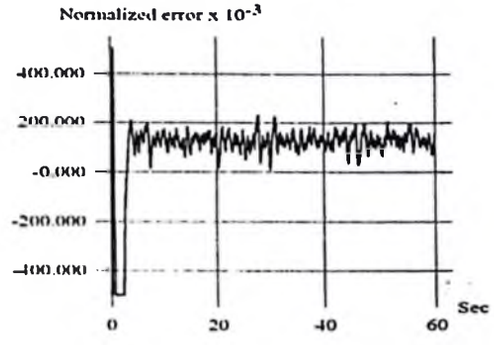
c) Drop probability for integral comp.



d) Drop probability for proportional comp.

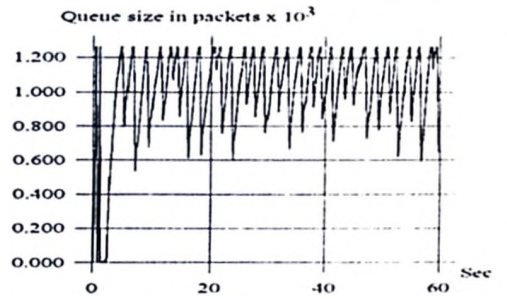


e) Normalized error for integral comp.

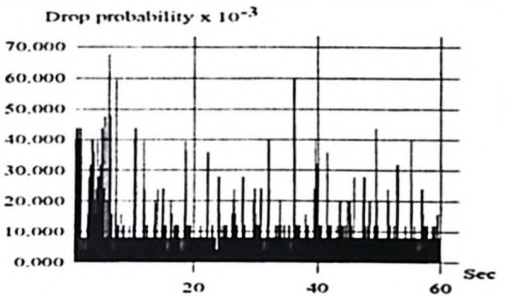


f) Normalized error for proportional comp.

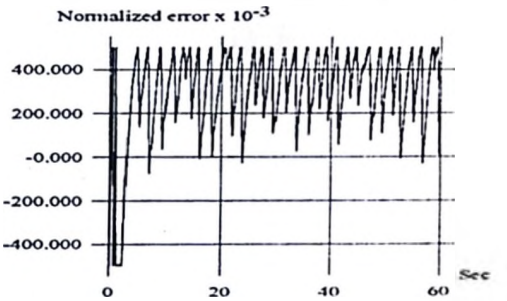
Figure 3.1: Queue size, drop probability, and normalized error for integral and proportional controllers



a) Queue size for differential comp.



b) Drop probability for differential comp.



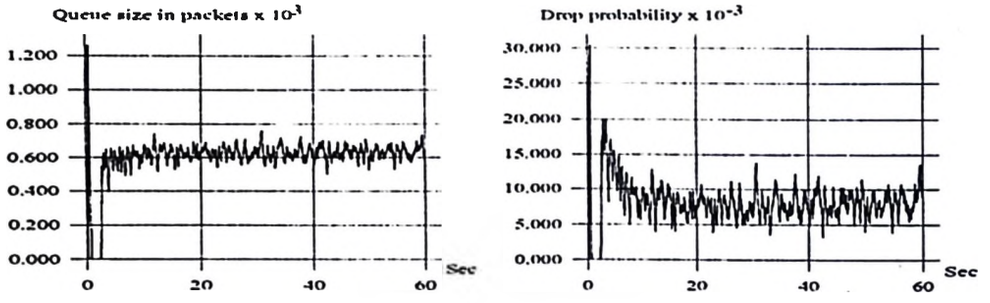
c) Normalized error for differential comp.

Figure 3.2: Queue size, drop probability, and normalized error for differential controller

simulation again. The results are shown in Figure 3.3. Comparing the queue size for original DRED (Figure 3.1*a*) and DRED with proportional component (Figure 3.3*a*), the average queue size for the former is 630.77 and for the latter is 630.70 packets which are very close to each other and also to the desired target. However, the standard deviation for the first case is 55.02 and for the second case is a 37.33. This is about 33% improvement. The queue size has less deviation around the target and the drop probability has more variation. That is due to the proportional component which acts faster than the integral component.

We cannot tell if including the proportional component will always improve the results as it depends on the values of the parameters α_p and α_i and also

the network topology. It may be worthwhile to include it with the integral component, however a more rigid conclusion needs further study including more simulations with different scenarios.



a) Queue size with PI controller

b) Drop probability with PI controller.

Figure 3.3: Queue size and drop probability with both proportional and integral comp.

Chapter 4

Implementation of DRED

This chapter describes our implementation of DRED under NS. The implementation closely follows the algorithm described in the first chapter. Names of member variables in C++ code for example match their names in the original article. The code also includes the improvements we made to the original DRED algorithm introduced in Chapter 3.

The following sections explain new classes defined for the implementation along with their methods and member variables. The complete source code is included in the attached CD.

4.1 Class DREDTimerClass

The DRED algorithm needs to update its drop probability every Δt seconds. In order to implement this requirement, we defined `DREDTimerClass`, which is a subclass of an already defined class in the compiled hierarchy: `TimerHandler`. The `TimerHandler` class has a timer that can be set to expire at some specified future time by calling its public functions. When the timer is expired, it will call a virtual method, `expire`. The classes derived from `TimerHandler` should redefine this method to handle the situation after the timer is expired.

4.1.1 Member Variables

There is only one protected member variable in this class,p_, which is a pointer to an object of type DRED. Using this pointer, methods defined in `DREDTimerClass` have access to the public variables and methods of the `DRED` object that the pointer is referring to. The value for this pointer will be set in the constructor of the class.

4.1.2 Methods

• **The Constructor**

```
DREDTimerClass (DRED *p)
```

The constructor of this class receives a pointer to a DRED object as its argument and stores it in `p_`. It also calls the constructor of its parent class: `TimerHandler`.

• **resched**

```
void resched(double delay)
```

This method is inherited from `TimerHandler` and makes the timer expire after `delay` seconds. We use this method to update the drop probability regularly.

• **expire**

```
virtual void expire(Event* e)
```

When the timer expires, this method will be called. We have redefined this method to call the `updatePDrop` method of the DRED object that `p_` is pointing to.

4.2 Class DREDClass

This class is a subclass of `TclClass` and is defined to map the DRED class in the interpreted hierarchy (`Queue\DRED`) to its corresponding class in the

compiled hierarchy (DRED).

4.2.1 Member Variables

This class does not have any member variables.

4.2.2 Methods

• **The Constructor**
`DREDClass()`

The constructor of `DREDClass` calls the constructor of `TclClass` and passes the string "`Queue/DRED`" as its argument. As was discussed in Section 2.3, this will make NS call the `create` method of `DREDClass` when a user instantiates an object of type `Queue/DRED` in a simulation script.

• **create**
`TclObject *create(int argc, const char * const *argv)`

The `create` method receives two arguments and returns an object of type `TclObject` to its caller. The arguments are useful when we expect the user to pass some parameters for creating the DRED object. Since we do not require any parameters, the `create` method returns a new object of class `DRED` as a `TclObject`.

4.3 Class DRED

The `DRED` class is the main class implementing the algorithm. It is a subclass of `Queue` and has its own internal queue of type `PacketQueue`. `PacketQueue` is a class that has already been defined in the compiled hierarchy.

4.3.1 Member Variables

Other than its internal queue (which is stored in the variable `q_`), the `DRED` class has all of the DRED algorithm parameters as its protected member variables. It also has two private member variables: `debug` and `firstPacketArrived`. The `debug` variable is used when the programmer needs to debug the code or experiment with changes to the original code. When the debug mode is on, the value of most variables will be output whenever the drop probability is updated. The variable `firstPacketArrived` is set when the router deploying DRED receives the first packet. From that point, the drop probability will be updated every ΔT seconds. This variable is useful mostly for optimization purposes. It prevents DRED from initially lowering the drop probability when packets have not reached the router because of the propagation delay. Table 4.1 lists the variable names and their descriptions. Note that the variables marked with (*) are bound to the same variable in the interpreted hierarchy so that they can be updated through NS scripts.

4.3.2 Methods

• **The Constructor**
`DRED()`

The DRED constructor has two main tasks: variable initialization and variable bounding. To initialize member variables that are objects themselves (`myTimer` and `q_`), it has to call their corresponding constructors. A reference to this new DRED object being created (`this` pointer) is passed as an argument when calling the constructor of `DREDTimerClass`. This will make `myTimer` call the `updatePDrop` method of the current object. It also calls the constructor of `PacketQueue` to initialize `q_`. The variables `pDrop`, `error`, `filteredError`, `prevFilteredError`, `prevIntegralOutput`, and `firstPacketArrived` are all initialized to 0. The files `dred-queue`, `dred_error`, and `dred-drop-prob` are created in the local directory to store the current queue size, the error from

Variable name	Description
Protected	
q_	internal queue
B(*)	total buffer size in packets
T(*)	target queue size in packets
L(*)	no-drop threshold
deltaT(*)	sampling interval in seconds
alpha(*)	control gain for the integral part
propGain(*)	control gain for the proportional part
diffGain(*)	control gain for the differential part
error	difference between current queue size and target queue size
beta(*)	filter gain for low pass filter
filteredError	the value of the current filtered error
prevFilteredError	the value of the filtered error in the previous state
normalizeOption(*)	if 0, we normalize error by B, otherwise by 2 * T
integralOutput	output from the integral component of the controller
prevIntegralOutput	output from the previous state of the integral component
pDrop	drop probability
theta(*)	pDrop should be between 0 and theta
trace(*)	shows if we want to trace the queue size, drop probability and error from target
qSizeFile	if trace = 1, the queue size will be recorded every deltaT seconds to the file pointed to by this variable
dropProbFile	if trace = 1, the drop probability will be recorded every deltaT seconds to the file pointed to by this variable
errorFile	if trace = 1, the error from target queue size will be recorded every deltaT seconds to the file pointed to by this variable
myTimer	timer used to update pDrop
Private	
debug(*)	if it holds 1, the code will run in debug mode
firstPacketArrived	determines if the router has received the first packet

Table 4.1: Member Variables of the DRED class

target, and the drop probability, respectively, in case `trace` is set to one during the simulation.

- **updatePDrop**

`void updatePDrop()`

The purpose of this method is to update the drop probability periodically. The method does not take any arguments and its return value is of type `void`. This method is the only public method defined in the class. As a public method, the `expire` method of the `DREDTimerClass` can call it when the timer is expired.

The following list outlines the tasks that are performed in this method:

- the `resched` method of the `myTimer` object is called to schedule the timer to expire after ΔT seconds
- the difference between the current internal queue size and the target is calculated and stored in `error`
- `error` is passed through a low pass filter
- based on the `normalizeOption`, the filtered error is normalized either by buffer size (`B`) or twice the target size (`2T`)
- the output for each of the controller's components (integral, proportional, and differential) is calculated separately based on the appropriate control gain (`alpha`, `propGain`, and `diffGain` respectively)
- the drop probability (`pDrop`) is calculated by adding the output of the three components
- in case `pDrop` is less than zero or greater than one, it will become zero or one respectively and the output for the integral component will be updated based on the last method described in Section 3.2.1
- in case the `trace` variable is set, the current values of queue size, drop probability, and error will be recorded in their corresponding files

- if the code is running in debug mode, more internal variables will be printed to the standard output
- the current values of the integral output and filtered error that will be used for the next calculation of `pDrop` are stored in `prevIntegralOutput` and `prevFilteredError` respectively

• **enque**

```
void enque(Packet *)
```

When a packet arrives, NS will call the `enque` method and pass the packet as an argument. In this method we can decide if we want to store the arriving packet in our internal queue or drop it. The internal queue is of type `PacketQueue` and keeps packets in a linked list. Virtually, it does not have a limit on the number of packets it can hold (the available memory is the only limit), but the `Queue` class has a member variable called `qlim_` which is the maximum number of packets allowed in the queue. The `Queue` class itself does not enforce this limit and it is up to its subclasses to control the maximum number of packets in the queue.

The `qlim_` variable of the `Queue` class in the compiled hierarchy is bound to the `limit_` variable of the `Queue` class in the interpreted hierarchy. In our NS scripts, we always set the total buffer size available (`B`) and `limit_` to the same value. So for example if our total buffer size is 1260 packets, the following lines will be placed at the beginning of the simulation script:

```
Queue/DRED set limit_ 1260
Queue/DRED set B [Queue/DRED set limit_]
```

Note that the above code is defining `limit_` and `B` as class variables with our desired values. These values will be used as default values when an object of type `Queue/DRED` is constructed during a simulation.

Here is how the method decides whether it should accept an arriving packet or drop it.

- If the current length of the queue is less than `L` (the no-drop threshold), the arriving packet will be placed at the end of the queue (by calling the `enque` method of `q_`).
- If the queue size is greater than or equal to `qlim_`, the arriving packet will be dropped; otherwise, the current drop probability will determine if we should keep the packet or drop it.
- Finally, if this is the first packet that has arrived at the router, `firstPacketArrived` will be set to one and the timer will be scheduled to expire after ΔT seconds. From that point, the drop probability will be updated periodically.

• deque

```
Packet *deque()
```

After the packet is transmitted over the outgoing link attached to a router, the `deque` method will be called by NS. The DRED algorithm does not require us to do any specific tasks on removing a packet from the queue, so this method simply calls the `deque` method of the internal queue, which in turn removes the first packet in its linked list and returns it to the caller.

• command

```
int command (int argc, const char * const * argv)
```

In this method, we check to see if we are asked to replace the internal queue with another object of type `PacketQueue`. If the string "'packetqueue-attach"' is placed in front of a DRED object along with an object of type PacketQueue, the old one will be removed from memory and the new one will replace it. NS expects objects derived from `Queue` to have this feature implemented although it is not necessary for our purposes.

We also needed a method to access the current queue size from Tcl scripts. We could put the proper instructions in the `command` method of the DRED

class, but we decided to put it in the `command` method of the `Queue` class, so it will be inherited to all of its subclasses such as DropTail or RED. This gives us the ability to find out the current queue size of a router during a simulation, without knowing which congestion management algorithm it is using. The `command` method of the `Queue` class is modified in a way that `length` appears as one of its procedures that returns the current queue size to its caller.

Chapter 5

Class-based DRED (CDRED)

5.1 The Design of CDRED

The CDRED algorithm is based on the DRED algorithm, but additionally tries to enforce fairness. Based on simulations, when homogeneous sources have the same network conditions, the bandwidth is shared equally amongst them under DRED. For example, when 100 TCP sources with the same RTT start to compete on a 45 Mbs link at the same time, each of them gets an approximate average bandwidth of 450 Kbs. However, when there are one or more unresponsive flows, or when sources have different RTTs, the available bandwidth will not be fairly shared. As an example, in a router with a 4.5 Mbs outbound link, a UDP source with a transmission rate of 4.5 Mbs can almost bring all of the TCP sources to a stop, while the queue size has reached its target and from DRED's perspective, there is no need to change the drop probability. As the number of multi-media streaming applications grows in the Internet, this problem could become more severe.

Since the bandwidth given to a source is proportional to the number of packets the source has in queue, an aggressive source, like a TCP source with a small RTT, can bring the queue size to its target under DRED and take all the available bandwidth. As the simulation results show, using a single

drop probability for heterogeneous sources cannot result in fairness. The algorithm should drop packets from each source or group of sources based on their aggressiveness in taking the available bandwidth.

As a network administrator, one might be interested in putting similar sources in one group (or class) and specifying the maximum amount of bandwidth each class should get along with the desired target queue size. This is actually the main objective of CDRED. The algorithm limits the bandwidth each group can take and brings the total queue size to the desired value at the same time.

CDRED takes the number of groups, n, and a user-assigned weight for each group, w_i, in addition to the total queue target T. The ratio of a group's weight to the sum of the given weights determines the maximum bandwidth the group can take (various ways of choosing the weights to accomplish different objectives are discussed in Chapter 7). Having homogeneous sources in a group, the bandwidth given to a group will be further divided equally to its members under DRED. CDRED assigns a separate target queue size for each group in a way that their sum is equal to the desired total target. Under CDRED each group has its own controller and drop probability. There are effectively n versions of the DRED algorithm running in parallel. There is still a single FIFO queue in the router, but CDRED keeps track of the number of packets of each group in the queue. By stabilizing the total queue size around its target and limiting each group to its specific share of queue, we can distribute the available bandwidth based on the given weights and have the benefits of DRED at the same time.

CDRED needs a method to recognize the class of a packet on its arrival. The way it is implemented right now is to call an external function which takes a packet and returns its class by looking at some specific field in the packet header (the Type-Of-Service (TOS) field in the IP header or flow-id field in IPv6 header are good candidates for this purpose). In other words, CDRED acts more as a back-end system where the front-end system is responsible for putting sources in different groups and detecting the group of a packet.

Regarding CDRED implementation, note that the algorithm only collects group-level information, not flow-level. As a result, there is no need to keep a list of active flows or track queue length for each flow. As long as there are a small number of groups, the memory used by CDRED will remain low and most likely the algorithm to detect the class of an arriving packet will not be complicated.

As a simple example, suppose we put the sources in two different classes: one for TCP sources and the other for UDP sources. The total available bandwidth is C and the total desired queue size is T. The bandwidth weights are w_1 and w_2 for each class. Since the link capacity is in bits (or bytes) per second, we also need average packet size for each class to determine the target queue size. Let p_1 and p_2 be the average packet size for each class. In a stabilized queue, class 1 will have T_1 packets and class 2 will have T_2 packets in the queue, where

$$T = T_1 + T_2. \tag{5.1}$$

The bandwidth given to class 1, C_1, can be calculated as the number of bytes class 1 has in queue divided by the time it takes to transfer all of the packets in queue.

$$C_1 = \frac{T_1 p_1}{\left(\frac{T_1 p_1 + T_2 p_2}{C}\right)} = \frac{T_1 p_1}{T_1 p_1 + T_2 p_2} C$$

To limit the class 1 bandwidth to its requested limit, we should have

$$\frac{T_1 p_1}{T_1 p_1 + T_2 p_2} C = \frac{w_1}{w_1 + w_2} C. \tag{5.2}$$

By using equations (5.1) and (5.2) and solving for T_1 and T_2 we have

$$T_1 = \frac{w_1 p_2}{w_1 p_2 + w_2 p_1} T, \tag{5.3}$$

$$T_2 = \frac{w_2 p_1}{w_1 p_2 + w_2 p_1} T. \tag{5.4}$$

For the general case, where there are n different classes, the algorithm should solve n equations with n unknowns $(T_1, T_2, \ldots, T_n)$. On the other hand,

if we specify our total queue target in bytes instead of packets, the problem will become much easier.

Suppose we want our queue to be filled up to M bytes. We can write

$$M = T_1 p_1 + T_2 p_2 + \cdots + T_n p_n.$$

By using the same logic, we can find the target values for each class

$$\begin{aligned} T_1 &= \frac{M}{p_1} \frac{w_1}{w_1 + w_2 + \cdots + w_n} \\ T_2 &= \frac{M}{p_2} \frac{w_2}{w_1 + w_2 + \cdots + w_n} \\ &\vdots \\ T_n &= \frac{M}{p_n} \frac{w_n}{w_1 + w_2 + \cdots + w_n}. \end{aligned} \tag{5.5}$$

The other advantage of using a target of M bytes in queue rather than T packets is that with different packet sizes for each class, it is not really clear how much buffer space the router should have. It is more convenient and practical to check the available memory of the router and set M corresponding to that (half the available memory, for example).

In order to divide the available bandwidth equally amongst all sources, we just have to set each class weight to the number of sources in the class. For example if there are 99 TCP sources in group 1 and a single UDP source in group 2, by having $w_1 = 99$ and $w_2 = 1$, each of those sources will get approximately 1/100 of the available bandwidth regardless of the value of T. This issue will be discussed further in Chapter 7.

5.2 Simulation Results

We implemented CDRED under NS and compared its performance with the DropTail, RED, and DRED algorithms.

The simulated network is the same as Figure 1.9 where two routers are connected via a 45 Mbs link. The senders and the receivers are connected

with a 100 Mbs link to the routers so the link between the two routers will be the only bottleneck. To be consistent amongst algorithms regarding their parameters, we used the version of CDRED that uses a total target size in packets. Maximum queue size is set to 1260 packets and the total target is set to 630 packets (half buffer size). The TCP segment size is set to 536 bytes but since there will be a 40 byte IP header on top of that, the total TCP packet size will become 576 bytes. All sources start at the same time and the network is simulated for 60 seconds.

Two different simulations are done. In the first one, there are TCP sources with different RTTs and in the second one TCP sources compete with one or more unresponsive UDP flows.

5.2.1 TCP Sources with Different RTTs

Three different classes were defined under CDRED. In each class, there are 50 FTP sources that always have data to send. In the first simulation, the RTT of the groups are different by a factor of 2 and 5, while in the second simulation they are different by a factor of 5 and 10. The parameters of each algorithm are given in Table 5.1. The bandwidth each source is taking is calculated by dividing the number of transmitted bytes from that source by the time it took to transmit those bytes. Then the average bandwidth and standard deviation are calculated for each class. The results are shown in Table 5.2.

Based on the results, sources with smaller RTTs get higher bandwidth under DropTail, RED, and DRED. That is basically because they receive acknowledgment packets earlier, so they can increase their window size faster. But under CDRED, the average bandwidth for each class is around 300 Kbs which is 1/150 of the 45 Mbs available. In the second simulation, when the difference between RTTs is more pronounced, the need to protect sources with long RTTs is more obvious. As shown in the results, the class with the shortest RTT gets an average of 679 Kbs under DropTail while the class with the longest RTT can get only 87 Kbs. CDRED still gives every source around

Algorithm	Parameter Name	Given Value
RED	Minimum threshold (min_{th} = 0.2 queue size)	252 packets
	Maximum threshold (max_{th} = 0.6 queue size)	756 packets
	Queue weight (w_q)	0.002
	Mean packet size	500 bytes
	Maximum drop probability (max_p)	0.1
DRED	Maximum buffer size (B)	1260 packets
	Queue size target (T)	630 packets
	No-drop threshold ($L = 0.9T$)	567 packets
	Sampling interval (ΔT)	1 ms
	Control gain (α)	5×10^{-5}
	Filter gain (β)	0.002
CDRED	Maximum buffer size	1260 packets
	Total queue size target (T)	630 packets
	Number of classes (n)	3
	Class 1 weight (w_1)	50
	Class 2 weight (w_2)	50
	Class 3 weight (w_3)	50
	Queue size target for all classes	210 packets
	No-drop threshold for all classes	189 packets
	Control gain for all classes	5×10^{-5}
	Filter gain for all classes	0.002

Table 5.1: Values used for the parameters of different algorithms for the simulation

300 Kbs of bandwidth.

5.2.2 TCP Sources Competing with Unresponsive Flows

In this simulation, one or more UDP sources with constant sending rates are competing with TCP sources on a 45 Mbs link. Two classes are defined for CDRED: one for UDP and the other one for TCP sources. The total number of TCP and UDP sources is 100. As a result, under a fair algorithm, each source should receive a bandwidth of 450 Kbs. In the first three simulations, there is one UDP source sending its fair share, twice its fare share, and 10 times its

Specification	Bandwidth (Kbs)	DropTail	RED	DRED	CDRED
Class 1 RTT: 24 ms	Class 1 avg. bandwidth	486	409	389	299
Class 2 RTT: 48 ms	Standard deviation	98	31	39	41
Class 3 RTT: 120 ms	Class 2 avg. bandwidth	289	299	316	304
	Standard deviation	92	28	46	75
	Class 3 avg. bandwidth	117	187	185	287
	Standard deviation	34	42	20	37
Class 1 RTT: 24 ms	Class 1 avg. bandwidth	679	532	504	311
Class 2 RTT: 120 ms	Standard deviation	125	52	46	35
Class 3 RTT: 240 ms	Class 2 avg. bandwidth	112	228	231	289
	Standard deviation	29	37	28	32
	Class 3 avg. bandwidth	87	136	143	279
	Standard deviation	25	24	16	41

Table 5.2: The bandwidth used by TCP sources with different RTTs under DropTail, RED, DRED, and CDRED

fair share respectively. In the last simulation, there are 10 UDP sources, each sending with a constant rate of 4500 Kbs, so their total sending rate will be equal to the total available bandwidth. Table 5.3 shows the average bandwidth each group received during the simulation.

Based on Table 5.3, the UDP sources get almost the same bandwidth as their sending rates under DropTail, RED, and DRED. When they send more than their fair share, they actually limit the TCP sources. For example, in the last simulation the UDP sources have almost monopolized the bandwidth. On the other hand, CDRED protects the TCP sources by limiting the UDP sources to their calculated queue share. Note that the numbers shown in Table 5.3 are average bandwidths. Since it takes some time for CDRED to initialize and stabilize the queue around the calculated target, the averages are a little bit higher than what is expected (450 Kbs). Figure 5.1 shows the queue size for the UDP sources. For the first 10 seconds, the UDP sources are getting more than their fair share but after the queue size is stabilized, in the next 50 seconds, the bandwidth they receive is around 450 Kbs.

Specification	Bandwidth (Kbs)	DropTail	RED	DRED	CDRED
No. of UDP: 1	Avg. TCP bandwidth	438	439	438	438
No. of TCP: 99	Standard deviation	63	30	44	42
UDP rate: 450 Kbs	Avg. UDP bandwidth	430	443	445	447
	Standard deviation	0	0	0	0
No. of UDP: 1	Avg. TCP bandwidth	433	435	433	436
No. of TCP: 99	Standard deviation	56	39	43	39
UDP rate: 900 Kbs	Avg. UDP bandwidth	872	889	889	612
	Standard deviation	0	0	0	0
No. of UDP: 1	Avg. TCP bandwidth	399	401	399	435
No. of TCP: 99	Standard deviation	53	32	37	41
UDP rate: 4500 Kbs	Avg. UDP bandwidth	4152	4129	4446	836
	Standard deviation	0	0	0	0
No. of UDP: 10	Avg. TCP bandwidth	77	49	52	423
No. of TCP: 90	Standard deviation	12	12	13	36
UDP rate: 4500 Kbs	Avg. UDP bandwidth	3811	4068	4056	701
	Standard deviation	932	3	74	43

Table 5.3: Average TCP and UDP bandwidths under different algorithms when there are one or more unresponsive flows

5.3 Dynamic Change of Targets

There is a special case when we have unresponsive flows in one group competing with responsive flows in another group. After assigning targets for each class based on the given weights, we may consider changing the targets dynamically if the unresponsive flows do not utilize their allocated bandwidth.

Suppose we have defined two classes for CDRED: class 1 for TCP and class 2 for UDP sources. Based on the given weights and the total queue target in bytes, M, CDRED calculates T_1 and T_2 for each class respectively so that

$$T_1 p_1 + T_2 p_2 = M.$$

If the sending rate of the the UDP sources turns out to be less than their allocated bandwidth, their average queue size becomes T_2' where $T_2' < T_2$ and the total queue size will drop to M' where

$$M' = T_1 p_1 + T_2' p_2. \tag{5.6}$$

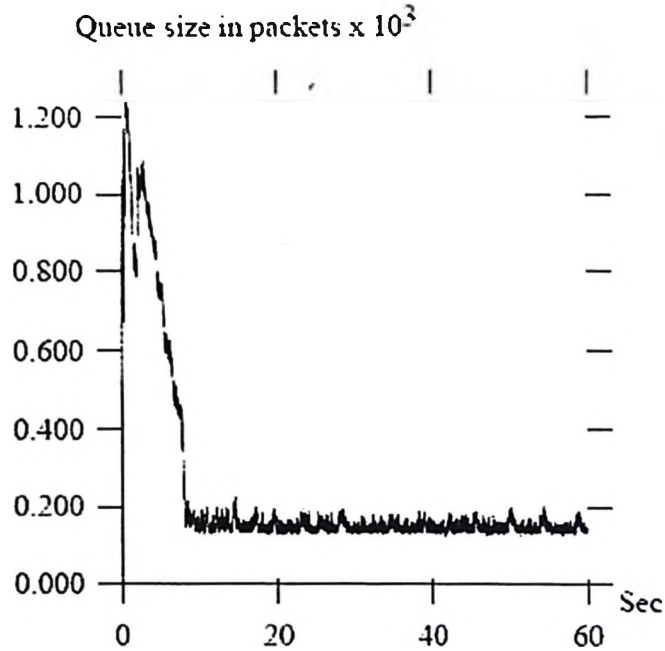


Figure 5.1: The queue size for UDP sources when sending 10 times their fair share under CDRED.

Since $M' < M$, we have

$$\frac{T_1 p_1}{M'} > \frac{T_1 p_2}{M},$$

which means that the TCP sources will automatically receive more bandwidth. Now if the queue delay is our only concern, there will be no need to change the targets as all sources will see a shorter delay when the number of bytes in the queue decreases; however, if we are concerned about delay jitter, we may consider changing the targets in a way that the unresponsive flows get a bandwidth equal to their sending rates and still the queue size reaches its desired target M.

The actual bandwidth UDP sources are using is

$$\frac{T_2' p_2}{M'} C. \tag{5.7}$$

We want to calculate new values for each class target (T_{1new}, T_{2new}) so that

$$T_{1new} p_1 + T_{2new} p_2 = M. \tag{5.8}$$

without limiting the UDP sources. The bandwidth the UDP sources will receive after recalculating targets will be

$$\frac{T_{2new} p_2}{M} C. \tag{5.9}$$

Equating (5.7) and (5.9) results in

$$T_{2new} = \frac{M}{M'} T_2'. \tag{5.10}$$

and consequently by using (5.6) and (5.8), we get

$$T_{1new} = \frac{M}{M'} T_1.$$

There are two issues one should be aware of when dynamically changing targets. First, by trying to keep the queue size around M bytes instead of T packets, the delay seen by sources is fixed while as when there are different packet sizes in queue, by bringing the total number of packets to T, there is no guarantee that there are the same number of bytes in queue.

Second, since responsive flows try to adjust their sending rates based on the drop probability, decreasing or increasing their target based on their bandwidth usage does not seem reasonable. In this thesis, we have assumed that the responsive flows always have some data to send (like FTP connections), so they can always use allocated bandwidth. In case there are other responsive flows with varying bandwidth demand (like TELNET connections), by decreasing their target and giving them less than their fair share of bandwidth, they will send with a slower rate so there is no way to tell if they need more bandwidth in the future.

To implement this idea, two more parameters should be given to CDRED. The first one is the condition on when to reassign targets and the second one is the frequency (or the period) to check the condition.

As for when to reassign targets, one can define $c_{thresh} < 1$ and change the targets when $T_2' < c_{thresh} T_2$. Since we want to keep the delay fixed, a better choice is to change targets when $M' < c_{thresh} M$. Yet another alternative is to change targets when

$$\frac{M - M'}{C} > \text{some time constant.}$$

This is a more natural and convenient way for a user to specify how much change in delay is acceptable. For example, we may only want to reassign the targets when the change in delay is greater than 100 ms.

For the frequency at which to check whether to recalculate targets, the user can define some time interval over which CDRED calculates the average queue size. At the end of each interval, it then decides if it should change the targets. It is better to recalculate the average queue size in each period so that it provides a better estimation of the current queue size.

Later, when UDP sources increase their sending rates, we can change the targets again to let them use their fair share. To detect this situation, one way is to look at the UDP drop probability. For example, if the average UDP drop probability is 0.3 during the user-defined interval, it means that the allocated bandwidth is 0.7 of the rate at which UDP sources are sending. By changing the UDP target from T_{2new} to $min(T_2, T_{2new}/0.7)$, we can make the situation fair again. Figure 5.2 shows the UDP drop probability during the second simulation when they were sending 10 times their fair share. After stabilizing, the drop probability is around 0.9.

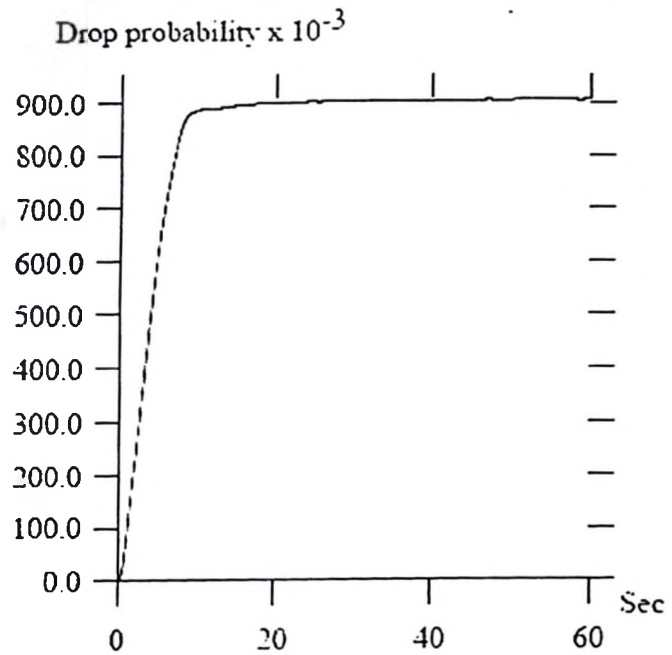


Figure 5.2: CDRED drop probability for UDP sources when sending 10 times their fair share.

Chapter 6

Implementation of CDRED

This chapter explains CDRED implementation under NS. There are a couple of implementation ideas, such as defining a new timer class to update the drop probability, that are the same as the DRED implementation. The following sections describe the classes defined for CDRED implementation along with their methods and member variables.

6.1 Class DREDSettings

As was discussed in Chapter 5, the CDRED algorithm uses several instances of DRED to control the queue size for each class or group of connections. Each of these classes may have their own DRED settings. The `DREDSettings` class keeps DRED parameters for each group or class of connections.

6.1.1 Member Variables

Member variables in this class are essentially the same as those defined in the DRED class (shown in Table 4.1) but with the following differences:

- Variables added

classID : Each class under CDRED has its own class ID, which is an integer greater than or equal to zero. This ID is stored in the `classID` variable.

queueLength : This variable keeps the number of packets for each class stored in the queue.

- Variables removed

q_ : As was explained in Chapter 5, CDRED has one internal queue which is shared amongst all classes, with processing based on a first-come-first-served policy. Therefore, there is no need for each class to have its own queue.

deltaT : Under CDRED, the calculation of drop probability is done at the same time for all classes. As a result, there is only one global sampling interval (`deltaT`) which is defined in the CDRED class. There are not individual sampling intervals for each class.

myTimer : The same argument as for `deltaT` holds for `myTimer`. There is only one timer of type `CDREDTimerClass` defined in the CDRED class [1].

error : This variable is no longer a member variable. It is defined as a local variable in the `updatePDrop` method.

errorFile : This variable was mainly defined in DRED to output the difference between the queue size and the target so we could see the effect of different choices of parameters for the PID controller. In CDRED there is no need to define this variable as a member variable.

- Modified variables

[1] Even though the current version of CDRED only has one timer, this can easily be changed in future versions in case each class needs its own sampling interval.

T and L : In DRED, these two variables are integers but in CDRED they are real numbers. That is because when calculating targets for each class, the result may not always be an integer.

6.1.2 Methods

This class does not have any functionality other than keeping DRED parameters. As a result, there is only one method, `printAll`, which is mainly used for documenting simulations and debugging purposes.

- **printAll**

`void printAll()`

This method prints out the values for the following variables to the standard output: `classID`, `B`, `T`, `L`, `alpha`, `propGain`, `diffGain`, `beta`, `pDrop`, `normalizeOption`, `theta`, `trace`, and `queueLength`.

6.2 Class CDREDTimerClass

This class is a subclass of `TimerHandler` and is very similar to DREDTimerClass, described in Section 4.1. The only difference is that its constructor takes a pointer to a CDRED object rather than a DRED object.

6.2.1 Member Variables

There is only one member variable, `p_`, which is a pointer to a CDRED object. This pointer will be initialized when the constructor of the class is called. By use of this pointer, methods of this class can call the public methods of a CDRED object.

6.2.2 Methods

• **The Constructor**

`CDREDTimerClass (CDRED *p)`

The constructor of this class, receives a pointer to a CDRED object as its argument and stores it in `p_`. It also calls the constructor of its parent class: `TimerHandler`.

• **resched**

`void resched(double delay)`

This method is inherited from `TimerHandler` and makes the timer expire after `delay` seconds. It is used to schedule the next time we update the drop probabilities for all of the groups (classes) that are defined under CDRED.

• **expire**

`virtual void expire(Event* e)`

This method is inherited from `TimerHandler` and will be called when the timer has expired. Since it is a virtual method, it has to be defined in any subclass of `TimerHandler`. In our implementation, this method calls the `updatePDrop` method of the CDRED object that `p_` is pointing to.

6.3 Class CDREDClass

This class is a subclass of `TclClass` and is defined to map the CDRED class in the interpreted hierarchy (`Queue\CDRED`) to its corresponding class in the compiled hierarchy (`CDRED`).

6.3.1 Member Variables

This class does not have any member variables.

6.3.2 Methods

• **The Constructor**
`CDREDClass()`

The `CDREDClass` constructor calls the `TclClass` constructor and passes the string "`Queue/CDRED`" as its argument. As was discussed in Section 2.3, this will make NS call the `create` method of `CDREDClass` when a user instantiates an object of type `Queue/CDRED` in a simulation script.

• **create**
`TclObject *create(int argc, const char * const *argv)`

The `create` method receives two arguments and returns an object of type `TclObject` to its caller. It is invoked when a user wants to create an object of type `Queue/CDRED` in a simulation script. If the user has included any arguments in the script, those arguments will also be passed to the CDRED constructor; otherwise, the `create` method will call the CDRED constructor without any arguments. In both cases, a new object of class `CDRED` is returned as a `TclObject`.

6.4 Class CDRED

This class is a subclass of the `Queue` class and it is the main class implementing the CDRED algorithm. Since CDRED is actually several instances of DRED running simultaneously, the class `DREDSettings` is used to keep the separate DRED parameters.

6.4.1 Member Variables

There are two sets of member variables defined in the CDRED class. The first group is simple variables which are common amongst all connection classes while the second group holds specific information for each class. Table 6.1

lists the member variables defined in the class. Here is the description for each of them:

- `PacketQueue` is the internal queue which holds packets from all classes of connections (or sources).
- `totalB` and `totalT` represent the total buffer space available in the router and the total target queue size, respectively.
- `noOfClasses` keeps the number of classes that the user wants to have under CDRED. On the other hand, `classWeights` is an array of length `noOfClasses` which holds the weight associated with each class. If we want to have an equal share of bandwidth amongst all flows, each class should have a weight equal to the number of flows existing in the class. More discussion about class weight is provided in Chapter 7. The number of classes and the class weights should be supplied by the NS user in the simulation script when using the CDRED algorithm.
- `classSettings` is an array of type `DREDSettings`. It has `noOfClasses` elements and it keeps the DRED settings for each class. There are default values for each of the DRED parameters that are assigned to them in the constructor method. Later, the user can access each individual DRED parameter for each class through the simulation script.
- The drop probability update interval is stored in `deltaT`.
- The `myTimer` variable, an object of type `CDREDTimerClass`, is defined to schedule the next update time.
- If the value for the `trace` variable is one, CDRED will record the total queue size every `deltaT` seconds to the file specified by the `qSizeFile` pointer.
- The `debug` variable is used if the programmer needs to monitor more variables.

- As in DRED, the `firstPacketArrived` variable serves as an optimization variable and will be set by CDRED when the router receives the first packet. The drop probability calculation and its periodic updating starts after this variable is set.

Variable name	Description
Protected	
q_	internal queue
totalB(*)	total buffer size in packets
totalT(*)	total target queue size in packets
defAlpha(*)	default control gain for the integral controller
defPropGain(*)	default control gain for the proportional controller
defDiffGain(*)	default control gain for the differential controller
defBeta(*)	default filter gain for the low pass filter
defTheta(*)	default value for theta
noOfClasses	number of DRED classes
classWeights	an array of integers, holding the weight associated with each class
weightsSum	the sum of weights in all classes
classSettings	an array of type `DREDSettings`, holding the DRED parameters for each class
trace(*)	shows if we want to trace the total queue size
qSizeFile	if trace = 1, the total queue size will be recorded every deltaT seconds to the file pointed to by this variable
deltaT(*)	sampling interval in seconds for all classes
myTimer	timer used to update pDrop for all classes
firstPacketArrived	determines if the router has received the first packet
Private	
debug(*)	if it holds 1, the code will run in debug mode

Table 6.1: CDRED Member Variables

6.4.2 Methods

• **The Constructor**

`CDRED(int = 0, const char * const * = NULL)`

The CDRED constructor takes two arguments. The first one holds the number of arguments and the second one is an array holding the arguments themselves. There are default values defined for each of these arguments (zero for the first argument and NULL for the second one). When the constructor is called without any arguments, default values will be used.

It is obvious that CDRED should act exactly like DRED if there is only one class of connection. In this case, no matter what the weight of the class, the result should be the same. By calling CDRED without any arguments, the constructor will assume that there is only one class of connection and will set its class weight to one. The sum of weights (`weightsSum`) will also be set to one.

When the user needs to have more than one class, CDRED arguments should be passed in the following format:

`<number of classes> <first class weight> <second class weight>...`

CDRED will parse the arguments and set its internal variables as follows:

- number of classes will be stored in `noOfClasses`
- the array `classWeights` will be dynamically created and each of its elements will store the weight for each class. So the n_{th} element in the array will be the weight for the n_{th} class
- `weightsSum` will be calculated to keep the sum of class weights

After parsing the arguments, CDRED will perform the following tasks:

- variables in Table 6.1 that are marked with (*) will be bound to variables with the same name in the interpreted hierarchy. This will also make those bound variables get their default values specified in the `ns-default.tcl` file

- the array `classSettings` will be dynamically created based on the number of connection classes
- each element of `classSettings`, which is an object of type `DREDSettings`, will be assigned an ID, starting from zero. This ID is stored in `classID`
- all connection classes defined under CDRED will get the same value for `B` (`totalB`). We do not divide the total buffer space between classes. As long as there is room in the buffer and the drop probability of the class allows, an arriving packet will be accepted
- the target queue size for class i, which is stored in `classSettings[i].T`, is a real number that is calculated as

$$\texttt{classSettings[i].T} = \texttt{totalT} * \frac{\texttt{classWeights[i]}}{\texttt{weightsSum}}$$

- the `L` parameter is assigned 0.9 of the calculated `T`
- parameters `alpha`, `propGain`, `diffGain`, `beta`, and `theta` will get their default values from `defAlpha`, `defPropGain`, `defDiffGain`, `defBeta`, and `defTheta`, respectively
- variables `pDrop`, `filteredError`, `prevFilteredError`, `integralOutput`, `prevIntegralOutput`, and `firstPacketArrived` will be initialized to zero
- `normalizeOption` will get one, so we normalize the error by $2T$
- for each class of connections, two files will be created in the local directory to record its queue size and drop probability. The file names are `dred-queue-<i>` and `dred-drop-prob-<i>` where `<i>` will be replaced by the class ID. The reference to these files will be stored in file pointers `DREDSettings[i].qSizeFile` and `DREDSettings[i].dropProbFile`, respectively

- a file with the name `cdred-queue` will be created in the local directory to record the total queue size. The reference for this file will be stored in `qSizeFile`

• **updatePDrop**

```
void updatePDrop()
```

This method is very similar to the `updatePDrop` method in the DRED class and performs the same tasks on every connection class defined under CDRED. The only difference is that for each class, its corresponding variables in the `classSettings` array will be used and updated. The only variables which are used for all classes are `deltaT` and `debug`. In the beginning of the method, the timer will be set to expire after `deltaT` seconds. If `debug` is set, after updating the drop probability for each class, selected variables of that class will be printed to the standard output.

Each class has its own `trace` variable (`classSettings[i].trace`) which will determine if the program should output the queue size and drop probability for that class. This variable can be changed by the user in a simulation script by the use of the `command` method, which will be described later in this section. There is also another `trace` variable which will determine if we want the total queue size to be reported. Since this variable is amongst bound variables, the user also has control over the value of this variable through the simulation script. The code in the `updatePDrop` method will check the values of all these variables and print out the desired variables to their corresponding files.

• **getPacketClass**

```
int getPacketClass(Packet *p)
```

This method receives a packet as its argument and returns back an integer which determines which class the packet belongs to. The method is used when we want to update the queue size of each class in `enque` and `deque` methods.

A programmer may have to change the code of this method based on the application CDRED is used for. For example if we divide sources into two classes: one for TCP and one for UDP sources, `getPacketClass` should look into a specific field in the IP header to find out if the received packet is a TCP packet or a UDP one. On the other hand, if all sources are TCP and we divide them based on the bandwidth we are willing to give them, then the code may have to look at some other field in the packet header such as Type-of-Service(TOS) to find out the class that the arrived packet belongs to.

- **enque**

```
void enque(Packet *)
```

When a new packet arrives, NS will call this method and pass the arriving packet as its argument. Tasks done in this method can be listed as follows:

- the total queue size will be compared to the queue limit (`qlim_`) and if the current queue size is greater than or equal to `qlim_`, the arriving packet will be dropped
- if there is still room left in the queue, `getPacketClass` will be called to return the class to which the packet belongs
- the integer returned from `getPacketClass` will be used as an index for the `classSettings` array to retrieve parameters `L`, `queueLength`, and `pDrop`
- if the queue size of the class is less than its no-drop threshold (`L`), the arrived packet will be accepted by calling the `enque` method of the internal queue (`q_`), otherwise its drop probability will determine if we should drop the arrived packet
- in case the packet is accepted, the queue length of its class will be incremented by one

- the variable `firstPacketArrived` will be checked. If this is the first packet arriving at the router, `firstPacketArrived` will be set to one and the timer will be set to expire after `deltaT` seconds

• **deque**

```
Packet *deque()
```

This method is called when NS is ready to transmit the next packet in the queue. It calls the `deque` method of the internal queue (`q_`) and stores the return value in `p`. If `p` is equal to `NULL`, it means that there are no packets left in the queue. Otherwise, the class to which the packet belongs will be determined by calling `getPacketClass` and its corresponding queue size will be decremented by one. Finally, the method returns `p` to its caller.

• **command**

```
int command (int argc, const char * const * argv)
```

As was discussed in Section 2.2.2, by the use of the `command` method, we can provide virtual methods for an object in the interpreted hierarchy. We used this feature and defined the virtual method `getValue` to get access to DRED settings of each connection class. The syntax to call this virtual method is:

```
<CDRED object> getValue <parameter name> <class number>
```

As an example, if we want to know the target value (`T`) for the second class (class number 1), we can put the following line in our simulation script

```
$myCDREDObj getValue T 1
```

where `$myCDREDObj` is an object of type CDRED. The parameters that can be accessed this way are: `classID`, `B`, `T`, `L`, `alpha`, `propGain`, `diffGain`, `beta`, `theta`, and `normalizeOption`.

We have also defined `trace` as a virtual method to modify the value of the `trace` parameter for each connection class. Here is how to call it:

```
$myCDREDObj trace <class number> <value>
```

where `value` is the new value (either zero or one) that will be assigned to the `trace` parameter of the class with its `classID` equal to `class number`.

The other virtual method defined is `closeAllFiles` which will call a method with the same name from the CDRED class. CDRED users should call this method at the end of their simulation to close all files which are used to record queue sizes and drop probabilities.

As NS sometimes needs to change the internal queue corresponding to the active queue management, we also defined `packetqueue-attach` as a virtual method. It will delete the current queue from memory and instead use the queue object passed as its argument.

When parsing the arguments for the above virtual methods, if any of them is invalid (for example an out of range class number for the second argument of `getValue`), the `command` method will return `TCL_ERROR`, otherwise it will return `TCL_OK`.

If the `command` method cannot match its arguments with any of the above cases, it will pass them to the `command` method of its parent class, `Queue`. That is where the virtual method `length` is defined, for example.

- **printAll**

```
void printAll()
```

This method is used for documenting the simulation and also for debugging purposes. All variables listed in Table 6.1, other than `q_`, `classWeights`, `classSettings`, `qSizeFile`, and `myTimer`, will be printed to the standard output. The method then calls the `printAll` method for each connection class defined under CDRED which will make them print their own settings.

- **closeAllFiles**

`void closeAllFiles()`

This method will close all files that are opened in the constructor to record simulation data. A user can call this method implicitly from his simulation script (by use of the `command` method). This method is also called from the CDRED destructor. The file pointers which are closed are `qSizeFile` from the CDRED class and `qSizeFile` and `dropProbFile` from each connection class defined under CDRED.

- **The Destructor**

`~CDRED()`

The destructor removes the internal queue (`q_`) from memory and deletes the two arrays `classWeights` and `classSettings` that were allocated dynamically. It also closes all open files by calling the `closeAllFiles` method.

Chapter 7

CDRED Applications and Future Work

Based on the application CDRED is used for and the way we want to share the bandwidth amongst flows (or sources), one can assign different weights for each class. As shown in Section 6.4.2, to calculate the queue target for each class, CDRED divides the corresponding class weight by the sum of the weights and then multiplies that by the total queue target. If each class has a different average packet size, the proportion of bandwidth (determined by user weights $w_1, w_2, \cdots, w_n$) that the user wishes to give to each class must be translated to CDRED weights. In Section 7.1.4 we will see that by using the total number of bytes in the buffer as the target, this conversion is not necessary.

This chapter has two sections. In the first section we describe some of the more common situations for which CDRED can be used. The second section includes a brief conclusion and discusses potential improvements to CDRED as future work.

7.1 CDRED Applications

7.1.1 Fairness at flow level

In this scenario, we want to divide the available bandwidth equally amongst all flows (in different classes). If we assume there are n different classes defined under CDRED and there are f_i flows in the i_{th} class, then each flow should get a bandwidth of $\frac{C}{f_1+f_2+\cdots+f_n}$, where C is the bandwidth of the outgoing link connected to the router. In the case that all classes have the same average packet size P, then the following equation should hold between the bandwidth class i is receiving and the bandwidth we are willing to give to class i.

$$\frac{T_i P}{(TP)/C} = \frac{f_i}{f_1 + f_2 + \cdots + f_n} C.$$

where T_i is the target queue size for class i and $T = T_1 + T_2 + \cdots + T_n$. By solving the above equation for T_i, we get

$$T_i = \frac{f_i}{f_1 + f_2 + \cdots + f_n} T. \tag{7.1}$$

which shows that if we let the weight of each class be equal to the number of flows in that class, then all flows passing through the router will get an equal share of the available bandwidth.

If each class has its own average packet size, the differences must be taken into account. Suppose we have n classes, each with their own average packet size P_i. In order to have equal share amongst all flows we should have

$$\frac{T_i P_i}{(T_1 P_1 + T_2 P_2 + \cdots + T_n P_n)/C} = \frac{f_i}{f_1 + f_2 + \cdots + f_n} C,$$

for $1 \leq i \leq n$. By using the above equations and the fact that $T = T_1 + T_2 + \cdots + T_n$, we get

$$T_i = \frac{f_i/P_i}{f_1/P_1 + f_2/P_2 + \cdots + f_n/P_n} T. \tag{7.2}$$

In this case, the weight for class i should be f_i/P_i. Note that when we set $P_1 = P_2 = \cdots = P_n = P$, (7.2) reduces to (7.1).

7.1.2 Fairness at class level

In this scenario, we want to give each class an equal share of bandwidth regardless of the number of flows in the class. An example for this case can be when the router has three input links from three different Internet Service Providers (ISP), and it wants to divide the bandwidth equally amongst them.

If all classes have the same average packet size P, by giving the same weight, w, to all classes, the bandwidth will be equally divided amongst them, i.e.

$$\frac{T_i P}{(TP)/C} = \frac{w}{\underbrace{w + w + \cdots + w}_{n \text{ terms}}} C$$

which will result in

$$T_i = \frac{1}{n} T.$$

In the case when each class has a different average packet size, the weights we should supply to the CDRED algorithm will be different. Suppose we have n classes, each with their own average packet size. Since we want to divide the bandwidth equally amongst classes, the weights the user supplies for all classes should be the same. As a result, for class i we should have

$$\frac{T_i P_i}{(T_1 P_1 + T_2 P_2 + \cdots + T_n P_n)/C} = \frac{w}{\underbrace{w + w + \cdots + w}_{n \text{ terms}}} C,$$

which will result in

$$T_i = \frac{1/P_i}{1/P_1 + 1/P_2 + \cdots + 1/P_n} T. \tag{7.3}$$

This shows that the weight parameter passed to the CDRED algorithm should be $1/P_i$ for class i.

Comparing (7.2) and (7.3), it can be interpreted that when we want to be fair amongst classes, we simply consider the whole class as a single flow (i.e. $f_1 = f_2 = \cdots = f_n = 1$).

Even though in this kind of application we do not worry about the number of flows in each class, one should be cautious when dealing with classes of

TCP flows. If there are too many flows in the class so that each of them gets an average of less than four packets in the queue, then if a packet is lost downstream, the sender will not be able to receive the three duplicate ACKs necessary to trigger the fast retransmit algorithm. On the other hand, TCP will have poor performance if its window size is not larger than four.

7.1.3 Assigning a specific bandwidth share to each class

Being fair amongst classes may not always be the ultimate goal. In some cases, the network administrator may want to assign a higher bandwidth to a specific class and a lower bandwidth to the other classes. This is a more general case than the previous ones. For example the administrator may want to assign 70% of the bandwidth to TCP flows and 30% to UDP and other flows. In this situation, there are n different user weights $(w_1, w_2, \cdots, w_n)$ for the n classes defined under CDRED.

If the average packet size is the same for all classes, the calculation will be easier and for class i we should have

$$\frac{T_i P}{(TP)/C} = \frac{w_i}{w_1 + w_2 + \cdots + w_n} C.$$

which gives

$$T_i = \frac{w_i}{w_1 + w_2 + \cdots + w_n} T.$$

This shows that the weight passed to CDRED for class i should be w_i.

On the other hand, if each class has its own average packet size, first we have to calculate the weights we should pass to CDRED. Assuming there are n classes, for class i we have

$$\frac{T_i P_i}{(T_1 P_1 + T_2 P_2 + \cdots + T_n P_n)/C} = \frac{w_i}{w_1 + w_2 + \cdots + w_n} C$$

where $1 \leq i \leq n$. By solving the above equations and using $T_1 + T_2 + \cdots + T_n = T$ we get

$$T_i = \frac{w_i / P_i}{w_1/P_1 + w_2/P_2 + \cdots + w_n/P_n} T.$$

This shows that CDRED should get w_i/P_i for class i as its weight.

7.1.4 Using the number of bytes in the buffer as the target

As shown in the previous sections, if the average packet size is different amongst classes, we have to first solve n equations with n unknowns to calculate the real weight each class should get when assigning targets. If we set the target to be the number of bytes in the buffer instead of the number of packets, the problem will become much easier.

Assume that we are asked to keep the number of bytes in the buffer at M bytes. In other words,

$$M = T_1P_1 + T_2P_2 + \cdots + T_nP_n,$$

where as a reminder, n is the number of classes. Using this technique, when we want to divide bandwidth equally amongst all flows, the target for class i will be

$$T_i = \frac{f_i}{f_1 + f_2 + \cdots + f_n}\frac{M}{P_i}.$$

When dividing bandwidth equally amongst classes, we will have

$$T_i = \frac{1}{n}\frac{M}{P_i}.$$

Finally, for assigning a specific bandwidth share to a class, we set its target based on

$$T_i = \frac{w_i}{w_1 + w_2 + \cdots + w_n}\frac{M}{P_i}. \tag{7.4}$$

As we can see, the weights that should be passed to CDRED are the same as those that the user intended. Only the constructor has to be changed to include the $\frac{M}{P_i}$ part when calculating the target for class i. We wrote another version of CDRED, called CDRED2, which takes its total target in bytes and calculates each class target based on (7.4).

7.2 Conclusion and Future Work

CDRED, as an active queue management algorithm, was designed to stabilize the queue size in routers and to try also to be fair amongst flows. A user can group flows into classes and decide how the bandwidth should be divided amongst the classes based on the weights he supplies to the algorithm. Since DRED is used to manage network congestion for each class, CDRED can successfully stabilize the queue size for each class, regardless of the network load. We also improved the DRED algorithm by including a proportional and differential controller in addition to the original integral controller, which results in smaller queue size variance. On the other hand, assigning targets for each class based on their weights and packet sizes solves the fairness problem that most other algorithms have when dealing with unresponsive flows or sources with different RTTs.

Despite its benefits and features, there are a couple of aspects of CDRED that can be improved. The first one is that we need some method to identify the class of an arriving packet. This can be as easy as identifying the input link the packet arrived from or as complicated as looking to some specific field of the packet header or doing some calculation on every packet.

The second limitation of CDRED is that we need to know the number of active flows if we want to be fair amongst all flows. We also need to know the average packet size for each class. Since CDRED is acting as a back-end system, the hope is that a higher level protocol could provide such data.

The other assumption we have made through this thesis is that our sources always have some data to send. The behavior of CDRED when there are short-lived connections, like HTTP, or on-off connections, like TELNET, needs to be studied.

Bibliography

[1] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP: Design, Implementation, and Internals*, volume II. Prentice Hall, Third edition, 1995.

[2] W.R. Stevens. *TCP/IP Illustrated. Volume 1: The Protocols*, Addison-Wesley, Reading, MA, USA, 1994. .

[3] K.K. Ramakrishnan, S. Floyd, and D. Black. *RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP*, 2001.

[4] M. Allman, V. Paxson, and W. Stevens. *RFC 2581: TCP Congestion Control*, 1999.

[5] A. Mankin. *Random Drop Congestion Control*, Proc. of ACM SIGCOMM, pp. 1-7, 1990.

[6] E. Hashem. *Analysis of random drop for gateway congestion control*, Tech. Rep. LCS TR-465, Laboratory for Computer Science, MIT, 1989.

[7] S. Floyd and V. Jacobson. *Random early detection gateways for congestion avoidance*, IEEE/ACM Transactions on Networking, vol. 1, No. 4, pp. 397–413, 1993.

[8] L. Zhang. *A New Architecture for Packet Switching Network Protocols*, MIT/LCS/TR-455, Laboratory for Computer Science, Massachusetts Institute of Technology, 1989.

[9] W. Feng, D.D. Kandlur, D. Saha, and K. Shin. *Blue: A New Class of Active Queue Management Algorithms.* Technical Report CSE-TR-387-99, University of Michigan, 1999.

[10] S. Athuraliya, V.H. Li, S.H. Low, and Q. Yin. *REM: Active Queue Management.* IEEE Network, 2001.

[11] B. Wydrowski and M. Zukerman. *GREEN: An Active Queue Management Algorithm for a Self Managed Internet.* Proceedings of ICC 2002, New York, vol. 4, pp. 2368-2372, 2002.

[12] I. Stocia, S. Shenker, and H. Zhang. *Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks,* Technical Report CMU-SC-98-136, Carnegie Mellon University, 1998.

[13] M. Parris, K. Jeffay, F.D. Smith. *Lightweight Active Router-Queue Management for Multimedia Networking.* ACM/SPIE Multimedia Computing and Networking, pp. 162-174, 1999.

[14] D. Lin and R. Morris. *Dynamics of Random Early Detection.* ACM Computer Communication Review, vol. 27, No. 4, pp. 127–137, 1997.

[15] J. Aweya, M. Ouellette, D.Y. Montuno. *A control theoretic approach to active queue management.* Computer Networks 36, pp. 203-235, 2001.

[16] R. Morris. *Scalable TCP congestion control.* Proc. IEEE INFOCOM, pp. 1176-1183, 2000.

[17] C. Zhu, O. Yang, J. Aweya, M. Ouellette, D.Y. Montuno. *A Comparison of Active Queue management Algorithms Using OPNET Modeler.* Proceedings of OPNETWORK 2001, Washington DC, 2001.

[18] M. Kwon and S. Fahmy. *A comparison of load-based and queue-based active queue management algorithms.* in Proc. SPIE ITCom, vol. 4866, 2002.

[19] K. Fall and K. Varadhan. *The ns Manual*. The VINT Project, 2003.

[20] *The Network Simulator - ns-2*. http://www.isi.edu/nsnam/ns/index.html, 2004.

621 55