Adaptive RED with Dynamic Threshold Adjustment

Submitted By

Md. Manzoor Murshed

A research report submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Department: Electrical and Computer Engineering
Major: Computer Engineering

Major Professor: Dr. Ahmed E. Kamal

Committee Members:

Dr. Manimaran Govindarasu
Dr. Wallapak Tavanapong

Iowa State University
Ames, Iowa
2005
### Table of Contents:

**Chapter 1: Introduction**
- Passive queue management ......................................................... 2
- Problems with Passive Queue Management ..................................... 2
- Active queue management ............................................................. 3
- Random Early Detection ................................................................. 3
  - Selection of Maximum Packet Drop Probability, $\text{Max}_{\text{drop}}$ .......... 7
  - Selection of $\text{Min}_{\text{th}}$ and $\text{Max}_{\text{th}}$ ................................................. 7
  - Selection of Average queue length and weight parameter $w$ ............. 7
- Modified Average queue calculation: .............................................. 8
- Classifying the RED Variants ......................................................... 9
- Adaptive RED .............................................................................. 10
- Research Contributions ................................................................ 10
- Report Outline ............................................................................ 11

**Chapter 2: Analytical Model of ARDTA**
- Model to calculate the thresholds: ................................................ 15
- Minimum and Maximum Thresholds calculation ............................ 14
- Modified average queue during idle time ..................................... 15
- Summary: ..................................................................................... 15

**Chapter 3: Simulation results and Analysis**
- Simulation Model .......................................................................... 16
- Simulation with Poisson Arrival Process ........................................ 16
  - Simulation with $\text{Min}_{\text{th}}=3$ and $\text{Max}_{\text{th}}=9$ (SIM$_1$) .................... 17
  - Simulation with $\text{Min}_{\text{th}}=4$ and $\text{Max}_{\text{th}}=12$ (SIM$_2$) ...................... 18
  - Simulation of ARDTA with Poisson arrival process (SIM$_3$) .......... 18
  - Summary of the Simulation for Poisson Process: .......................... 19
- Simulation with Self similar traffic ................................................. 20
  - Simulation of Self similar traffic with $\text{Min}_{\text{th}}=3$ and $\text{Max}_{\text{th}}=9$ (SIM$_4$) 20
  - Simulation of Self similar traffic with $\text{Min}_{\text{th}}=4$ and $\text{Max}_{\text{th}}=12$ (SIM$_5$) 21
  - Simulation of ARDTA with Self similar traffic (SIM$_6$) .................. 22
  - Summary of the Simulation with Self similar traffic ..................... 23
- Simulation with FTP traffic using TCP (FTP_TCP) ......................... 23
- Comparison of ARDTA with RED and adaptive RED ..................... 24
- Comparison of ARDTA with ARED$_{Floyd}$ and ARED$_{Feng}$ ............. 26
- Summary .................................................................................... 28

**Chapter 4: Conclusions and Future Research**
- Conclusions ................................................................................. 29
- Future Research .......................................................................... 30

**Appendix: A** ............................................................................... 31
**Appendix B** ............................................................................... 35
**References** ................................................................................ 37
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>General algorithm for RED gateways</td>
<td>4</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>RED gateway drop function</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Detailed algorithm for RED gateways</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Modified algorithm to calculate the average queue</td>
<td>8</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>C++ code to calculate the average queue during idle time</td>
<td>9</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>Our algorithm (ARDTA)</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Model to calculate the threshold</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Topology used for the Simulation</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Total throughput for SIM₁</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Average queue for SIM₁</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Total throughput for the SIM₂ experiment</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Average queue for the SIM₂ ex.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Total throughput for the SIM₃ experiment</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Average queue for the SIM₃ experiment</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Total throughput for the SIM₄ experiment</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Average queue for SIM₄ Experiment</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Total throughput for the SIM₅ experiment</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Average queue for the SIM₅ experiment</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Total throughput for the SIM₆ experiment</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Average queue for the SIM₆ experiment</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Topology for the simulation of section 3.4</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Average queue for RED with and Increase in Congestion</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>Average queue for Adaptive RED with and Increase in Congestion</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>Average queue for ARDTA with and Increase in Congestion</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Topology for the simulation of the section 3.5</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>Topology used for the simulation of section 3.5</td>
<td>28</td>
</tr>
</tbody>
</table>
Abstract

Random Early Detection (RED) is one of the most prominent congestion avoidance schemes in the Internet routers. The performance of TCP/IP over RED has been widely studied. The studies revealed that although RED can improve TCP performance under certain parameter settings and network conditions, the basic RED algorithm is still susceptible to several problems, such as low throughput, high delay jitter, and bandwidth unfairness. To overcome the limitations of the basic RED algorithm, researchers proposed several variants of RED. In this research we propose an algorithm “RED with Dynamic Threshold Adjustment”, with minimal changes to the overall RED algorithm. Our objectives are the maximization of throughput and minimization of packet drop and delay. Based on traffic conditions, we dynamically modify the thresholds using an exact expression of average queue size for a given burst size and number of nodes. In our algorithm, we set minimum threshold using an expression that we derive assuming a burst size, the maximum thresholds is changed dynamically based on traffic conditions and buffer size which also taking into account the burst size. The assumption behind this update is that maximum threshold will be reached when the instantaneous queue size reaches the maximum buffer size. We implemented this proposed algorithm using the ns-2 simulator. Simulation studies show that our algorithm improves the performance of RED.
Chapter 1
Active Queue Management in TCP/IP Networks

Queue management is defined as the algorithms that manage the length of packet queues by dropping packets when necessary. Performance of the TCP-based applications depends on the choice of queue management. In this chapter we will explain an active queue management technique known as Random Early Detection (RED). This chapter also gives a general description of RED research and our proposed algorithm ARDTA.

1.1 Introduction

Today's Internet is a best-effort connectionless service using IP protocol. There is no guarantee as to the timeliness of delivery, or even actual delivery. This design has many well established advantages in terms of flexibility and robustness. Yet, good and reliable service under heavy load requires careful design, without which internet meltdown\(^1\) can be common and unavoidable.

In 1988 Van Jacobson was the first to provide a fix for Internet meltdown. The congestion avoidance mechanisms he came up with are now standard in TCP implementations. Such mechanisms cause TCP connections to “back off” during congestive periods. However, the TCP congestion avoidance mechanisms, especially at the endpoint, have not proven to be a consistently sufficient solution, and additional mechanisms in the routers are needed as complements.

Two classes of router algorithms related to congestion control are: "queue management" and "scheduling" algorithms. The choice of queue management in the network links is critically relevant for the performance of TCP-based applications. Queue management is defined as the algorithms that manage the length of packet queues by dropping packets when necessary or appropriate. On the other hand scheduling algorithms determine which packet to send next and are used primarily to manage the allocation of bandwidth among flows. From the point of dropping packets, queue management can be classified into two categories. The first category is passive queue management (PQM), which does not employ any preventive packet drop before the router buffer gets full or reaches a specified value. The second category is active

\(^1\) “Internet meltdown”, technically called “congestion collapse” is an extended period of congestion, during which the network is able to perform little or no useful work.
queue management (AQM), which employs preventive packet drop before the router buffer gets full.

1.2 Passive queue management

Passive queue management techniques do not take any preventive packet drops for arriving packets until the buffer level reaches some specified value. But if the buffer level reaches a specified value then all arriving packets are dropped with a probability of one. Passive queue management, therefore, has two states: (1) no packet drop and (2) 100% packet drop. It does not send early congestion warning to senders to decrease their traffic rate with a view to relieving network congestion. A 100% packet drop causes all senders to back off. Two popular PQM techniques are “tail-drop” and “drop-from-front”. The tail-drop scheme drops packets from the tail of the queue. Once the buffer level reaches a certain threshold, all arriving packets are discarded. Packets already in the queue are not affected. On the other hand, drop-from-front discards packets from the front of the queue when the buffer is full or reaches a specified threshold. The arriving packet is accepted, while the packet that is buffered at the front of the queue is discarded; therefore, drop-from-front drops the packet in the buffer with the oldest age. It causes the traffic senders to see the packet loss one buffer drain time earlier than that in the tail-drop case.

1.3 Problems with Passive Queue Management

An Internet router normally is attached to a large number of hosts whose total bandwidth requirements, at certain times, exceed the transmission capacity of the gateway. Buffer is used in the routers to absorb the difference between the required and available capacity. There is a trade-off between the buffer size and QoS. A larger buffer size results in a higher throughput but may result in long delays. On the other hand, for a given buffer size, the buffer management scheme affects the QoS of connections. In addition to these, the following two problems have been observed when they are used to manage buffers that carry traffic that is controlled by the window-based congestion control algorithms of TCP.

- Lock Out: Lock out occurs when a tail-drop scheme allows a single connection or a few connections to monopolize the buffer space of the router by preventing other connections from getting space in the router queue. This results in unfair sharing of network resources among the connections.
• Full queue: Full queue occurs, because tail-drop does not drop packets before the queue is full. It results in long queuing delays because the router queue being full for a long period of time.

It is important to reduce the steady-state queue size, because low end-to-end delay is more important than high throughput. Again packets often arrive at routers in bursts. If the queue is full or almost full, an arriving burst will cause multiple packets to be dropped. This can result in a global synchronization of flows throttling back, followed by a sustained period of lowered link utilization, reducing overall throughput.

The point of buffering in the network is to absorb data bursts and to transmit them during the (hopefully) ensuing bursts of silence. This is essential to permit the transmission of bursty data. Maintaining normally-small queues can result in higher throughput as well as lower end-to-end delay. Queue limits should not affect the steady state queues we want maintained in the network; instead, they should affect the size of bursts we need to absorb.

1.4 Active queue management

Active queue management (AQM) provides preventive measures to manage a buffer to eliminate the problems associated with passive buffer management. In AQM, preventive random packet drop is performed before the buffer is full. The probability of preventive packet drop increases with increasing levels of congestions. Preventive packet drop provides implicit feedback mechanism to notify senders of the onset of congestion. The feedback is used by the senders to reduce their traffic randomly, which prevents all senders from backing off simultaneously and thereby eliminate global synchronization. The goals of AQM as follows:

• Reduce the number of packets dropped in routers to improve throughput;
• Provide a low-delay interactive services by maintaining a small queue size, which reduces the delay seen by flows;
• Avoid lock-out behavior by sharing the bandwidth fairly among the competing flows.

After the IETF recommendation on AQM, many AQM algorithms have been proposed. The default AQM scheme recommended by IETF for the next generation Internet routers is Random Early Detection (RED) [1].

1.5 Random Early Detection

Random Early Detection (RED), was proposed by Floyd and Jacobson [2] in 1993. Figures 1 and 2 show the algorithm and drop function of RED. A router implementing RED accepts all
Adaptive RED with Dynamic Threshold Adjustment Page 4

packets until the queue reaches $\text{Min}_{\text{th}}$, after which it drops a packet with a linear probability distribution function. When the queue length reaches $\text{Max}_{\text{th}}$, all packets are dropped with a probability of one. The basic idea behind RED is that a router detects congestion early by computing the average queue length $\text{avg}$ and sets two buffer thresholds $\text{Max}_{\text{th}}$ and $\text{Min}_{\text{th}}$ for packet drop as shown in Figure 2. The average queue length at time $t$, defined as

$$\text{avg}_t = (1-w) \times \text{avg}_{t-1} + w \times q_t$$

This equation is used as a control variable to perform active packet drop. The $\text{avg}_t$ is the new value of the average queue length at time $t$, $q_t$ is instantaneous queue length at time $t$, and $w$ is a weight parameter in calculating $\text{avg}$. One of RED’s main goals is to use this combination of queue length averaging (which accommodates bursty traffic) and early congestion notification (which reduces the average queue length) to simultaneously achieve low average queuing delay and high throughput. Simulation experiments and operational experience suggest that RED is quite successful in this regard [10]. Normally, $w$ is much less than one. The packet-drop probability, $p$ is calculated by

$$p = \text{Max}_{\text{drop}} \frac{\text{avg} - \text{Min}_{\text{th}}}{\text{Max}_{\text{th}} - \text{Min}_{\text{th}}}.$$  

The RED algorithm, therefore, includes two computational parts: computation of the average queue length and calculation of the drop probability.

\begin{verbatim}
for each packet arrival
  calculate the average queue size \text{avg}
if $\text{Min}_{\text{th}} \leq \text{avg} < \text{Max}_{\text{th}}$
  calculate probability $p_a$
  with probability $p_a$:
  mark the arriving packet
else if $\text{Max}_{\text{th}} \leq \text{avg}$
  mark the arriving packet
\end{verbatim}

**Figure 1.1 General algorithm for RED gateways**

The RED algorithm involves four parameters to regulate its performance. $\text{Min}_{\text{th}}$ and $\text{Max}_{\text{th}}$ are the queue thresholds to perform packet drop, $\text{Max}_{\text{drop}}$ is the packet drop probability at $\text{Max}_{\text{th}}$, and $w$ is the weight parameter to calculate the average queue size from the instantaneous queue length. The average queue length follows the instantaneous queue length. However, because $w$ is much less than one, $\text{avg}$ changes much slower than $q$. Therefore, $\text{avg}$ follows the long-term
changes of $q$, reflecting persistent congestion in networks. By making the packet drop probability a function of the level of congestion, RED gateway has a low packet-drop probability during low congestion, while the drop probability increases as the congestion level increases.

The packet drop probability of RED is small in the interval $\text{Min}_{th}$ and $\text{Max}_{th}$. Moreover, the packets to be dropped are chosen randomly from the arriving packets from different hosts. As a result, packets coming from different hosts are not dropped simultaneously. RED gateway, therefore, avoid global synchronization by randomly dropping packets. The performance of RED significantly depends on the values of its four parameters $[3, 4]$, $\text{Max}_{\text{drop}}$, $\text{Min}_{th}$, $\text{Max}_{th}$, and $w$.

The packet drop probability of RED is small in the interval $\text{Min}_{th}$ and $\text{Max}_{th}$. Moreover, the packets to be dropped are chosen randomly from the arriving packets from different hosts. As a result, packets coming from different hosts are not dropped simultaneously. RED gateway, therefore, avoid global synchronization by randomly dropping packets. The performance of RED significantly depends on the values of its four parameters $[3, 4]$, $\text{Max}_{\text{drop}}$, $\text{Min}_{th}$, $\text{Max}_{th}$, and $w$.

**Figure 1.2: RED gateway drop function**

The detailed algorithm for the RED gateway is given in Figure 3. The gateway’s calculations of the average queue size take into account the period when the queue is empty (the idle period) by estimating the number $m$ of small packets that could have been transmitted by the gateway during the idle period. After the idle period the gateway computes the average queue size as if $m$ packets had arrived to an empty queue during that period. As $\text{avg}$ varies from $\text{Min}_{th}$ to $\text{Max}_{th}$, the packet-marking probability $p_b$ varies linearly from 0 to $\text{Max}_p$:

$$p_b \leftarrow \frac{\text{Max}_p(\text{avg} - \text{Min}_{th})}{(\text{Max}_{th} - \text{Min}_{th})}$$

The final packet-marking probability $p_a$ increases slowly as the count increases since the last marked packet:

$$p_a \leftarrow p_b/(1 - \text{count} \times p_b)$$

This ensures that the gateway does not wait too long before marking a packet. The gateway marks each packet that arrives at the gateway when the average queue size $\text{avg}$ exceeds $\text{Max}_{th}$. One option for the RED gateway is to measure the queue in bytes rather than in packets. With this option, the average queue size accurately reflects the average delay at the gateway. When
this option is used, the algorithm would be modified to ensure that the probability that a packet is marked is proportional to the packet size in bytes:

\[ p_b \leftarrow \text{Max}_p(\text{avg} - \text{Min}_{th})/(\text{Max}_{th} - \text{Min}_{th}) \]

\[ p_b \leftarrow p_b \cdot \text{PacketSize}/\text{MaximumPacketSize} \]

\[ p_a \leftarrow p_b/(1 - \text{count} \times p_b) \]

In this case a large FTP packet is more likely to be marked than a small TELNET packet. The following sections discuss in detail the setting of the various parameters for RED gateways.

*Initialization:* avg \( \leftarrow 0; \) count \( \leftarrow -1; \) for each packet arrival calculate the new average queue size avg:

  if the queue is nonempty
  \[ \text{avg} \leftarrow (1 - w_q) \text{avg} + w_q q \]
  else
  \[ m \leftarrow f(\text{time} - q_{\text{time}}) \]
  \[ \text{avg} \leftarrow (1 - w_q)m \text{avg} \]

if \( \text{Min}_{th} \leq \text{avg} < \text{Max}_{th} \)
increment count calculate probability \( p_a: \)

\[ p_b \leftarrow \text{Max}_p(\text{avg} - \text{Min}_{th})/(\text{Max}_{th} - \text{Min}_{th}) \]

\[ p_a \leftarrow p_b/(1 - \text{count} \times p_b) \]

with probability \( p_a: \)
mark the arriving packet

\[ \text{count} \leftarrow 0 \]
else if \( \text{max}_{th} < \text{avg} \)
mark the arriving packet

\[ \text{count} \leftarrow 0 \]
else count \( \leftarrow -1 \)

when queue becomes empty
\[ q_{\text{time}} \leftarrow \text{time} \]

*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
- Maxp: maximum value for \( p_b \)

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

*Figure 1.3: Detailed algorithm for RED gateways.*
1.5.1 Selection of Maximum Packet Drop Probability, $\text{Max}_{\text{drop}}$

The selection of the maximum drop probability ($\text{Max}_{\text{drop}}$) significantly affects the performance of RED. If $\text{Max}_{\text{drop}}$ is too small, then active packet drops are not enough to prevent global synchronization. Too large a value of $\text{Max}_{\text{drop}}$ decreases the throughput. Although a $\text{Max}_{\text{drop}}$ value of 0.1 is generally suggested [5], the selection of an optimal value of $\text{Max}_{\text{drop}}$ according to network and traffic situation is still an open issue [3, 4]. Feng et al. [6] demonstrated that the value of $\text{Max}_{\text{drop}}$ depends not only on the bandwidth delay but also on the number of connections. The upper bound of packet drop probability ($\text{Max}_{\text{drop}}$) can be expressed as:

$$\text{Max}_{\text{drop}} \leq \frac{N \times SS \times C}{B \tau}$$

where $N$ is the number of connections, $B$ is the total bandwidth, $SS$ is the segment size, $\tau$ is the round-trip time, and $C$ is a constant. From this equation, it is not possible to fix a value of $\text{Max}_{\text{drop}}$ for a dynamically changing network environment, that is, number of connections, round-trip time etc.

1.5.2 Selection of $\text{Min}_{\text{th}}$ and $\text{Max}_{\text{th}}$

For a RED gateway carrying only TCP traffic, $\text{Min}_{\text{th}}$ should be around five packets, and $\text{Max}_{\text{th}}$ should be at least three times $\text{Min}_{\text{th}}$ [5]. Non-TCP traffic does not employ the congestion control mechanisms of TCP. A different set of values are, therefore, required for $\text{Min}_{\text{th}}$ and $\text{Max}_{\text{th}}$ to protect TCP traffic from non-TCP traffic [7, 8].

1.5.3 Selection of Average queue length and weight parameter $w$

RED uses the average queue length as a control variable to perform active packet drop. Calculation of the average queue length involves the previous average queue length and the instantaneous queue length modified by a weight parameter $w$. The average queue length, therefore, is the result of applying a low pass filter to the instantaneous queue size. Thus, the short-term increases in the queue size that result from bursty traffic or from transient congestion do not result in a significant increase in the average queue size. The low-pass filter is an exponential weighted moving average:

$$\text{avg} \leftarrow (1-w_q) \text{avg} + w_q \text{q}$$

The weight $w_q$ determines the time constant of the low-pass filter. The calculation of the average queue size can be implemented particularly efficiently when $w_q$ is a (negative) power of two. The average queue length is required to track persistent network congestion that occurs over a long time range while, at the same time, filtering our short time congestion. This requirement imposes limitations on the selection of $w_q$. If $w_q$ is too small, the average queue length does not catch up
with the long range congestion that may result in the failure of Active Queue Management (AQM). If \( w_q \) is too large, the average queue length tracks the instantaneous queue, which also degrades the performance of AQM. Therefore, the value of \( w_q \) should be related to the traffic flowing in the queue. A simple model to calculate \( w_q \) was developed in [2, 5]. However, the assumptions in developing the model of \( w_q \) were too simple to reflect real TCP traffic. Therefore, in certain situations, the values given in [2, 5] may result in nonoptimal performance of the RED queue [9]. A more realistic model for determining \( w_q \) has been proposed in [9], where the aggregate TCP traffic has been taken into consideration. Results have shown that the values (0.05, 0.07) obtained from the model in [9] give better performance than the values (0.001, 0.002) in [2, 5] in certain cases.

1.6 Modified Average queue calculation

After the burst router will take packet from the ready queue to serve. When the queue is empty, router will use the second method for calculation of average queue. \( \text{avg}_i = (1-w)m \text{avg}_{i-1} \). In this equation, \( m= \text{idle time/s} \), where \( s \) is the service time of a small packet. Default value of the small packet is 100 bytes, for faster decrease. For our simulation we checked both packet size of 100 bytes and 500 bytes, and did not notice any significant difference in the result. Average queue size after the burst sometimes might be above 1 still with empty queue. Therefore when the average queue size is non zero and if the ready queue is empty we can initialize the average queue back to zero. Current RED algorithm calculates average queue for each packet arrival. When the queue is empty, it calculates the average queue using the following formula.

\[
m = f(t - q \text{ time}), \quad \text{avg}_i = (1-w)m \times \text{avg}_{i-1}.
\]

But when the buffer is empty we can initialize the average queue back to 0.

```
for each packet arrival
    calculate the new average queue size avg:
    if the queue is nonempty
        \( \text{avg}_i = (1-w_q) \text{avg}_{i-1} + w_q q_{i-1} \)
    else
        \( m = f(t - q \text{ time}) \)
        if \( B = 0 \) then \( \text{ave} = 0 \)
        else \( \text{avg}_i = (1-w_q)m \text{avg}_{i-1} \)
```

**Parameters:**
- \( \text{avg} \): average queue size; \( q \text{ time} \): start of the queue idle time
- \( w_q \): queue weight; \( q \): current queue size; \( m \): current time
- \( f(t) \): a linear function of the time \( t \)

---

Figure 1.4: Modified algorithm to calculate the average queue
We added the C++ code of figure 5 in the NS-2 simulator to see the result.

```cpp
/*
 * Compute the average queue size.
 * Nqueued can be bytes or packets.
 */
double REDQueue::estimator(int nqueued, int m, double ave, double q_w) {
    double new_ave, old_ave;
    new_ave = ave;
    while (--m >= 1) {
        if nqueued = 0 // added code to calculate new average
            new_ave = 0
        else new_ave *= 1.0 - q_w;
    }
    old_ave = new_ave;
    new_ave *= 1.0 - q_w;
    new_ave += q_w * nqueued;
    double now = Scheduler::instance().clock();
    if (edp_.adaptive == 1) {
        if (edp_.feng_adaptive == 1)
            updateMaxPFeng(new_ave);
        else if (now > edv_.lastset + edp_.interval)
            updateMaxP(new_ave, now);
    }
    return new_ave;
}
```

Figure 1.5: C++ code to calculate the average queue during idle time

1.7 Classifying the RED Variants

The studies revealed that although RED can improve TCP performance under certain parameter settings and network conditions, the basic RED algorithm is still susceptible to several problems, such as low throughput, high delay jitter, and bandwidth unfairness. To overcome the limitations of the basic RED algorithm, researchers proposed several variants of RED. Research on RED and its variants can be classified into two broad categories.

1. The first category deals with modifying the calculation of the control variable and/or drop function.

2. The second category is concerned with configuring and setting RED’s parameters.

In the first category, called aggregate control, the packet-drop probability is nondiscriminative to connections. In the second category, called per-flow control, the packet-drop probability applied to arriving packets can be discriminative to different TCP connections. In per-flow control, the thresholds for the gateway buffer to perform packet drop can be set according to the traffic type
(TCP vs. UDP) resulting in class-based threshold. To improve the performance of the original RED, several variants of RED have been proposed and studied. RED algorithm consists of two parts:

- Calculation of the drop function (linear function vs. step function)
- Calculation of the control variables (aggregate control vs. per-flow control)

### 1.8 Adaptive RED

The RED active queue management algorithm allows network operators to simultaneously achieve high throughput and low average delay. However, the resulting average queue length is quite sensitive to the level of congestion and to the RED parameter settings, and is therefore not predictable in advance. Delay being a major component of the quality of service delivered to their customers, network operators would naturally like to have a rough a priori estimate of the average delays in their congested routers; to achieve such predictable average delays with RED would require constant tuning of the parameters to adjust to current traffic conditions. Adaptive RED solves the problem of RED’s sensitivity to parameters by auto-tuning the various RED parameters to achieve reliably good results [10].

### 1.9 Research Contributions

The performance of RED depends on the thresholds. Link utilization is very low if the thresholds are small. If the thresholds are set too high, then congestion might occur before the end-nodes are notified. The optimal values for Min\(_{th}\) and Max\(_{th}\) depend on the desired average queue size. If the typical traffic is fairly bursty, then Min\(_{th}\) must be correspondingly large to allow the link utilization to be maintained at an acceptably high level [1]. If the average queue is below or upto Max\(_{th}\) before the buffer gets full, then it is possible to avoid some unnecessary packet drops.

To overcome this problem, we propose an effective thresholds selection strategy. We adjust maximum threshold dynamically based on traffic condition and therefore call our algorithm ‘‘Adaptive RED with Dynamic Threshold Adjustment (ARDTA)’’. Using an analytical model, we derive an exact expression to calculate the minimum threshold Min\(_{th}\) for a given burst and buffer size. Then we derive the maximum value of the maximum threshold Max\(_{th}\) assuming average queue will reach the Max\(_{th}\) on or before buffer overflow to avoid any packet drop using droptail. The optimal value for Max\(_{th}\) depends in part on the maximum average delay that can be allowed by the gateway. We initially set Max\(_{th} = 2 \times\) Min\(_{th}\) because the RED gateway functions
most effectively, when $\text{Max}_{th} - \text{Min}_{th}$ is larger than the typical increase in the calculated average queue size in one roundtrip time, and a useful rule-of-thumb is to set $\text{Max}_{th}$ to at least twice $\text{Min}_{th}$ [5]. We then changed $\text{Max}_{th}$ dynamically based on the traffic condition by setting $a_0 = \text{current value of average queue and } q_0 = \text{current value of instantaneous queue.}$

At every packet arrival:

\[
\begin{align*}
\text{if} \ (\text{Max}_{th} < \text{target}) \ {\{}
\text{if} \ (\text{increase} \leq 2 \ * \ \text{Min}_{th})
\quad \text{Max}_{th} = 2 \ * \ \text{Min}_{th} ;
\text{else}
\quad \text{Max}_{th} = \text{Min}_{th} + \text{increase} ;
\}
\text{else} \ \text{Max}_{th} = \text{target} ;
\end{align*}
\]

Variables:

\begin{itemize}
\item $\text{Min}_{th} = \text{Minimum threshold (Set initially based on burst and number of nodes)}$
\item $\text{Max}_{th} = \text{Dynamic Maximum threshold}$
\item $\text{target} = \text{Maximum value of } \text{Max}_{th} \text{ based on Buffer, burst and number of nodes}$
\item $\text{increase} = (1-w)^n k \times a_0 + (1-(1-w)^n k) \times q_0 \ [\text{explained in next section}]$
\item $w = 0.002, n=\text{number of nodes, } k=\text{burst size}$
\end{itemize}

Figure 1.6: Our algorithm (ARDTA)

1.10 Report Outline

The rest of this report is as follows. In Chapter 2 we derive an exact expression of the evolution of the average queue size under given burst characteristics. In Chapter 3 we present our simulation results demonstrating and comparing the performance of ARDTA to other popular algorithm using various traffic conditions. In Chapter 4 we give some concluding remarks together with further research ideas for the improvement of our algorithm. In Appendix A and Appendix B we give proof of some derivations.
Chapter 2:
Analytical Model of ARDTA

In this chapter we will derive an equation for the average queue size under RED. In section 2.1 we describe an analytical mode and using this model we develop an exact expression for average queue size and instantaneous queue size. Section 2.2 explains our thresholds selection strategy using the derived equations of average queue size and instantaneous queue size. In section 2.3, we will explain a modified average queue calculation technique during idle time.

### 2.1 Model to calculate the thresholds

The performance of RED is highly dependent on the thresholds. Most of the threshold selection strategies are based on heuristics and simulations. The problem with these approaches is that they might be good for a particular traffic condition and could give worse result under different traffic situation. In this section we propose a systematic threshold selection strategy.

The basic premise of our strategy is the average queue size should reach the maximum threshold when or before the instantaneous queue size reaches the maximum buffer size and the router experiences heavy load continuously. That is, deterministic dropping of packets according to RED should coincide with a full queue. For the sake of simplicity we assume all the packets have fixed size, the time is slotted and a time slot is equal to the packet transmission time. We also assume that during one time slot, the number of arriving packets is always n, which corresponds to n active router input ports. The router output port can transmit exactly one packet if the output port queue is not empty.

![Figure 2.1: Model to calculate the threshold](image-url)
We consider an output port, and we assume a worst case condition in which $n$ input ports become active simultaneously, and are transmitting to the same output port. The maximum packet burst size on all ports is the same and is equal to $m$. In the following, $q^k_j$ and $a^k_j$ are used to denote the instantaneous queue size and the average queue size after the $j^{th}$ arriving packet in the $k^{th}$ time slot, respectively. Hence $q^k_0$ and $a^k_0$ are the instantaneous queue size and the average queue size at the beginning of the $k^{th}$ slot, respectively. The initial instantaneous queue size and the average queue size of the system are $q^1_0$ and $a^1_0$. We also assume that the maximum buffer size is $B$ packets and the minimum threshold we need to compute is $\min_{th}$. In the following we consider the evolution of the instantaneous queue size and the average queue size. In this model we will consider the interarrival and also the service of the packets. In our model, when the first packet arrives from all $n$ input ports, exactly one packet will be served by the router.

**Time slot 1:**

After the arrival of packet 1:

$q^1_1 = q^1_0$, Because the arriving packet will go to the router for service

$a^1_1 = (1 - w) \times a^1_0 + w \times q^1_1$

After the arrival of packet 2:

$q^1_2 = q^1_0$, (Because router will schedule it as a next packet to serve)

$a^1_2 = (1 - w) \times a^1_1 + w \times q^1_2$

After the arrival of packet 3:

$q^1_3 = q^1_0 + 1,$

$a^1_3 = (1 - w) \times a^1_2 + w \times q^1_3$

... 

After the arrival of packet $n$:

$q^1_n = q^1_0 + (n - 2),$

$a^1_n = (1 - w) \times a^{n-1}_n + w \times q^1_n$

From the above relations, we have:

$a^1_n = (1 - w)^{n-2} [(1 - w) \times a^1_0 + w \times q^1_1] + (1 - w)^{n-2} \times w \times q^1_2 + \ldots + (1 - w) \times w \times q^1_{n-1} + w \times q^1_n$  \hspace{1cm} (1)
Which simplifies to:

\[
\begin{align*}
    a_n^1 &= (1-w)^n \times a_0^1 + wi + w \times [(1-w)^{n-1} + (1-w)^{n-2} + \ldots + (1-w)^2 + (1-w) + 1] + \\
    &\quad w \times [(1-w)^{n-3} + 2 \times (1-w)^{n-4} + 3 \times (1-w)^{n-5} + \ldots + (n-3) \times (1-w) + (n-2)] \\
    &= (1-w)^n \times a_0^1 + w \times a_0^1 \sum_{i=0}^{n-1} (1-w)^i + w \times \sum_{i=0}^{n-3} (n-2-i) \times (1-w)^i \\
    &= (1-w)^n \times a_0^1 + q_0^1 + \left[1 - (1-w)^n\right] + w \times (n-1) - 1 + (1-w)^{n-1} \\
    &= (1-w)^n \times a_0^1 + q_0^1 + \left[1 - (1-w)^n\right] + w
\end{align*}
\]

(2)

Similarly we can derive the equation for \( a_n^k \) and \( q_n^k \) (for \( k > 1 \)):

\[
\begin{align*}
    a_n^k &= (1-w)^{n-1} \times [(1-w) \times a_0^k + w \times q_1^k] + (1-w)^n \times w \times q_2^k + \ldots + (1-w) \times w \times q_{n-1}^k + w \times q_n^k \\
    q_n^k &= q_0^1 + k \times (n-1) - 1 \\
    a_0^k &= a_n^{k-1} \\
    q_0^k &= q_n^{k-1} = q_0^1 + (k-1) \times (n-1) - 1 \\
    q_1^k &= q_0^k + (k-1) \times (n-1) - 1 \quad \text{[Because one packet will go for service]}
\end{align*}
\]

From the above, we obtain (for \( k > 1 \)):

\[
\begin{align*}
    a_n^k &= (1-w)^n \times a_0^1 + w \times a_0^1 \sum_{i=0}^{n-1} (1-w)^i + w \times \sum_{i=1}^{n-2} i \times (1-w)^{n-2-i} + \\
    &\quad w \times \sum_{j=1}^{k-1} \sum_{i=0}^{n-1} \sum_{l=0}^{i-1} [j(n-1)-1+l] \times (1-w)^{n-k} \\
    a_n^k &= (1-w)^n \times a_0^1 + q_0^1 \times [1 - (1-w)^n] + w \times \sum_{i=1}^{n-2} i \times (1-w)^{n-2-i} + \\
    &\quad w \times \sum_{j=1}^{k-1} \sum_{i=0}^{n-1} \sum_{l=0}^{i-1} [j(n-1)-1+l] \times (1-w)^{n-k} \\
    &= a_n^k = a_n^M = a_0^1 = w \times \sum_{i=1}^{n-2} i \times (1-w)^{n-2-i} + w \times \sum_{j=1}^{k-1} \sum_{i=0}^{n-1} \sum_{l=0}^{i-1} [j(n-1)-1+l] \times (1-w)^{n-k}
\end{align*}
\]

(3)

The details of the derivations of equations (1), (2) and (3) are given in Appendices A.1, A.2, and A.5, respectively.

### 2.2. Minimum and Maximum Thresholds calculation

Suppose burst size = \( M \) (\( k=M \)) and initially when the connection starts the \( a_0^1 = q_0^1 = 0 \)

Therefore from equation (3) we find, average queue

\[
\begin{align*}
    a_n^k &= a_n^M = a_0^1 = w \times \sum_{i=1}^{n-2} i \times (1-w)^{n-2-i} + w \times \sum_{j=1}^{k-1} \sum_{i=0}^{n-1} \sum_{l=0}^{i-1} [j(n-1)-1+l] \times (1-w)^{n-k}
\end{align*}
\]
Number of packets in the buffer will be \( q_n^k = q_n^M = q_0^1 + k \times (n - 1) - 1 \)

We set our minimum threshold (\( \text{Min}_{th} \)) equal to \( a_n^M \). Initially we set a target Maximum threshold (\( \text{Max}_{th} \)) equal to twice of \( \text{Min}_{th} \). We check the increment of average queue size at every packet arrival by setting the current value of average queue and instantaneous queue in equation (3). When the value of the average queue exceeds our target maximum threshold, we start changing the maximum threshold dynamically based on the increment. If the average queue goes below the target Maximum threshold then we reset the Maximum threshold back to target.

### 2.3 Modified average queue during idle time

During idle time RED router use the equation \( \text{avg}_i = (1 - w)^m \text{avg}_{i-1} \) to calculate average queue. Here \( m = (\text{time} - q\_time)/s \), where \( s \) is the time to serve a small packet. We propose a strategy that, when the instantaneous queue becomes empty, we will reset the average queue back to zero. We implemented this strategy using ns-2 simulator and found better performance of the RED router during idle time.

### 2.4 Summary

In this chapter we derived an exact expression for the average queue size for a given burst size and number of nodes. We use this expression to set the thresholds for our algorithm ARDTA. We use this thresholds selection strategy for our simulation in the next chapter. We also propose a strategy for calculation average queue during idle time.
Chapter 3

Simulation results and Analysis

This chapter summarizes the simulation results of our algorithm using ns-2. Section 3.1 gives a simulation model. We present the simulation result for Poisson Process, self similar traffic and FTP traffic using TCP in Sections 3.2, 3.3 and 3.4, respectively. In Section 3.5 we present and compare the simulation results of some popular algorithm with our proposed algorithm. At the end of the chapter in Section 3.6 we summarize the results.

3.1 Simulation Model

The simulation model is based on the topology of Figure 3.1. We used ns-2 network simulator for the simulations. The network consists of five sources, one destination node and one router. All the five sources, S₁ through S₅, are connected to the router R₁ using 10 Mb/s and 10ms link. Bottleneck node router R₁ implements RED and the destination node D₁ is directly connected to this node using a 20 Mb/s and 20ms link. For all the simulations, we used a fixed packet size of 500 bytes. RED parameter w is equal to 0.002, buffer size is 100 packets and maximum packet drop probability is 1/50. We experimented with several packet arrival processes including a Poisson arrival process, self similar traffic and self similar traffic with random start time.

3.2 Simulation with Poisson Arrival Process

To generate Poisson traffic we used the Exponential On/Off generator of NS-2 simulator. Exponential On/Off generator can be configured to behave as a Poisson process by setting the variable burst_time_ to 0 and the variable rate_ to a very large value. The C++ code guarantees that even if the burst time is zero, at least one packet is sent. Additionally, the next interarrival time is the sum of the assumed packet transmission time (governed by the variable rate_) and the random variable corresponding to idle_time_. Therefore, to make the first term in the sum very small, make the burst rate very large so that the transmission time is negligible compared to the typical idle times [12].

In the first test we used fixed Min_th and Max_th values. Then we applied the ARDTA algorithm. We calculate the throughput, average delay and average packet drop ratios in all cases.
3.1.1 Simulation with $\text{Min}_{th}=3$ and $\text{Max}_{th}=9$ (SIM$_1$)

In this experiment we used traffic of Poisson arrival process with fixed minimum and maximum thresholds ($\text{Min}_{th}=3$ and $\text{Max}_{th}=9$). Result of this simulation is given in the Table 3.1 below. Figure 3.2 and 3.3 are the graph of total throughput and average queue.

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total average throughput</td>
<td>18.3</td>
</tr>
<tr>
<td>Number of packets dropped</td>
<td>81</td>
</tr>
<tr>
<td>Average packet drop</td>
<td>0.350437%</td>
</tr>
<tr>
<td>Total average delay (in seconds)</td>
<td>0.001095</td>
</tr>
</tbody>
</table>
```

Table 3.1: Simulation result of the SIM$_1$ experiment
3.1.2 Simulation with Minₜₜ=4 and Maxₜₜ=12 (SIM₂)

In this experiment also we used traffic of Poisson arrival process with different fixed minimum and maximum thresholds (Minₜₜ=4 and Maxₜₜ=12). Results of this simulation are given in the Table 3.2 below. Figure 3.4 and 3.5 are the graph of total throughput and average queue.

<table>
<thead>
<tr>
<th>Total average throughput</th>
<th>18.44</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of packets dropped</td>
<td>65</td>
</tr>
<tr>
<td>Average packet drop</td>
<td>0.276091%</td>
</tr>
<tr>
<td>Total average delay (in seconds)</td>
<td>0.001209</td>
</tr>
</tbody>
</table>

Table 3.2: Simulation result of SIM₂

3.1.3 Simulation of ARDTA with Poisson arrival process (SIM₃)

In this simulation experiment we implemented the ARDTA algorithm with the Poisson arrival process. We set Minₜₜ equal to 4.15 assuming a burst of 15 packets from all the 5 nodes \(a₁^{15} = 4.14967\) and we changed Maxₜₜ dynamically by setting current queue size equal to \(q₀\), and the current average equal to \(a₀\) for each transmitted packets.

For the simulation we set buffer size equal to 100 and number of nodes equal to 5. Results of this simulation are given in Table 3.3. Figure 3.6 and 3.7 are the graphs of total throughput and average queue.
Total average throughput | 18.50
--- | ---
Number of packets dropped | 50
Average packet drop | 0.213895%
Total average delay (in seconds) | 0.001068

Table 3.3: Simulation result of SIM\textsubscript{3}

![Figure 3.6: Total throughput for the SIM\textsubscript{3} experiment](image1)

![Figure 3.7: Average queue for the SIM\textsubscript{3} experiment](image2)

### 3.1.4 Summary of the Simulation for Poisson Process

We summarize the simulation results of the simulations SIM\textsubscript{1} SIM\textsubscript{2} and SIM\textsubscript{3} in the Table 3.4. From this Table it is clear that our algorithm gives better performance than RED with fixed thresholds for Poisson process. By implementing our algorithm, we found that throughput has increased by 1.09% and packet drop decreased by 38% from Poisson Process (Min\textsubscript{th}=3 Max\textsubscript{th}=9). Comparing our algorithm with Poisson Process (Min\textsubscript{th}=4 Max\textsubscript{th}=12) we found that throughput has increased by 0.33% and packet drop decreased by 23%.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Throughput</th>
<th>Percent of packets Drop</th>
<th>Mean Delay</th>
<th>Total drops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson Process (Min\textsubscript{th}=3 Max\textsubscript{th}=9)</td>
<td>18.3</td>
<td>0.35%</td>
<td>0.000939</td>
<td>81</td>
</tr>
<tr>
<td>Poisson Process (Min\textsubscript{th}=4 Max\textsubscript{th}=12)</td>
<td>18.44</td>
<td>0.28%</td>
<td>0.001209</td>
<td>65</td>
</tr>
<tr>
<td>ARDTA</td>
<td>18.50</td>
<td>0.21%</td>
<td>0.001068</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 3.4: Summary of the simulations SIM\textsubscript{1} SIM\textsubscript{2} and SIM\textsubscript{3}
3.2 Simulation with Self similar traffic

A self-similar or long-range-dependant (LRD) network traffic can be generated by multiplexing several sources of Pareto-distributed ON and OFF periods. In a context of a packet-switched network the ON periods correspond to packet train – packets transmitted back to back, or separated only by a relatively small preamble. OFF periods are the periods of silence between packet trains. Multiple sources contributing to resulting synthetic traffic trace may be thought of as individual flows (connections). It is reasonable to assume that packet sizes within a connection remain constant. Different connections, however, can have packets of different sizes. Pareto distribution has the following probability density function:

$$P(x) = \frac{ab^x}{x^{a+1}} \quad x \geq b$$

Where $\alpha$ is a shape parameter (tail index), and $b$ is minimum value of $x$. When $\alpha \leq 2$, the variance of the distribution is infinite. When $\alpha \leq 1$, the mean value is infinite as well. For self-similar traffic, $\alpha$ should be between 1 and 2. The lower the value of $\alpha$, the higher the probability of an extremely large $x$. We used ns-2 simulator to generate this self-similar traffic. To generate Pareto On/Off traffic we used packet size of 200Kb, rate 4.5Mb/s, burst time 100ms and idle time 100ms and $\alpha$ value of 1.5. We used the topology of Figure 3.1 for this simulation.

3.2.1 Simulation of Self similar traffic with Min$_{th}$=3 and Max$_{th}$=9 (SIM$_{4}$)

In this experiment we used Self-similar traffic with fixed minimum and maximum thresholds (Min$_{th}$ =3 and Max$_{th}$ = 9). Result of this simulation is given in the Table 3.5 below. Figure 3.8 and 3.9 are the graph of total throughput and average queue.

<table>
<thead>
<tr>
<th>Total average throughput</th>
<th>13.93</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of packets dropped</td>
<td>486</td>
</tr>
<tr>
<td>Average packet drop</td>
<td>1.08%</td>
</tr>
<tr>
<td>Total average delay (in seconds)</td>
<td>0.0004</td>
</tr>
</tbody>
</table>

Table 3.5: Simulation result of the SIM$_{4}$ experiment
3.2.2 Simulation of Self similar traffic with Min\text{th}=4 and Max\text{th}=12 (SIM\textsubscript{5})

In this experiment also we used Self similar traffic with fixed minimum and maximum thresholds (Min\text{th}=4 and Max\text{th}=12). Results of this simulation are given in the Table 3.5 below. Figure 3.10 and 3.11 are the graph of total throughput and average queue.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total average throughput</td>
<td>14.1</td>
</tr>
<tr>
<td>Number of packets dropped</td>
<td>352</td>
</tr>
<tr>
<td>Average packet drop</td>
<td>0.811%</td>
</tr>
<tr>
<td>Total average delay (in seconds)</td>
<td>0.000472</td>
</tr>
</tbody>
</table>

Table 3.6: Simulation result of SIM\textsubscript{5}
3.2.3 Simulation of ARDTA with Self similar traffic (SIM$_6$)

In this simulation experiment we implemented the ARDTA algorithm with the Self similar traffic. We set Min$_{th}$ equal to 4.15 assuming a burst of 15 packets from all the 5 nodes ($a_5^{15} = 4.14967$) and we changed Max$_{th}$ dynamically by setting current queue size equal to $q_0$, and the current average equal to $a_0$ for each transmitted packets.

For the simulation we set buffer size equal to 100 and number of nodes equal to 5. Results of this simulation are given in Table 3.7. Figure 3.12 and 3.13 are the graphs of total throughput and average queue.

<table>
<thead>
<tr>
<th>Total average throughput</th>
<th>15.86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of packets dropped</td>
<td>173</td>
</tr>
<tr>
<td>Average packet drop</td>
<td>0.43%</td>
</tr>
<tr>
<td>Total average delay (in seconds)</td>
<td>0.000384</td>
</tr>
</tbody>
</table>

Table 3.7: Simulation result of SIM$_6$
3.2.4 Summary of the Simulation with Self-similar Traffic (SST)

We summarize the simulation results of the simulations SIM$_4$, SIM$_5$ and SIM$_6$ in the Table 3.8. From this Table it is clear that our algorithm gives better performance than RED with fixed thresholds for Poisson process. By implementing our algorithm, we found that throughput has increased by 1.09% and packet drop decreased by 38% from Poisson Process (Min$_{th}$=3 Max$_{th}$=9). Comparing our algorithm with Poisson Process (Min$_{th}$=4 Max$_{th}$=12) we found that throughput has increased by 0.33% and packet drop decreased by 23%.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Throughput</th>
<th>Percent of packets Drop</th>
<th>Mean Delay</th>
<th>Total drops</th>
</tr>
</thead>
<tbody>
<tr>
<td>SST (Min$<em>{th}$=3 Max$</em>{th}$=9)</td>
<td>13.93</td>
<td>1.08%</td>
<td>0.0004</td>
<td>486</td>
</tr>
<tr>
<td>SST (Min$<em>{th}$=4 Max$</em>{th}$=12)</td>
<td>14.1</td>
<td>0.81%</td>
<td>0.00047</td>
<td>352</td>
</tr>
<tr>
<td>ARDTA</td>
<td>15.86</td>
<td>0.43%</td>
<td>0.000384</td>
<td>173</td>
</tr>
</tbody>
</table>

Table 3.8: Summary of the simulations SIM$_4$ SIM$_5$ and SIM$_6$

3.3 Simulation with FTP traffic using TCP (FTP_TCP)

For this experiment we used FTP application protocol over TCP agent. We used the same topology of Figure 3.1. All the sources, S$_1$ through S$_5$, are connected to destination D$_1$ by a TCP agent. The window size of all the sources is 60 and they use the application FTP to send packets
from source to destination. At first in the router we used RED with fixed thresholds and then finally we used ARDTA. Start time of all the sources is 0. Results of this simulation are summarized in the Table 3.9 below.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Throughput</th>
<th>Mean Delay</th>
<th>Total drops</th>
<th>Percent Drops</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTP_TCP using RED</td>
<td>11.96</td>
<td>0.00052</td>
<td>108</td>
<td>0.36%</td>
</tr>
<tr>
<td>(Min_th=3 Max_th=9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FTP_TCP using RED</td>
<td>12.70</td>
<td>0.000885</td>
<td>84</td>
<td>0.26%</td>
</tr>
<tr>
<td>(Min_th=4 Max_th=12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FTP_TCP using ARDTA</td>
<td>13.81</td>
<td>0.000881</td>
<td>47</td>
<td>0.14%</td>
</tr>
</tbody>
</table>

Table 3.9: Summary of the simulations of FTP application over TCP

From Table 3.9 it is clear that our algorithm gives better performance than RED with fixed thresholds for self-similar traffic. By implementing our algorithm we found that throughput has increased by 15.47% and packet drop decreased by 56% from FTP_TCP using RED (Min\_th=3 Max\_th=9). Comparing our algorithm with FTP_TCP using RED (Min\_th=4 Max\_th=12), we found throughput has increased by 8.74% and packet drop decreased by 44%.

3.4 Comparison of ARDTA with RED and adaptive RED

For this comparison we used the topology shown in Figure 3.14, which was used in reference [10]. Sources S\_1 and S\_2 are connected to router R\_1 with a 10mb 0ms and 10mb 1ms link, and sources S\_3 and S\_4 are connected to router R\_2 with a 10mb 2ms and 10mb 3ms link respectively. Router R\_1 and R\_2 are connected by a 1.5mb 10ms link and both of them implement RED with a buffer size of 100. The forward traffic consists of two long-lived TCP flows, and the reverse traffic consists of one long-lived TCP flow. At time 25, twenty new flows start, one every 0.1 seconds, each with a maximum window of twenty packets. This is not intended to model a realistic load, but simply to illustrate the effect of a sharp change in the congestion level. The graphs in Figures 3.15, 3.16 and 3.17 show the average of the three algorithms with an increase in congestion. We also checked the average queue size individually. From the graph of the average queue it is clear that our algorithm has better response (less oscillation) during
congestion. As a result we get better throughput and fewer drops; Table 3.10 summarizes the results with ARDTA, where the throughput has increased by 1.08% and packet drops had decreased by 6.95%.

Table 3.10: Summary of the simulations of self similar traffic

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>aggregate per-link throughput(%)</th>
<th>aggregate per-link drops(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>93.87</td>
<td>8.2</td>
</tr>
<tr>
<td>Adaptive RED</td>
<td>94.33</td>
<td>7.8</td>
</tr>
<tr>
<td>ARDTA</td>
<td>94.88</td>
<td>7.63</td>
</tr>
</tbody>
</table>

Figure 3.14: Topology for the simulation of section 3.4

Figure 3.15: Average queue for RED with and Increase in Congestion

Figure 3.16: Average queue for Adaptive RED with and Increase in Congestion
3.5 Comparison of ARDTA with ARED\textsubscript{Floyd} and ARED\textsubscript{Feng}

For the purpose of this comparison we used the topology in Figure 3.18, which was described in reference [11]. In the experiment we compare our algorithm with the Adaptive RED by Floyd (ARED\textsubscript{Floyd}) and the Adaptive RED by Feng (ARED\textsubscript{Feng}). Sources $S_1$ and $S_2$ are connected to router $R_1$ with a 10mb 2ms and 10mb 3ms link respectively. Router $R_1$ is connected to destination $S_3$ through the router $R_2$. Link between $R_1$ and $R_2$ is 1.5mb 3ms and link between $R_2$ and $S_3$ is 10mb 4ms. The forward traffic consists of TCP connections supporting long lived FTP traffic from $S_1$ to $S_3$ at time 0 second, from $S_3$ to $S_1$ at time 1 second and from node $S_2$ to $S_3$ at time 3 seconds. The TCP congestion window size is 15 packets and the packet size is 1600 bytes.
Figure 3.18: Topology for the simulation of the section 3.5

<table>
<thead>
<tr>
<th></th>
<th>$\text{Th}_{\text{min}}$</th>
<th>$\text{Th}_{\text{max}}$</th>
<th>Throughput</th>
<th>Drop</th>
<th>% drop</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARDTA</td>
<td>4.15</td>
<td>Dynamic</td>
<td>1.41</td>
<td>77</td>
<td>0.21</td>
<td>0.120701</td>
</tr>
<tr>
<td>ARED$_{\text{Floyd}}$</td>
<td>5</td>
<td>15</td>
<td>1.40</td>
<td>84</td>
<td>0.23</td>
<td>0.120517</td>
</tr>
<tr>
<td>ARED$_{\text{Feng}}$</td>
<td>5</td>
<td>15</td>
<td>1.08</td>
<td>993</td>
<td>2.73%</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 3.11: Summary of the simulation results of section 3.5 using topology of figure 3.18

We also checked this algorithms using topology of the Figure 3.19 with bottleneck router equal to 10Mbps. The results of the simulation are summarized in the Table 3.12.

<table>
<thead>
<tr>
<th></th>
<th>Threshold</th>
<th>Mean Delay</th>
<th>Average Throughput</th>
<th>Total drops</th>
<th>Percent Drops</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARED$_{\text{Floyd}}$</td>
<td>$\text{Th}_{\text{min}} = 5$</td>
<td>0.003315</td>
<td>7.16</td>
<td>56</td>
<td>0.31%</td>
</tr>
<tr>
<td></td>
<td>$\text{Th}_{\text{max}} = 15$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARED$_{\text{Feng}}$</td>
<td>$\text{Th}_{\text{min}} = 5$</td>
<td>0.002120</td>
<td>7.44</td>
<td>127</td>
<td>0.672%</td>
</tr>
<tr>
<td></td>
<td>$\text{Th}_{\text{max}} = 15$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARDTA</td>
<td>$\text{Th}_{\text{min}} = 4.15$</td>
<td>003519</td>
<td>7.95</td>
<td>55</td>
<td>0.14%</td>
</tr>
<tr>
<td></td>
<td>$\text{Th}_{\text{max}} = \text{changed dynamically}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.12: Summary of the simulation results using topology of figure 15
From Table 3.11 and 3.12 it is clear that our algorithm give better result than Adaptive RED of Floyd and Feng. By implementing our algorithm we found the average throughput has increased by 11% and packet drop decreased by 1.8% from ARED$_{floyd}$. Comparing our algorithm with ARED$_{feng}$ we found throughput has increased by 6.8% and packet drop decreased by 9.6%.

![Topology diagram](image)

**Figure 3.19: Topology used for the simulation of the section 3.5**

### 3.6 Summary

In this section we conducted several simulations using network simulator ns-2 to check the performance of our algorithm. For all the cases we have noticed that our algorithm gives higher throughput as a result of decrease in packet drops. A major disadvantage of our algorithm is that we have to assume a burst size in advance, which may not be correct all the time.
Chapter 4:  
Conclusions and Future Research

This Chapter gives some concluding remarks of our research. Section 4.1 briefly explains our strategies and findings; Section 4.2 gives some further research ideas for the improvement of our algorithm.

4.1 Conclusions

Random Early Detection gateways are an effective mechanism for congestion avoidance at the gateway, by using average queue size. This provides an upper bound on the average delay at the gateway. The probability that the RED gateway chooses a particular connection to notify during congestion is roughly proportional to that connection’s share of the bandwidth at the gateway. This approach avoids a bias against bursty traffic at the gateway. For RED gateways, the rate at which the gateway marks packets depends on the level of congestion, avoiding the global synchronization that results from many connections decreasing their windows at the same time.

The RED algorithm allows network operators to simultaneously achieve high throughput and low average delay. However, the resulting average queue length is quite sensitive to the level of congestion and to the RED parameter settings, and is therefore not predictable in advance. Delay being a major component of the quality of service delivered to their customers, network operators would naturally like to have a rough a priori estimate of the average delays in their congested routers. To achieve such predictable average delays, RED would require constant tuning of the parameters to adjust to current traffic conditions.

In this report we addressed the above problem with minimal changes to the overall RED algorithm. Our objective was to maximize throughput, and minimize the packet drop ratio as well as the packet delay. To do so we derived an exact expression of the average queue size in terms of the burst size and number of nodes and proved it with simulation. We also proposed an algorithm “ARDTA” for adjusting the minimum and maximum thresholds. In the algorithm, we set minimum threshold using our derived expression, assuming a burst size and then changed maximum thresholds dynamically based on traffic conditions and buffer size. We dynamically adjusted the maximum threshold in order to guarantee that the average queue size will not reach the maximum threshold value until the instantaneous queue size reaches the maximum buffer
size. By controlling the average queue size before the gateway queue overflows, RED gateways could be particularly useful in networks where it is undesirable to drop packets at the gateway. This strategy, however, leaves the choice of the burst size to network operators who must make a policy tradeoff between utilization and delay.

4.2 Future Research

There are many areas for extending research on this topic. One way of extending this research could be to change the packet drop probability dynamically together with maximum thresholds, again, taking into account the buffer size. For example, the packet dropping probability should be computed to make sure that with a steady increase in packet arrivals, dropping a packet will cause a reduction in packet arrival rates before the buffer fills, assuming a worst case propagation delay. In addition, we would like to extend this approach to the case in which the burst sizes are different. The model we assume in this report is conservative since it assumes the worst case burst size.
Appendix: A

A.1

Suppose n = 5

\[ a_1^n = (1 - w)\times a_0^n + w\times q_1^n \]
\[ a_2^n = (1 - w)\times a_1^n + w\times q_2^n \]
\[ a_3^n = (1 - w)\times a_2^n + w\times q_3^n \]
\[ a_4^n = (1 - w)\times a_3^n + w\times q_4^n \]
\[ a_5^n = (1 - w)\times a_4^n + w\times q_5^n \]

\[ \therefore a_5^n = (1 - w)\times a_4^n + w\times q_5^n \]
\[ = (1 - w)\times [(1 - w)\times a_3^n + w\times q_4^n] + w\times q_5^n \]
\[ = (1 - w)^2\times a_3^n + w(1 - w)\times q_4^n + w\times q_5^n \]
\[ = (1 - w)^2\times [(1 - w)\times a_2^n + w\times q_3^n] + w\times (1 - w)\times q_4^n + w\times q_5^n \]
\[ = (1 - w)^3\times a_2^n + w(1 - w)^2\times q_3^n + w\times (1 - w)\times q_4^n + w\times q_5^n \]
\[ = (1 - w)^3\times [(1 - w)\times a_1^n + w\times q_2^n] + w\times (1 - w)^2\times q_3^n + w\times (1 - w)\times q_4^n + w\times q_5^n \]
\[ = (1 - w)^4\times a_1^n + w(1 - w)^3\times q_2^n + w\times (1 - w)^2\times q_3^n + w\times (1 - w)\times q_4^n + w\times q_5^n \]
\[ = (1 - w)^4\times [(1 - w)\times a_0^n + w\times q_1^n] + w\times (1 - w)^3\times q_2^n + w\times (1 - w)^2\times q_3^n + w\times (1 - w)\times q_4^n + w\times q_5^n \]
\[ \quad \quad \quad \quad \quad = (1 - w)^n \times [a_0^n + w \times q_1^n] + (1 - w)^{n-1} \times w \times q_2^n + \ldots + (1 - w) \times w \times q_{n-1}^n + w \times q_n^n \quad \ldots \text{(a.1.1)} \]

Therefore after nth arrival we will have:

\[ a_0^n = (1 - w)^{n-1} \left[ a_0^n + w \times q_1^n \right] + (1 - w)^{n-2} \times w \times q_2^n + \ldots + (1 - w) \times w \times q_{n-1}^n + w \times q_n^n \]

A.2

\[ q_1^n = q_0^n, \quad q_2^n = q_0^n + 1, \quad q_3^n = q_0^n + 2, \quad q_4^n = q_0^n + 3, \ldots, q_n^n = q_0^n + (n - 2) \]
in equation a.1.1 if we substitute the values we find

\[
a_s^1 = (1-w)^5 \times a_0^1 + w \times (1-w)^4 \times q_0^1 + w \times (1-w)^3 \times q_0^1 + w \times (1-w)^2 \times [q_0^1 + 1] + w \times (1-w) \times [q_0^1 + 2] + w \times [q_0^1 + 3]
\]

\[
= (1-w)^5 \times a_0^1 + w \times q_0^1 \times [(1-w)^4 + (1-w)^3 + (1-w)^2 + (1-w) + 1] + w \times [(1-w)^2 + 2 \times (1-w) + 3]
\]

Similarly after nth packet arrival we will have

\[
a_n^1 = (1-w)^n \times a_0^1 + w \times q_0^1 \times \sum_{i=0}^{n-1} (1-w)^i + w \times \sum_{i=0}^{n-3} (n-2-i) \times (1-w)^i \quad \ldots \ldots \text{(a.2.1)}
\]

A.3

\[
w \times q_0^1 \times \sum_{i=0}^{n-1} (1-w)^i
\]

Let \[S = \sum_{i=0}^{n-1} (1-w)^i = 1 + (1-w) + (1-w)^2 + \ldots + (1-w)^{n-2} + (1-w)^{n-1} \quad \ldots \ldots \text{(a.3.1)}\]

\[
(1-w) \times S = \sum_{i=0}^{n-1} (1-w)^i = (1-w) + (1-w)^2 + \ldots + (1-w)^{n-1} + (1-w)^n \quad \ldots \ldots \text{(a.3.2)}
\]

If we subtract equation a.3.2. from equation a.3.1 we find

\[
w \times S = 1 - (1-w)^n\]

\[
S = \frac{1-(1-w)^n}{w}
\]
\\[ w \times \sum_{i=0}^{n-3} (n - 2 - i) \times (1 - w)^i \]

Let

\[ S = \sum_{i=0}^{n-3} (n - 2 - i) \times (1 - w)^i = (n - 2) + (n - 3) \times (1 - w) + (n - 4) \times (1 - w)^2 + \ldots + (1 - w)^{n-3} \ldots a.4.1 \]

\[ S \times (1 - w) = (n - 2) \times (1 - w) + (n - 3) \times (1 - w)^2 + (n - 4) \times (1 - w)^3 + \ldots + (1 - w)^{n-2} \ldots a.4.2 \]

Therefore if we subtract a.4.2 from a.4.2 we will find

\[ S \times w = (n - 2) - [(1 - w) + (1 - w)^2 + (1 - w)^3 + \ldots + (1 - w)^{n-2}] \]
\[ = (n - 1) - [1 + (1 - w)^2 + (1 - w)^3 + \ldots + (1 - w)^{n-2}] \]
\[ = (n - 1) - \sum_{i=0}^{n-2} (1 - w)^i \]
\[ = (n - 1) - \frac{[1 - (1 - w)^{n-1}]}{1 - (1 - w)} \]
\[ \therefore S = \frac{w \times (n - 1) - 1 + (1 - w)^{n-1}}{w^2} \]

\[ \therefore w \times \sum_{i=0}^{n-3} (n - 2 - i) \times (1 - w)^i = \frac{w \times (n - 1) - 1 + (1 - w)^{n-1}}{w} \]

Equation a.2.1 will be simplified to
\[ a_i^1 = (1 - w)^n \times a_0^1 + q_0^1 \times [1 - (1 - w)^n] + \frac{w \times (n - 1) - 1 + (1 - w)^{n-1}}{w} \]

A.5

\[ w \times q_0^1 \times \sum_{i=0}^{n_k-1} (1 - w)^i \]

Let \[ S = \sum_{i=0}^{n_k-1} (1 - w)^i = 1 + (1 - w) + (1 - w)^2 + \ldots + (1 - w)^{n_k-2} + (1 - w)^{n_k-1} \quad \ldots \text{(a.6.1)} \]

\[ (1 - w) \times S = \sum_{i=0}^{n_k-1} (1 - w)^i = (1 - w) + (1 - w)^2 + \ldots + (1 - w)^{n_k-1} + (1 - w)^{n_k} \quad \ldots \text{(a.6.2)} \]

If we subtract equation a.6.2. from equation a.6.1 we find

\[ w \times S = 1 - (1 - w)^n_k \]

\[ S = \frac{1 - (1 - w)^n_k}{w} \]

\[ \therefore w \times q_0^1 \times \sum_{i=0}^{n_k-1} (1 - w)^i = q_0^1 \times (1 - (1 - w)^n_k) \]
Appendix B:

B.1

\[ a_n^k = (1 - w)^{nk} \times a_0^1 + q_0^1 \times [1 - (1 - w)^{nk}] + w \times \sum_{i=1}^{n-2} i \times (1 - w)^{nk-2-i} + \]
\[ w \times \sum_{j=1}^{k-1} \sum_{i=0}^{n-1} [j(n-1) - 1 + l] \times (1 - w)^{nk-nj-i} \]

Initially when \( a_0^1 = q_0^1 = 0 \), we can calculate the Minimum threshold to avoid unnecessary packet discard for \( n \) number of connections with a given burst size of \( M \).

\[ a_n^k = Min_{th} = w \times \sum_{i=1}^{n-2} i \times (1 - w)^{nk-2-i} + w \times \sum_{j=1}^{k-1} \sum_{i=0}^{n-1} [j(n-1) - 1 + l] \times (1 - w)^{nk-nj-i} \]

B.2 C++ code to calculate the \( \text{Thresh}_{\text{min}} \)

```cpp
#include <iostream.h>
#include <math.h>

int main()
{

    //programmer: Md. Manzoor Murshed
    //Program: A program to calculate exact minimum
    //Threshold of a RED router for a given burst size
    //Input: n: Number of active connection
    //k: Maximum Burst size in each connection
    //Output: Threshold(min)

    #include <iostream.h>
    #include <math.h>

    int main()
```
```c
{

int n, i, j, l;
double k, t, t1, t2;

const double w= 0.002;
t=t1=t2=0;

cout << "n (number of active connection) = " ;
cin >> n;
cout << "K (Maximum burst size)= " ;
cin >> k;

for (i=1; i<=n-2; i++)
{
t1+= (i* pow((1-w), (n*k-2-i))); //
}

for (j=1; j<=k-1; j++)
{
    for (l=0; l<=n-1; l++)
    {
        t2+=((j*(n-1)-1+l)*pow((1-w), (n*k-1-n*j-l)));
    }
}

 t = t1+t2;
cout <<"Thresh = " << t*w;
return 0;
}
```
References


