Implementation of Quality of Service using DiffServ and AQM algorithm

Final Report

30th November 2014

Team

Ankita Pise
Joshua Clark
Priyanka Tiwari Dikshit
Yinglei Zhang

Advisor: Dr. Rudra Dutta

Abstract

In today's times, we see that majority of the 'traffic classification' in a corporate or a large network setup is done using Virtual LANs (VLANs). However, this requires every switch in the path of traffic flow to be configured for the respective VLANs so that QoS rates are adhered to. This project aims at achieving the objective of using a simplified network setup to provide different QoS to different kinds of traffic based on the priority. This will be implemented by using the DSCP (Differentiated Services Code Point) bits integrated into an Active Queue Management algorithm to classify the incoming traffic into different levels of priority. A switch that uses this AQM algorithm will then have the ability to perform the classful routing of traffic based on priority without having to set up different VLANs.

Table of Contents

Abstract……

Problem Statement.....**Error! Bookmark not defined.**

Proposed Solution.....**Error! Bookmark not defined.**

System Description.....**Error! Bookmark not defined.**

Block Diagram.....**Error! Bookmark not defined.**

Development and Test Topology.....**Error! Bookmark not defined.**

Packet Flow.....**Error! Bookmark not defined.**

Low Level Design.....**Error! Bookmark not defined.**

Reusable Functions 11

RED Implementation with Integrated Classful Probability Marking**Error! Bookmark not defined.**

BLUE Implementation with Integrated Classful Probability Marking**Error! Bookmark not defined.**

Test Plan.....**Error! Bookmark not defined.**

Test/Demo Network Topology.....**Error! Bookmark not defined.**

Test Setup.....**Error! Bookmark not defined.**

Test Procedures .....**Error! Bookmark not defined.**

Traffic Test Plan.....**Error! Bookmark not defined.**

Demonstration.....**Error! Bookmark not defined.**

Demo Network Topology.....**Error! Bookmark not defined.**

Timeline .....**Error! Bookmark not defined.**

Conclusion .....**Error! Bookmark not defined.**

References..... 29

Problem Statement

Current traffic management solutions focus on dividing traffic. One of the ways this is done is by using Virtual LANs (VLANs). VLANs physically separate traffic into travelling through different pipes. Other techniques, like the ingress policing techniques typically separate traffic into different queues, relying on a scheduler to shape the outgoing packets in a fair manner. However, both of these solutions can experience bloat in an extensive and iterative implementation [1].

There is a different solution possibly resulting in less bloat that involves policing a single queue and considering the class of traffic when deciding to allow or deny entry to the queue. The current implementation of the project emphasizes on using this concept in handling traffic of the following three types: Video traffic, VoIP and data transfer.

Proposed Solution

This project implements modified version of an existing Active Queue Management (AQM) algorithm Random Early Detection (RED). This algorithm typically marks the packets being sent over the network with a probability and based on this probability, the packets are enqueued or dropped. The modified version reads the DSCP marker in the IP header, uses the DSCP to classify the traffic into three classes, and influences the marking probability of each packet based on its class. This is an example of Classful Probability Marking Active Queue Management (CPM AQM).

There are three standard DSCP codes typically used for video streaming (0x32), VoIP (0x46), and data transfer (0x00) [9]. These are representative of our Class 2 (best performance), Class 1 (middle of the road), and Class 0 (best effort) QoS goals.

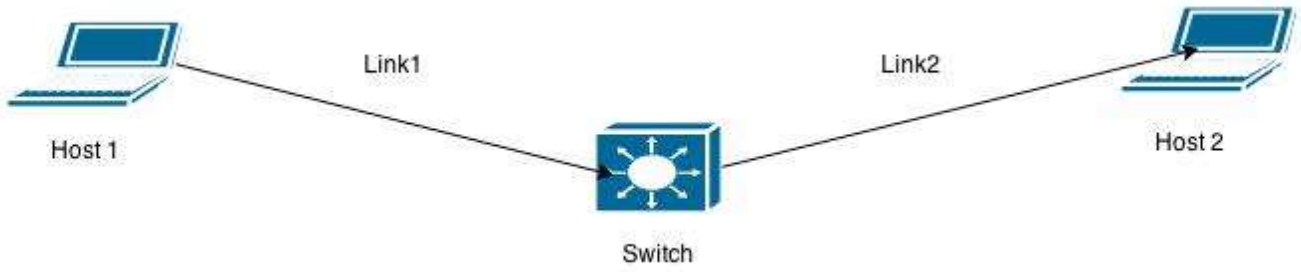
The algorithm is tested on an OVS switch with several hosts connected to it. Each host transmits multiple types of traffic flows, and gathered statistical information is compared to see how the AQM algorithm improves the QoS in a scenario where different types of traffic are competing.

System Description

The components of the network include:

- A switch configured with IP forwarding used as the switching component of the network. The CPM AQM Algorithm is configured to run at the switch.
- A traffic generator - iPerf used to generate UDP traffic at the two hosts connected to the switch, with the DSCP bit marked on the packets.
- A traffic analyzer - Wireshark - used at the output end to measure throughput.
- Exo-GENI is used as the environment for development and testing of this project.

Development and Test Topology



The above network topology is created using Exo-GENI. All the nodes are Ubuntu 14.04 nodes.

- The IP addresses of all the interfaces in the network are as follows:

  Host 1: 192.168.24.21/24

  Switch Host 1 interface (eth2): 192.168.24.22/24

  Switch Host 2 interface (eth1): 192.168.25.23/24

  Host 2: 192.168.25.24/24

- The link bandwidths are:

  Link1: 20 Mbps

  Link2: 10 Mbps

Configuring the switch

The switch being used is an Ubuntu 14.04 node with linux kernel version 3.13. It was configured for IP forwarding and the modified version of the AQM algorithm, that is, CPM RED was attached to it at the input and output interfaces. The steps followed for these configurations were as follows:

- First, the routes for both the hosts were added into the routing table of the switch. The commands used for this are as shown below.

  For Host 1:

  ```
  route add –inet 192.168.24.0 gateway 255.255.255.0 dev eth1
  ```

  For Host 2:

  ```
  route add –inet 192.168.25.0 gateway 255.255.255.0 dev eth2
  ```

- IP forwarding was enabled at the switch using the following command.

  ```
  echo 1 > /proc/sys/net/ipv4/ip_forward
  ```

  This sets up the forwarding mechanism of the switch.

- Next step was configuring the AQM algorithm to run on the switch.

  This project is based on kernel module development. The kernel module which contains the modified CPM AQM algorithm was first inserted into the kernel using the `insmod` command.

  The source code for this module is available in the following github repository: https://www.github.com/jcinma/AQM
  It is the `red_csc573.c` file under the RED directory. On running the make command, we can obtain the `red_csc573.ko` kernel module which can be inserted into the kernel using the command

```
insmod red_csc573.ko
```

The switch is configured with this AQM using the traffic control, that is, the ‘tc’ tool present in the iproute2 package. This package can be downloaded from the following link:

```
https://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git
```

The traffic control subsystem, that is the ‘tc’ command, was used to attach the ‘Queueing discipline’on the egress port of the switch, that is the interface that connects to Host 2, if we are sending traffic from Host 1 to Host 2, which would add the CPM AQM algorithm (CPM RED) for traffic control. It was also used to set the input and output rates of the packets on a particular interface. It basically allows us to have granular control over the various queueing mechanisms of a networked device. [10] http://tldp.org/HOWTO/Traffic-Control-HOWTO/overview.html

CPM RED: For the modified version of the RED algorithm, the tc command was used to attach the AQM algorithm to the queue and to specify the parameters specific to the RED algorithm, that is limit, min, max, average packet size, burst, probability and bandwidth.

The tbf (Token bucket filter) was configured at the root of the switch node at the 'egress' interface i.e. the interface connecting the switch to Host 2.

This qdisc is built on tokens and buckets. It simply shapes traffic transmitted on an interface. To limit the speed at which packets will be dequeued from a particular interface, the TBF qdisc is the perfect solution. It simply slows down transmitted traffic to the specified rate. Packets are only transmitted if there are sufficient tokens available. Otherwise, packets are deferred. Delaying packets in this fashion will introduce an artificial latency into the packet's round trip time.

The CPM RED qdisc was configured as a child of the 'tbf' qdisc.

The commands used to configure the switch in this way are as follows. [6][7]

For Token Bucket Filter:

```
./tc qdisc add dev eth1 root handle 1: tbf rate 8mbit
burst 10k limit 200k
```

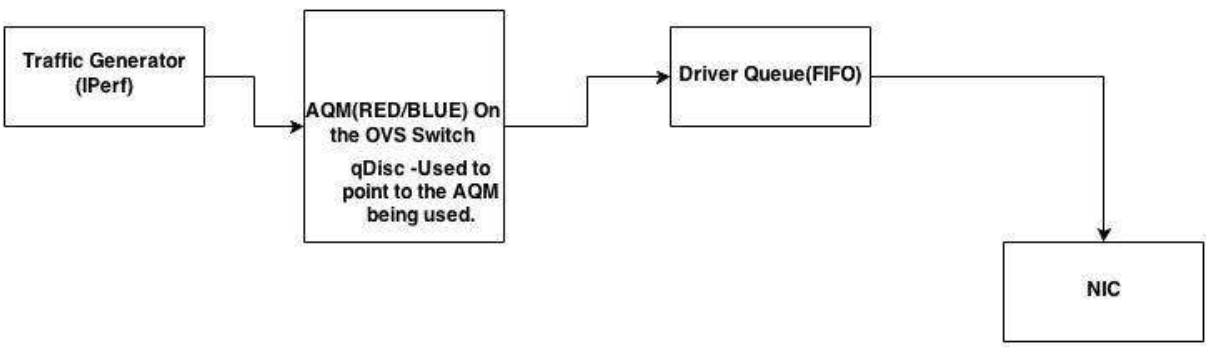
For CPM RED:

```
./tc qdisc add dev eth1 parent 1: red_csc573 limit 400kb
min 16kb max 32kb avgpkt 1000 burst 17 probability 0.2
bandwidth 8mbit
```

Configuration of parameters for RED:

- limit – The maximum length of the queue.
- min – The minimum threshold value of queue length. This value is ideally set as the link bandwidth multiplied by the maximum acceptable throughput.
- max – The maximum threshold value of queue length. This is twice the minimum threshold value, or on slow links, sometimes upto four times the minimum.
- avpkt – The average packet size.
- burst – The maximum size of burst allowed by the algorithm. A burst of (2*min+max)/(3*avpkt) should be efficient.
- probability – The maximum probability that is used in the RED algorithm.

Packet Flow Diagram



Since this project focuses on the AQM algorithm in very specific terms, it is useful to show the context. In the switch, a packet arrives (from a host running iperf) to a port. Because of

the tc configuration, this packet runs through the test CPM AQM algorithm. This algorithm classifies the packet based on the DSCP and decides whether the packet should be dropped or enqueued. If it is enqueued, it is placed in the driver queue[4]. Once in the driver queue, the NIC calls the dequeue function and transmits the packet.

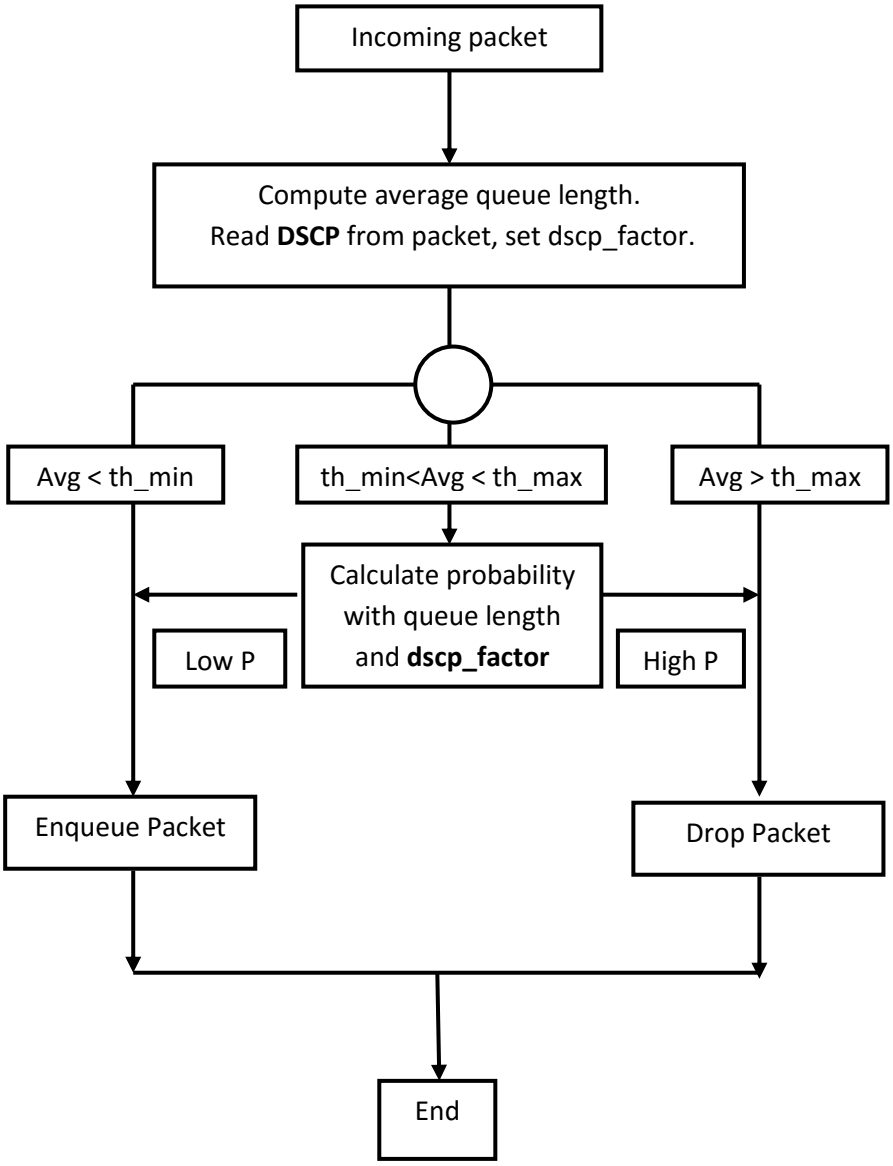
Flow Chart

A flow chart explaining how the CPM RED algorithm works can be shown as below.

avg – Average queue length

th_min – Minimum threshold value for RED

th_max – Maximum threshold value for RED



Flow Chart for CPM RED algorithm

When a packet is coming in, the average queue length at this moment is calculated, and the DSCP bits are read from the packet. The value of dscp_factor, which can also be seen as the variable class_coeff in subsequent implementations, is set according to the different values of the DSCP bits. Three dscp_factors are used in this project, dscp_factor = 0.5 is for the best performance traffic, dscp_factor = 0.7 is for the middle in the road traffic, and dscp_factor = 0.9 is for the best effort traffic. Then, according to the calculated probability, a packet will be dropped if P is too high, and it will be queued if P is low enough.

Low Level Design

The low level design demonstrates in pseudocode how the packets generated by the traffic generator and marked using DSCP is handled by the AQM algorithm. A packet enters one of the ports of the switch. Each port will be configured with tc as explained above.

The AQM algorithm controls three elements: enqueue(), dequeue(), and drop(). The CPM modification will affect the behavior of enqueue() and drop(). The dequeue() function will be handled in the default manner by the algorithm.

The CPM AQM algorithm will enqueue as many packets as possible to a FIFO queue called the Driver Queue which contains descriptors that point to socket buffers (SKB's) and will dequeue them for transmission as required.

RED Implementation with Integrated Classful Probability Marking

In theory, RED defines two thresholds in the queue. If the queue remains below the lower threshold, any new packets will be added to the queue. If the queue rises above the lower threshold, some packets will be dropped according to a probability test. If the queue is full beyond the upper limit, all incoming packets will be dropped [2].

Added Function

There is one function that is added to the RED implementation that helps integrate the concept of DSCP with the AQM. This function is used to read the DSCP field from the IP header and set the `class_coeff` variable. The `class_coeff` variable is being used to carry the multiplication factor that will be used to manipulate the marking probability of the packets.

```
read_and_interpret_DSCP(*packet) {

      char offset = 1; //DSCP starts on 9th bit in IP header
      *dscp = *packet + offset:
```

```
        switch(*dscp){
                case CLASS2 //compare with bit mask
                        class_coeff = 0.5;    //Best performance traffic
                case CLASS1
                        class_coeff = o.7;    //Middle of the road traffic
                case CLASS0
                        class_coeff = 0.9;   //Best effort traffic
        }
}
```

As seen above, the different classes of traffic have been given a `class_coeff` factor based on the QoS they demand. At a later stage in the algorithm, this `class_coeff` factor is used to manipulate the marking probability, based on which it will be decided whether the packet is to be dropped or enqueued. The class coefficients are hard-coded for development convenience, but may be tweaked in testing.

Modifying RED to consider classes in its marking probability requires a thorough knowledge of RED's nested functions. The class_coeff variable is passed through these nested functions and applied only to the probability function itself.

In order to carry this class coefficient, a new line is created in the `red_sched_data` struct. The existing C code for the RED implementation as seen in the linux kernel is modified to add the coefficient as follows:

```
struct red_sched_data {
        u32                limit;             /* HARD maximal queue length */
        unsigned char      flags;
        float              class_coeff;       //add class coefficient for dscp
        struct timer_list adapt_timer;
        struct red_parms  parms;
        struct red_vars   vars;
        struct red_stats  stats;
        struct Qdisc     *qdisc;
};
```

Then, the `red_enqueue()` function will read the DSCP bit from the packet, use a lookup table to assign class, and then `red_action()` and `red_mark_probability()` will have an extra argument added for the `class_coeff`.

`red_mark_probability()` is the single line function that determines whether a packet will be dropped or not. First, it takes the current average queue length `qavg` and compares it to the minimum average queue length `qth_min` (essentially a low constant queue length). This number is bit-shifted by `Wlog` (a normalizing number) and multiplied by the number of packets that have been compared to an un-altered random number `qR`. If this result is greater than the random number, the packet will be dropped.

red_mark_probability:

```
red_mark_probability(red_parms *p, qavg, *packet, class_coeff)
{
      P_mark = !(((qavg - p->qth_min) >> p->Wlog) * p->qcount*class_coeff < p->qR);

      return P_mark;
}
```

The coefficient `class_coeff` is multiplied by the number being compared to the random number. Thus, based on the values we have provided for the class coefficients, Class 0 traffic will have a higher probability of being greater than the random number, and Class 2 will have a much lower probability. In this way, it is ensured that when traffic of these three types are competing at the egress where the Active Queue management is being carried out, the one with the higher priority gets precedence over the one with the lower priority.

At the top level is the `red_enqueue()` function. This is where all of the queueing logic is handled. Specifically, there is a case statement to control whether a packet is sent through, automatically dropped (hard dropped), or has a probability evaluated for its drop. The function that controls this logic is the `red_action()` function.

red_enqueue function:

```
red_enqueue(*packet) {

    float class_coeff = read_and_interpret_DSCP(*packet);

    switch (red_action(red_parms *p, qavg, *packet, class_coeff) {
    case RED_DONT_MARK: //enqueue a packet
            break;

    case RED_PROB_MARK: // drop a packet by probability
            goto congestion_drop;

    case RED_HARD_MARK: // drop a packet because q_length exceeds the
threshold
            goto congestion_drop;
}
```

The `red_action()` function will take the red parameters struct, the length of the queue, the packets and the class coefficient as its arguments, and returns the decision of whether to drop the packet or not. The change of probability is implemented in the `red_mark_probability()` function, as explained above.

red_action function:

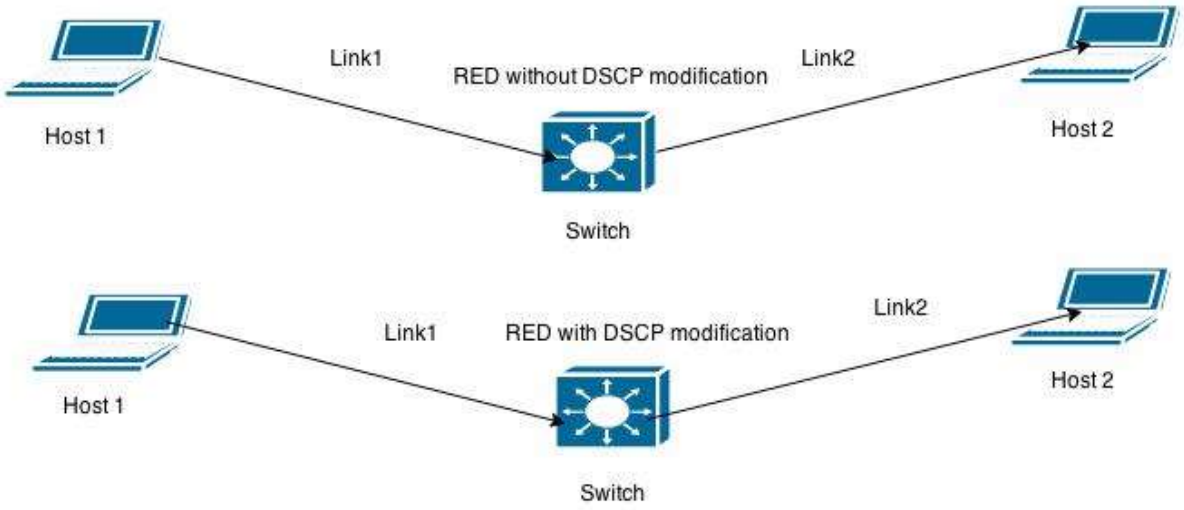
```
red_action(red_parms *p, qavg, *packet, class_coeff)
{
      switch (red_cmp_thresh(p, qavg)) {  //comparing the threshold values
with average queue length qavg
            case RED_BELOW_MIN_THRESH:
                  return RED_DONT_MARK;
```

```
            case RED_BETWEEN_THRESH:
                  if (++p->qcount) {
                  if (red_mark_probability(p, qavg, *packet, class_coeff)){
                              return RED_PROB_MARK;
                        }
                  } else
                         return RED_DONT_MARK;


            case RED_ABOVE_MAX_THRESH:
                  return RED_HARD_MARK;
      }


      return RED_DONT_MARK;
}
```

The `red_action()` function carries out the heavy lifting in RED. It ultimately makes the decision of whether to pass, drop, or mark each packet. When the queue is between thresholds, and the marking probability logic is active, `red_mark_probability()` is calling the shots.

Test Plan

Test Topology



Test Setup

In the figure above, two different networks scenarios are described. The difference between them is the AQM algorithm being run on the switch. In the first network scenario, the switch runs the basic RED algorithm, that is, no modification involving the DSCP bits is made. The second has the modified version of RED, that is, CPM RED, which has the classful probability marking, running on the switch. Thus, the improvement in the performance of the RED algorithm with the DSCP modification over the basic RED algorithm without the DSCP modification is being tested here. The IP addresses and the link bandwidths are the same as mentioned in the system description.

Test Procedure

This test plan includes configuration information for the network topology. The tests that were carried out for both the network network scenarios mentioned above typically involved carrying out the following steps:

1. **Preparing an Exo-GENI slice:**

The first step involved creating the test network topology using Flukes in Exo-GENI. The topology consisted of 2 hosts attached to a switch as shown in the diagrams above.

2. **Configuring the forwarding mechanism on the switch:**
   Next, the switch was configured to add routes for both the hosts in the routing table of the switch, as explained in the system description section above.

3. **Testing the network for connectivity:**
   `ping` was used to check if Host 1 was connected to Host 2.

4. **Installing Iperf and Wireshark on both the switches:**
   Next Iperf and tshark, the CLI version of Wireshark was installed on both the hosts.

5. **Configure the switch with the qdisc using tc:**
   The switch was configured to run the AQM algorithm, either RED or CPM RED. The detailed instructions for doing so are provided in the Demo Instructions section below.

6. **Starting Wireshark capture at Host 2:**
   Wireshark capture was started at Host 2 and the results were written into a dump file.

7. **Sending traffic from Host1 to Host 2:**
   Using Iperf, traffic was sent from Host 1 to Host 2. At Host 2, the iperf servers were set up at three different ports, 5010, 5011 and 5012, for listening to connection requests. The command for this is:
   `iperf3 –s –p 5010 & iperf3 –s –p 5011 & iperf3 –s –p 5012`

   Multiple flows marked with different DSCP markings were sent from Host 1 to Host 2 by using the & between two iperf commands and varying the rate at which each flow is sent according to the test scenario.

   The total bandwidth to be sent on the link was limited to approximately 3 Mb and this bandwidth was divided among the different flows proportional to the percentages

given in the test cases. For example, if all three types of traffic were to be sent, the bandwidth was divided into the flows as 0.9 Mb for each flow, and this bandwidth was specified in the bandwidth parameter of the iperf command.

All the tests were run for 75 seconds each, which is specified in the time parameter of the iperf command.

The iperf command used for generating and sending the traffic for all the three flows is:

```
iperf3 –c 192.168.25.24 –p <port no.> –u –b <bandwidth> –S
<DSCP value> –t 75 –T <ClassN>
```

Here, the options stand for:

-c – Client for iperf followed by the interface address to which it connects

-p – Port number to which to connect

-u – Signifies UDP traffic

-b – Sets the rate at which traffic is being sent

-S – Sets the DSCP bits

-t – Time for which traffic is being sent

-T – To set labels to differentiate the flows

8. **Take the Wireshark dump file to check the throughput:**
   The dump files were analysed using Wireshark and the throughputs for every test case were noted. Here, the total number of packets sent, the total number of packets received and the total number of packets dropped for every flow of traffic is noted down and compared.

Test Cases and Results

The test procedures described above were run on each test network scenario and the following observations were made:

AQM on Network	Expected Result	Observed Result	Conclusion
RED algorithm without Classful Probability Marking.	Baseline result. The throughput for each type of traffic is random, almost equal if equal bandwidth is provided for each type.	The percentage of packets being dropped for every Class of traffic was random, almost equal in some cases, and the throughput was not related to the type of traffic.	All kinds of traffic receive equal treatment at the switch. The basic RED algorithm drops packets randomly, without taking into consideration the type of traffic.
CPM RED algorithm	The traffic with highest priority, that is, Class 2 traffic shows increased throughput as compared to the middle priority Class 1 traffic, which in turn shows increased throughput than the best effort Class 0 traffic.	The percentage of packets being dropped was the least for Class 2 traffic. Class 1 traffic showed lesser percentage of packets dropped than Class 0.	The CPM RED algorithm drops the packets based on their type. That is, the packet marked as Class 2 has a lesser probability of being dropped than Class 1. And the Class 1 packet has a lesser probability of being dropped than Class 0.

The specific test cases and the test results are as below:

- **Test results for testing with basic RED algorithm, without Classful Probability Marking, running on the switch**

Test ID	Test Scenario	Class of Traffic	Total no. of packets sent	No. of packets transmitted	No. of packets dropped	% of packets dropped
1	Sending all three types of traffic over the link (equal bandwidth for each)	Class 0	1197	927	270	22.56
		Class 1	1199	890	309	25.77
		Class 2	1199	895	304	25.35
2	Sending only Class 0 and Class 1 traffic (equal bandwidth for each)	Class 0	1798	1373	425	24
		Class 1	1798	1311	487	27
3	Sending only Class 0 and Class 2 traffic (equal bandwidth for each)	Class 0	1798	1362	436	24
		Class 2	1798	1295	503	28
4	Sending only Class 1 and Class 2 traffic (equal bandwidth for each)	Class 1	1798	1359	439	24.4
		Class 2	1798	1290	508	28.2
5	Sending 25% Class 1, 25% Class 2 and 50% Class 0 traffic	Class 0	1798	1289	510	28.36
		Class 1	899	689	210	23.35
		Class 2	899	680	219	24.36

- **Test results for testing with CPM RED algorithm, with the Classful Probability Marking, running on the switch**

Test ID	Test Scenario	Class of Traffic	Total no. of packets sent	No. of packets transmitted	No. of packets dropped	% of packets dropped
6	Sending all three types of traffic over the link (equal bandwidth for each)	Class 0	1199	802	397	33.11
		Class 1	1199	934	265	22.10
		Class 2	1199	1199	0	0
7	Sending only Class 0 and Class 1 traffic (equal bandwidth for each)	Class 0	1797	1290	507	28.21
		Class 1	1798	1462	336	18.69
8	Sending only Class 0 and Class 2 traffic (equal bandwidth for each)	Class 0	1798	1354	444	25
		Class 2	1798	1474	324	18
9	Sending only Class 1 and Class 2 traffic (equal bandwidth for each)	Class 1	1798	1404	394	22
		Class 2	1798	1528	270	15
10	Sending 25% Class 1, 25% Class 2 and 50% Class 0 traffic	Class 0	1798	1303	495	28
		Class 1	899	738	161	18
		Class 2	899	764	135	15

Demonstration

Demo Network topology

The same network topology used for testing will be used for the demo as well. The Demo will involve sending different types of traffic simultaneously over the same links from Host 1 to Host 2 via the switch in both the scenarios. Wireshark is used to capture the packets being received at Host 2 in a dump file. The two dump files will be analyzed using Wireshark. This analysis will involve applying various filters on the packet dump to see the number of packets of each type of traffic being received and thus analyze which type of traffic shows an improved throughput.

Demo Instructions

The step-by-step demo instructions are as follows:

Step 1: Set up the topology using Flukes on Exo-GENI

The network configurations are the same as given in the system description.

Step 2: Configure the switch for IP forwarding

The switch is then configured to send traffic from Host 1 to Host 2 and vice-versa. It is done using the following commands:

For Host 1:

```
route add –inet 192.168.24.0 gateway 255.255.255.0 dev
eth1
```

For Host 2:

```
route add –inet 192.168.25.0 gateway 255.255.255.0 dev
eth2
```

To enable IP forwarding:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Step 3: Install Iperf v3 on both the Hosts

Iperf v3 is installed on both the hosts using the following instructions.

```
wget http://downloads.es.net/pub/iperf/iperf-3.0.6.tar.gz
```

```
tar zxvf iperf-3.0.6.tar.gz

cd iperf-3.0.6

./configure

make

make install
```

step 4: Install tshark (Wireshark) on both the Hosts

tshark, the command line version of Wireshark is installed on both the hosts for packet analysis.

```
sudo apt-get install tshark wireshark

/etc/init.d/buddyweb restart
```

Step 5: Configure the switch with basic RED algorithm

The 'tc' command is used to attach the basic RED qdisc to the interface that connects to Host 2. Also, the TBF algorithm is used to limit the rate of the egress traffic. The commands used are:

For Token Bucket Filter (to limit the rate of the egress traffic):

```
tc qdisc add dev eth1 root handle 1: tbf rate 8mbit burst 10k limit 200k
```

For RED:

```
tc qdisc add dev eth1 parent 1: red limit 400kb min 16kb max 32kb avgpkt 1000 burst 17 probability 0.2 bandwidth 8mbit
```

Step 6: Login to Host 2 and start the Iperf servers

The Iperf servers are started at three ports on Host 2 that will listen for traffic cming from Host 1. Three ports are used since we want to send three flows of udp traffic, all on the same link, and each marked with a different DSCP marking. The command used at this step is:

```
iperf3 –s –p 5010 & iperf3 –s –p 5011 & iperf3 –s –p 5012
```

Step 7: Start the wireshark capture at Host 2

Using tshark, the packets being received at Host 2 at interface eth1 are captured into a dump file. The command used for this is as follows:

```
tshark –i eth1 –w /capture1.cap
```

Step 8: Start sending traffic from Host 1 to Host 2

Iperf is then used to send three flows of traffic from Host 1 to Host 2. Each of these flows will be marked by a different DSCP, one will be 0x32 for Class 2 traffic, one will be 0x46 for Class 1 traffic, and last will be 0x00 for Class 0 traffic.

```
iperf3 –c 192.168.25.24 –p 5010 –u –b 0.9m –S 0x32 –t 75 –T
Class2 & iperf3 –c 192.168.25.24 –p 5011 –u –b 0.9m –S 0x46 –t
75 –T Class1 & iperf3 –c 192.168.25.24 –p 5012 –u –b 0.9m –S
0x00 –t 75 –T Class0
```

Step 9: Configure the Switch with CPM RED

The next step involves using tc to configure the modified version of RED, that is, CPM RED and attach it to the egress port of the switch. For that, first the source code for it is obtained by downloading it from the github repository.

```
git clone https://www.github.com/jcinma/AQM
```

The module containing the CPM AQM algorithm is then inserted into the kernel. The steps for doing this are:

```
cd AQM

cd RED

make

insmod red_csc573.ko
```

The iproute2 package is also downloaded in order to attach the modified qdisc to the switch.

```
git clone
git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iprou
te2.git
```

To attach the CPM RED qdisc to the switch interface, the tc command from the iproute2 package is used.

```
cd iproute2/tc
```

For Token Bucket Filter (to limit the rate of the egress traffic):

```
./tc qdisc add dev eth1 root handle 1: tbf rate 8mbit burst
10k limit 200k
```

For CPM RED:

```
./tc qdisc add dev eth1 parent 1: red_csc573 limit 400kb min
16kb max 32kb avgpkt 1000 burst 17 probability 0.2 bandwidth
8mbit
```

Step 10: Repeat steps 6 to 8 for all the test cases mentioned in the Demo Test Cases

Step 11: Analyse the results

The analysis of all the capture files using Wireshark is done on a windows machine by using FTP to transfer these files. This is done for the convenience of a graphical user interface which helps better analyse the packets. Once the files have been transferred, we analyse each file. The analysis is done by applying the following filters.

To count the number of packets marked by DSCP 0x32 (Class 2) received:

```
ip.proto == UDP && ip.dsfield == 0x32
```

To count the number of packets marked by DSCP 0x46 (Class 1) received:

```
ip.proto == UDP && ip.dsfield == 0x46
```

To count the number of packets marked by DSCP 0x00 (Class 0) received:

```
ip.proto == UDP && ip.dsfield == 0x00
```

Using these filters, the actual throughput, that is the number of packets received at Host 2, are noted for each type of traffic. The throughput obtained when Classful probability Marking was not used is compared to that obtained with the CPM RED running at the switch. It will be seen that for CPM RED, the number of packets transferred is the highest for Class 2, lower for Class 1 and the lowest for Class 0. The demo test cases and expected results are summarized in the table below.

Demo Test Cases and Test Results

The specific Demo Test Cases that will be carried out during the Demo are as follows:

Demo ID	Demo Test Case	Expected Observation	Conclusion
RED running at the switch (without Classful Probability Marking)			
1	Send all three types of traffic over the link with equal bandwidth for each flow	The percentage of packets being dropped is almost	The basic RED algorithm treats all three kinds of traffic

	using Iperf and analyse the results using Wireshark	equal for all three types of traffic.	equally and hence all the flows show almost equal throughput.
CPM RED running at the switch (with Classful Probability Marking)			
2	Send all three types of traffic over the link with equal bandwidth for each flow using Iperf and analyse the results using Wireshark	The percentage of packets dropped is the least for Class 2 traffic, it is relatively higher for Class 1 traffic and it is the highest for Class 0 traffic.	The CPM RED algorithm makes sure that the throughput of the traffic is proportional to its priority. That is, Class 2 traffic, which is the best performance traffic has the maximum throughput, whereas Class 1 traffic has medium throughput and Class 0 traffic has the least throughput.

Self-Study Plan

During the course of this project, a number of concepts were studied and researched about. The learnings out of all the study are as follows:

Random Early Detection (RED) algorithm:

In the conventional tail drop algorithm, a router or other network component buffers as many packets as it can and simply drops the ones it can't buffer. If buffers are constantly full, the network is congested. Tail drop distributes buffer space unfairly among traffic flows. Tail drop can also lead to TCP global synchronization as all TCP connections "hold back" simultaneously, and then step forward simultaneously. Networks become under-utilized and flooded by turns. RED addresses these issues.

RED monitors the average queue size and drops (or marks when used in conjunction with ECN) packets based on statistical probabilities. If the buffer is almost empty, all incoming packets are accepted. As the queue grows, the probability for dropping an incoming packet grows too. When the buffer is full, the probability has reached 1 and all incoming packets are dropped.

RED is more fair than tail drop, in the sense that it does not possess a bias against bursty traffic that uses only a small portion of the bandwidth. The more a host transmits, the more

likely it is that its packets are dropped as the probability of a host's packet being dropped is proportional to the amount of data it has in a queue. Early detection helps avoid TCP global synchronization.

BLUE algorithm:

Blue is a scheduling discipline for the network scheduler developed by graduate student Wu-chang Feng for Professor Kang G. Shin at the University of Michigan and others at the Thomas J. Watson Research Center of IBM in 1999. Like random early detection (RED), it operates by randomly dropping or marking packet with explicit congestion notification mark before the transmit buffer of the network interface controller overflows. In contrary to RED, however, it requires little or no tuning on the part of the network administrator. A Blue queue maintains a drop/mark probability *p*, and drops/marks packets with probability *p* as they enter the queue. Whenever the queue overflows, *p* is increased by a small constant p_d, and whenever the queue is empty, *p* is decreased by a constant $p_i < p_d$.

If the mix of traffic on the interface does not change, *p* will slowly converge to a value that keeps the queue within its bounds with full link utilization.

Iperf:

Iperf is a commonly used network testing tool that can create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams and measure the throughput of a network that is carrying them.

Iperf allows the user to set various parameters that can be used for testing a network, or alternatively for optimizing or tuning a network. Iperf has a client and server functionality, and can measure the throughput between the two ends, either unidirectionally or bi-directionally. It is open-source software and runs on various platforms including Linux, Unix and Windows.

Typical Iperf output contains a time-stamped report of the amount of data transferred and the throughput measured.

Iperf is significant as it is a cross-platform tool that can be run over any network and output standardized performance measurements. Thus it can be used for comparison of both wired and wireless networking equipment and technologies. Since it is also open source, the measurement methodology can be scrutinized by the user as well.

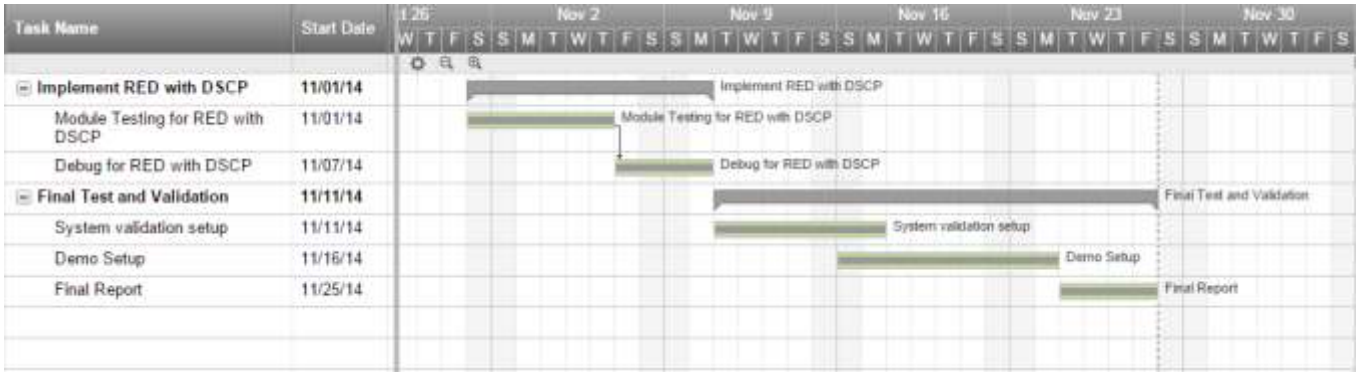
Traffic Control (TC):

Linux has a sophisticated system for bandwidth provisioning called Traffic Control. This system supports various method for classifying, prioritizing, sharing, and limiting both inbound and outbound traffic.

Iproute2 consists of a variety of utilities for controlling TCP / IP networking and traffic control in Linux. It is actively maintained and frequently updated, and now considered the preferred package for modern network technologies which includes important network tools such as ip and tc, used for IPv4 and IPv6 configuration and traffic control. It currently replaces the outdated Net-Tools package encompassing commands such as ifconfig, route, arp, etc, which are now known to be mostly inadequate as they provide limited features.

Timeline

The timeline followed for the course of the project is as follows.



Conclusion:

In this work, the modified RED AQM algorithm based on the DSCP bits will be implemented. With this novel algorithm, the dropping/marking probability will not only rely on the traditional parameters, such as the length of the queue to update the probability, but also on the incoming packets, and traffic requiring better throughput will have a higher priority than other traffic streams. Thus, this is Classful Probability Marking Active Queue Management.

These AQM algorithms should be able to handle congestion that would cause problems in a limited pipe or low bandwidth situation. What is more, these algorithms will only involve policing a single queue and considering the class of traffic when deciding to allow or deny entry to the queue, so it will not experience bloat in an extensive and iterative implementation like other traffic control solutions. The implementation will be completed as mentioned in the timeline, and validation will be performed on the EXO-GENI platform with the topology structure proposed in this document.

References

[1] Bernet, Yoram. "The complementary roles of RSVP and differentiated services in the full-service QoS network." *Communications Magazine, IEEE* 38.2 (2000): 154-162.

[2] S. Floyd and V. Jacobson, "*Random early detection gateways for congestion avoidance*," IEEE/ACM Trans. Networking, vol. 1, pp. 397-413,Aug. 1993.

[3] W. Feng et al., "*The BLUE active queue management algorithms*," IEEE/ACM Trans. Networking, vol. 10, pp. 513-528, Apr. 2002.

[4] http://www.coverfire.com/articles/queueing-in-the-linux-network-stack/ ("Queueing in the linux network stack." *Dan Siemon*.)

[5] http://www.tldp.org/HOWTO/html_single/Traffic-Control-HOWTO/#s-iproute2("Traffic Control HOWTO." *Traffic Control HOWTO*.)

[6] http://www.protocog.com/trgen.html ("Protocol Testing - Theory, Test Suites, Tools, Formal Methods." *Protocol Testing - Theory, Test Suites, Tools, Formal Methods*.)

[7] http://wiki.openwrt.org/doc/howto/packet.scheduler/packet.scheduler("OpenWrt." *Network Traffic Control*.)

[8] http://lartc.org/manpages/ ("Linux Advanced Routing & Traffic Control." *Manpages*.)

[9] "RFC 4594 - Configuration Guidelines for DiffServ Service Classes." *RFC 4594 - Configuration Guidelines for DiffServ Service Classes*.