Advanced Computer Networks (CS G525)
First Semester 2018-2019
Lab Session #6
Topic: Custom Congestion Control and Queueing Algorithm

Objectives:

- **To modify existing congestion control algorithm.**
- **To get familiar with Queuing Algorithm in ns3.**

Module 1: Implement Custom Congestion Algorithm:

In the last lab, we explored ns3 TCP and UDP application and also plotted the variation of congestion window with time.

The congestion control algorithm used for all the tasks were not specified in the code but by default, ns3 was using TCP New Reno as the congestion control algorithm. Explicitly this can be specified using:

```
Config::SetDefault ("ns3::TcpL4Protocol::SocketType",StringValue("ns3::TcpNewReno"));
```

Note: There are some other methods as well to achieve this task but for this lab sheet above method should be enough.

ns3 provides different variants of congestion control algorithms called “TCP flavours”. Some of them are as follows:

- ns3::TcpBic
- ns3::TcpHighSpeed
- ns3::TcpVeno
- ns3::WestWood

More Details at https://www.nsnam.org/doxygen/group__congestion_ops.html#details

We can specify any of the variants similarly as we did for **TcpNewReno**.

Today, we will discuss how we can change the codebase to add our own congestion control algorithm. The method we will discuss focuses more on how to create a custom file and link that to ns3 such that it can be used in simulations. It is not a perfect implementation of developing a congestion control from scratch but a head start in that direction. For building any such TCP variant you can refer https://www.nsnam.org/docs/models/html/tcp.html after you are comfortable with this method.

Steps to be followed:

Please be very careful while following the instructions as errors will affect the complete ns3 module on your machine while compiling.

1. Implementation of all the congestion control algorithm are stored at:
   ~/ns-allinone-3.28.1/ns-3.28.1/src/internet/model/
   Switch to this directory.
2. We will use **tcp-highspeed.cc** and **tcp-highspeed.h** as the base files for the task.
3. Make a copy of **tcp-highspeed.cc** and rename it to **tcp-custom.cc**
4. Make a copy of **tcp-highspeed.h** and rename it to **tcp-custom.h**
5. Now open **tcp-custom.cc** and replace all the instances of **TcpHighSpeed** with **TcpCustom.** Replace the header file "**tcp-highspeed.h**" with "**tcp-custom.h**"
6. Open **tcp-custom.h** and replace all instances of **TcpHighSpeed** with **TcpCustom.**
7. Now for linking the file with ns3. Switch to the following directory and open **wscript** file:
   ~/ns-allinone-3.28.1/ns-3.28.1/src/internet/
8. Search for '**model/tcp-highspeed.cc**' and add '**model/tcp-custom.cc**' below that, place comma operator properly.
9. Search for '**model/tcp-highspeed.h**' and add '**model/tcp-custom.h**' below that, place comma operator properly.

After correctly completing the above steps you should be able to add this TCP variant in your simulation code.

Open myalgo.cc provided with the lab sheet and set TcpCustom as the congestion control algorithm using Config::SetDefault.

Run ***myalgo file*** using command **./waf --run myalgo** after placing it in the scratch directory. First time compilation may take a while to complete.

By completing all the above steps now you have made a copy of **tcp-highspeed** congestion control algorithm but it can be attached to the simulation file by the name we have specified i.e**. tcp-custom**.

Tasks:

1. Plot the congestion window w.r.t. time for the trace generated using myalgo.cc and TcpCustom as a variant (Tcp Custom is still same as TcpHighSpeed).

2. Run the file with the following command

```
./waf --run myalgo NS_LOG=TcpCustom=level_info
```

From where are you getting extra output? Try to print the coeffA variable in the log output as well.

3. As mentioned earlier **TcpCustom** is just a copy of **TcpHighSpeed** which is implemented as follows.

The **TcpHighSpeed** makes the **cWnd** grow during congestion avoidance using the equation (i). The function **a()** is calculated using lookup table taken from RFC 3649).

$$cWnd = cWnd + \frac{a(cWnd)}{cWnd} \qquad \text{(i)}$$

For each congestion event, the slow start threshold is decreased by a value that depends on the size of the slow start threshold itself. The lookup table for the function **b()** is taken from the same RFC.

$$cWnd = (1 - b(cWnd)) \cdot cWnd \qquad \text{(ii)}$$

Modify (i) such that **cWnd** increases using equation: **cWnd = cWnd + 1/cWnd**

And Modify (ii) such that **cWnd** decreases using equation: **cWnd = 0.5 * cWnd**

Make these changes in tcp-custom.cc and finally plot the congestion window.

Module 2: Router Queuing Algorithms

An Internet router maintains one queue for its each interface, that hold packets scheduled to forward on that interface. Historically, such queues use a FIFO+ DropTail in which a packet is put onto the queue until it becomes full (measured in packets or in bytes), and dropped otherwise.

Active queue algorithms drop or mark packets before the queue is full. Typically, they operate by maintaining one or more drop/mark probabilities, and probabilistically dropping or marking packets even when the queue is not full with the goal of reducing network congestion or improving end-to-end latency

TASK: Understanding the importance of good parameters in RED algorithm

Random Early Detection (RED) is a queue discipline that aims to provide early signals to transport protocol congestion control (e.g. TCP) that congestion is imminent, so they back off their rate gracefully rather than with a bunch of tail-drop losses in FIFO + DropTail (possibly incurring TCP timeout).

The **RED** queue disc does not require packet filters, uses a single queue. If not provided by the user, a **DropTail** queue (e.g., FIFO) operating in the same mode (packet or byte) as the queue disc and having a size equal to the **RED MaxSize** attribute is created. Otherwise, the capacity of the queue disc is determined by the capacity of the internal queue provided by the user.

Some attributes that ns3 provides for **RED** is as follows:

MeanPktSize: Average of packet size.

IdlePktSize: Average packet size used during idle times.

Wait: True for waiting between dropped packets.

Gentle: True to increases dropping probability slowly when average queue exceeds **maxthresh**.

MinTh: Minimum average length threshold in packets/bytes.

MaxTh: Maximum average length threshold in packets/bytes.

More attributes can be found on https://www.nsnam.org/doxygen/classns3_1_1_red_queue_disc.html

Instructions to execute the file:

Running basic RED algorithm queue and understanding the algorithm:

Get the provided **myred-tests.cc** file in the scratch directory

$cd ~/ns-allinone-3.28.1/ns-3.28.1

~/ns-allinone-3.28.1/ns-3.28.1 $ ./waf --run myred-tests

Note: You should see some queue event logs initially followed by the **RED Queue stats** which is of interest to us which shows the packet drops and other relevant information.

It should be something like below:

```
*** RED stats from Node 2 queue disc ***

Packets/Bytes received: 2253 / 1990950
Packets/Bytes enqueued: 2063 / 1875906
Packets/Bytes dequeued: 2063 / 1875906
Packets/Bytes requeued: 0 / 0
Packets/Bytes dropped: 190 / 115044
Packets/Bytes dropped before enqueue: 190 / 115044
  Forced drop: 108 / 58700
  Unforced drop: 82 / 56344
Packets/Bytes dropped after dequeue: 0 / 0
Packets/Bytes sent: 2063 / 1875906
Packets/Bytes marked: 0 / 0

*** RED stats from Node 3 queue disc ***

Packets/Bytes received: 2055 / 116844
Packets/Bytes enqueued: 2055 / 116844
Packets/Bytes dequeued: 2055 / 116844
Packets/Bytes requeued: 0 / 0
Packets/Bytes dropped: 0 / 0
Packets/Bytes dropped before enqueue: 0 / 0
Packets/Bytes dropped after dequeue: 0 / 0
Packets/Bytes sent: 2055 / 116844
Packets/Bytes marked: 0 / 0
```

Code overview of myred-tests.cc

Our topology of interest is a “dumb-bell” topology that we had in the previous lab and which you can find in the **myred-tests.cc** file as comments at the top.

```
/** Network topology
 *
 *    10Mb/s, 2ms                          10Mb/s, 4ms
 * n0--------------|                      |---------------n4
 *                 |   1.5Mbps/s, 20ms    |
 *                n2--------------------n3
 *    10Mb/s, 3ms  |                      |    10Mb/s, 5ms
 * n1--------------|                      |---------------n5
 *
 *
 */
```

We will attach **RED** queues to both the bottleneck link nodes n2 and n3. While FIFO queues will be attached to the nodes n0, n1, n4, n5.

Attaching **RED** queues is done using the below code snippet:

```
TrafficControlHelper tchRed;
 tchRed.SetRootQueueDisc ("ns3::RedQueueDisc", "LinkBandwidth",
StringValue (redLinkDataRate),"LinkDelay", StringValue
(redLinkDelay));
QueueDiscContainer queueDiscs = tchRed.Install (devn2n3);
```

devn2n3 corresponds to the link between nodes 2 and 3 as mentioned above.
TrafficControlHelper class is used to attach queues to **netdevices** as shown above, we are also providing the link delay and bandwidth that we expect the bottleneck to have.

Note: *By default, the **InternetStackHelper** aggregates a **TrafficControlLayer** object to every node. When invoked to assign an IPv{4,6} address to a device, the IPv{4,6} **AddressHelper**, besides creating a IPv{4,6} Interface, also installs the default queue, **PfifoFastQueueDisc**, on the device, unless a queue disc has been already installed. Thus, devices get the default queue disc installed even if they are added to the node after the Internet stack has been installed on the node.*
To install a queue disc other than the default one, it is necessary to install such queue disc before an IP address is assigned to the device.

Other nodes have FIFO queues as mentioned above and they are used as applied as shown below:

```
TrafficControlHelper tchPfifo;
uint16_t handle = tchPfifo.SetRootQueueDisc("ns3::PfifoFastQueueDisc");
tchPfifo.AddInternalQueues (handle, 3, "ns3::DropTailQueue", "MaxSize",
StringValue ("1000p"));
p2p.SetQueue ("ns3::DropTailQueue");
p2p.SetDeviceAttribute ("DataRate", StringValue ("10Mbps"));
p2p.SetChannelAttribute ("Delay", StringValue ("2ms"));
NetDeviceContainer devn0n2 = p2p.Install (n0n2);
tchPfifo.Install (devn0n2);
```

PfifoFastQueueDisc is a standard/default queue type provided by Linux. As you can see, it will have 3 sub queues (from the third line) that will be processed in a FIFO manner. The 3 queues normally do have priority based on the TOS (Type of Service) bits of the traffic entering the queue.
Higher priority traffic will be processed first always. There is no other logic in this thus making it one of the most simple and naive queues available.

RED Queue parameters are set using the below code snippet:

```
Config::SetDefault ("ns3::RedQueueDisc::MaxSize", StringValue ("1000p"));
```

This line sets the maximum queue size to 1000 packets. Other such parameters can be found in the code and more details can be found in the link above.

On the application layer, we are using an **On Off application** that we are defining using the below code. Basically it will send packets at particular instants of time and it will not send anything at other instants of time. The time instants are random and are decided by the number of packets that need to be sent and the sending rate.

```
   OnOffHelper clientHelper1 ("ns3::TcpSocketFactory", Address ());
      clientHelper1.SetAttribute ("OnTime", StringValue
("ns3::ConstantRandomVariable[Constant=1]"));
      clientHelper1.SetAttribute ("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));
      clientHelper1.SetAttribute ("DataRate", DataRateValue (DataRate
("10Mb/s")));
      clientHelper1.SetAttribute ("PacketSize", UintegerValue (1000));
```

We are also specifying the packets to be sent (1 or 0) at on and off times respectively.

Next, we will run the same code as shown below:

~/ns-allinone-3.28.1/ns-3.28.1$ ./waf --run "myred-tests --testNumber=3"

Note: The value 3 will be set in "**redTest**" variable in the code.

You should see the below output:

```
*** RED stats from Node 2 queue disc ***

Packets/Bytes received: 2614 / 2187198
Packets/Bytes enqueued: 2126 / 1865922
Packets/Bytes dequeued: 2126 / 1865922
Packets/Bytes requeued: 0 / 0
Packets/Bytes dropped: 488 / 321276
Packets/Bytes dropped before enqueue: 488 / 321276
  Forced drop: 344 / 223334
  Unforced drop: 144 / 97942
Packets/Bytes dropped after dequeue: 0 / 0
Packets/Bytes sent: 2126 / 1865922
Packets/Bytes marked: 0 / 0

*** RED stats from Node 3 queue disc ***

Packets/Bytes received: 2124 / 128280
Packets/Bytes enqueued: 2124 / 128280
Packets/Bytes dequeued: 2124 / 128280
Packets/Bytes requeued: 0 / 0
Packets/Bytes dropped: 0 / 0
Packets/Bytes dropped before enqueue: 0 / 0
Packets/Bytes dropped after dequeue: 0 / 0
Packets/Bytes sent: 2124 / 128280
Packets/Bytes marked: 0 / 0
```

Compared to our previous run, what difference do you see in the Red Queue Stats?
Hint: Check the dropped Packets at Node 2.

Exercise: **Identify the cause for increase in packet drops in the second case by going through the code. (The obvious reason is the name of this task!). What is changed by setting the *redTest* in the code?**

TASK: Setting up Adaptive Random Early Detection (ARED) for auto-tuning of parameters.

ARED is a variant of **RED** with two main features: (i) automatically sets **Queue weight**, **MinTh** and **MaxTh** and (ii) adapts maximum drop probability. The model in ns-3 contains implementation of both these features.

Simulating ARED:

The script to be used to simulate ARED is by modifying the file provided to you i.e. **myred-tests.cc**
To switch on ARED algorithm, the attribute ARED must be set to true, as done in **src/traffic-control/examples/adaptive-red-tests.cc:**
Config::SetDefault ("ns3::RedQueueDisc::ARED", BooleanValue (true));

Setting ARED to **true implicitly configures both**:

1. Automatic setting of Queue weight, **MinTh** and **MaxTh**
2. Adapting **AdaptMaxP**.

To configure (i); **Queue weight**, **MinTh** and **MaxTh**, all must be set to 0, as done in **src/traffic-control/examples/adaptive-red-tests.cc** (change existing values to zeroes)

Config::SetDefault ("ns3::RedQueueDisc::QW", DoubleValue(0.0);
Config::SetDefault ("ns3::RedQueueDisc::MinTh", DoubleValue(0));
Config::SetDefault ("ns3::RedQueueDisc::MaxTh", DoubleValue(0));

To configure (ii)
AdaptMaxP must be set to true, as done in **src/traffic-control/examples/adaptive-red-tests.cc**

Config::SetDefault ("ns3::RedQueueDisc::AdaptMaxP", BooleanValue(true));
Config::SetDefault ("ns3::RedQueueDisc::LInterm", DoubleValue (10));

LInterm is the marking/dropping probability that is set. Internally again it will be in between 0-1.

Now run the code again as

~/ns-allinone-3.28.1/ns-3.28.1$ ./waf --run "myred-tests --testNumber=3"

What do you see?

You should see lower drop rates as compared to the last time you ran the same code. **Why?** (Better **RED** parameters!)

PRACTICE QUESTIONS:

1. Change one flow to TCP and other to UDP in mytopology.cc from last lab and observe the throughput this time with **RED/ARED** at the bottleneck link.
2. Change the timing of the start of one of the flows in the above question to see the effect of Queue at the bottleneck node again with **RED/ARED.**

Further Reading:

There are several other variants of **RED** algorithms have been proposed in the literature for which you can follow the below link for example codes and different parameter values for algorithms.

https://www.nsnam.org/docs/models/html/traffic-control.html

*****