Tickling Bufferbloat Problem in 5G to efficiently support TCP and Low Latency traffic by making suitable modifications RLC Layer in 5G NR
NS-3 module

<u>Team Details :</u>
Neel Yogendra Kansagra (CS22MTECH11002)
Nikhil Kori (CS22MTECH11014)
K Saravanan (CS22MTECH12007)

<u>TA:</u>
Harinder Kaur (CS21RESCH11008)

<u>Note: Updated Sections include Graphs (Results) & Conclusion</u>

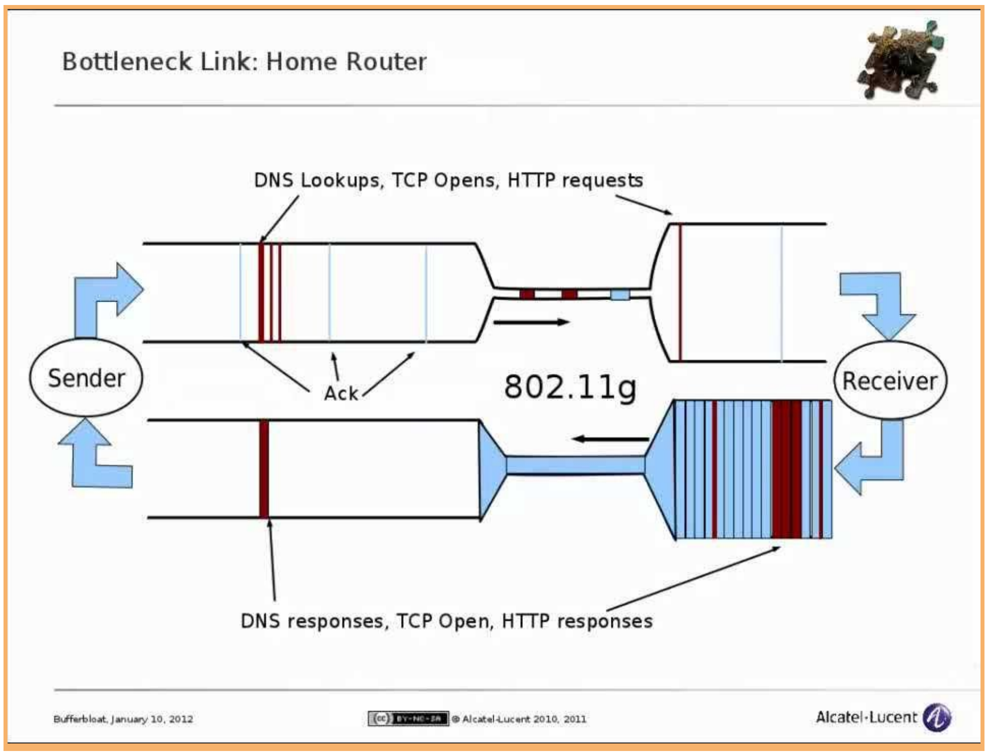
<u>Problem Statement</u>

To simulate (using NS-3) the Bufferbloat problem in 5G and simulate proposed solutions in 5G NR and verify its effectiveness.

<u>Project Description</u>

The Bufferbloat Problem discussed here is regarding the queuing latency in radio networks. Anywhere a network queue is present, the possibility of bufferbloat exists. In the 3GPP model for 4G/5G, the layer responsible for buffering packets before they are sent out over wireless medium is the RLC Layer.

In the below example we can see small packets like DNS responses are intermixed with large size packets in Buffer, hence causing sudden drop in internet performance, when in reality there is no congestion.



Bottleneck Link: Home Router
DNS Lookups, TCP Opens, HTTP requests
Sender
Ack
802.11g
Receiver
DNS responses, TCP Open, HTTP responses
Bufferbloat, January 10, 2012
© Alcatel-Lucent 2010, 2011
Alcatel·Lucent

Fig. 1 [1]

Now we used the paper - "Smart Backlog Management to Fight Bufferbloat in 3GPP Protocol Stacks" and made the following observations.

Even though multiple Radio Bearers can be mapped to different types of flows, In reality operators use a single bearer for all types of typical internet flow, to reduce network management overheads. This means at the RLC Buffer a single queue contains low latency as well as elephant flow packets, thus causing random bufferbloat.

This bufferbloat happens at the RLC Buffer, to counter this we use the innovative approach used in the paper - Introduce a Traffic Control mechanism on top of the 3GPP Stack similar to the one used by Linux Kernel.

We implemented the BQL Algorithm with other AQM Algorithms like FQCodel. We do not go into details about FQCodel, but we briefly discuss BQL.

Byte Queue Limits

This is how the algorithm works in Linux (We assume any normal TCP flow in ethernet).

1. The outgoing packet is enqueued in a queuing discipline (qdisc) which is basically a virtual connection to the outgoing connection.
2. The packet is then removed from the qdisc buffer and given to the network driver which buffers them in the transmission ring before sending it to outgoing hardware ports.
3. The BQL algorithm dynamically allocates the maximum transmission ring buffer size such that latency is minimised but starvation is also avoided.

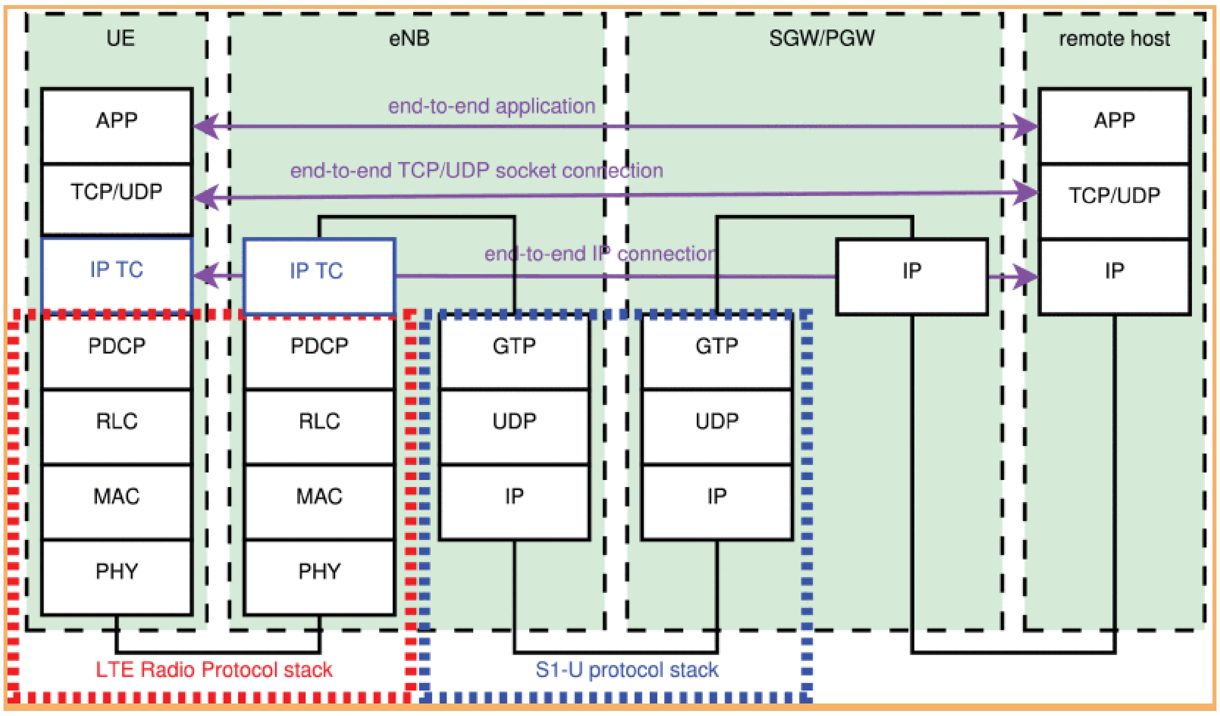


Fig. 2 [2]

Later in our final implementation we also referred to the paper - "**Dynamic Buffer Sizing and Pacing as Enablers of 5G Low-Latency Services.**"

Implementations in NS-3

Traffic Control Layer:

In NS-3 the Traffic Control Layer was introduced to imitate the Linux Traffic Control Infrastructure. This layer is in between the NetDevices (L2) and any network protocol (eg. IP). This layer Intercepts both the outgoing packet as well the incoming packet from and to the network layer and network device. In particular, outgoing packets are enqueued in a queuing discipline, which can perform multiple actions on them. It is in charge of processing packets and performing actions on them: scheduling, dropping, marking, policing, etc.

Implementation at gNB:

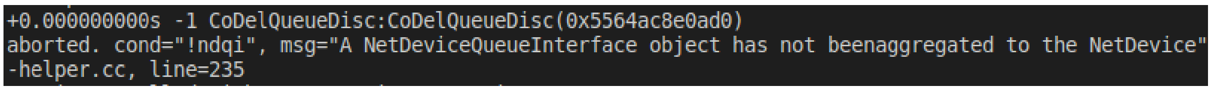
Traffic Control Helper is a class which can be used for setting up a traffic control layer above a NetDevice. Using Traffic Control Helper we can set the queues, queuing algorithms (Like FQCodel, Red Codel), algorithm parameters.

But a gNB is implemented as a derived class of NetDevice parent class, so has a lot of missing functions that are present in NetDevice class (used for point to point links) but are not present in gNB NetDevice. For this reason the Traffic Control Layer is unable to call a lot of essential functions.

For example, let's take the scenario we tried to implement.

Challenge #1:

When we tried to install the root queue on gNB NetDevice we got the following error.



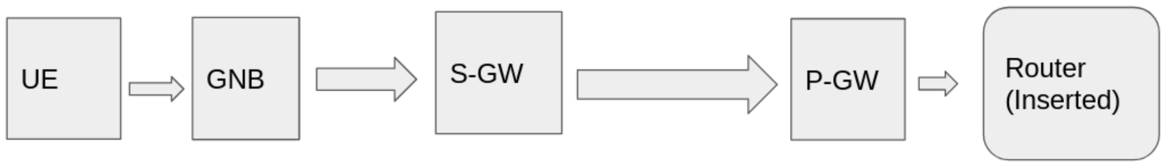
```
+0.000000000s -1 CoDelQueueDisc:CoDelQueueDisc(0x5564ac8e0ad0)
aborted. cond="!ndqi", msg="A NetDeviceQueueInterface object has not beenaggregated to the NetDevice"
-helper.cc, line=235
```

This is because the gNB NetDevice is a wireless link, unlike a point to point link which has a fixed interface. The lack of a fixed interface means, queue cannot be installed. Hence This approach did not work out. By fixed interface we mean that the queue knows where to listen for incoming packets. But in a wireless link, the queue is not aware which frequency range or RB, packets may arrive on, hence the error.

Challenge #2:

We tried to implement the BQL Algorithm from scratch on gNB (By studying functions and classes in traffic-control.cc). Again the issue we faced was we were unable to identify a fixed interface for incoming packets from where we could queue them.

Preliminary Solution



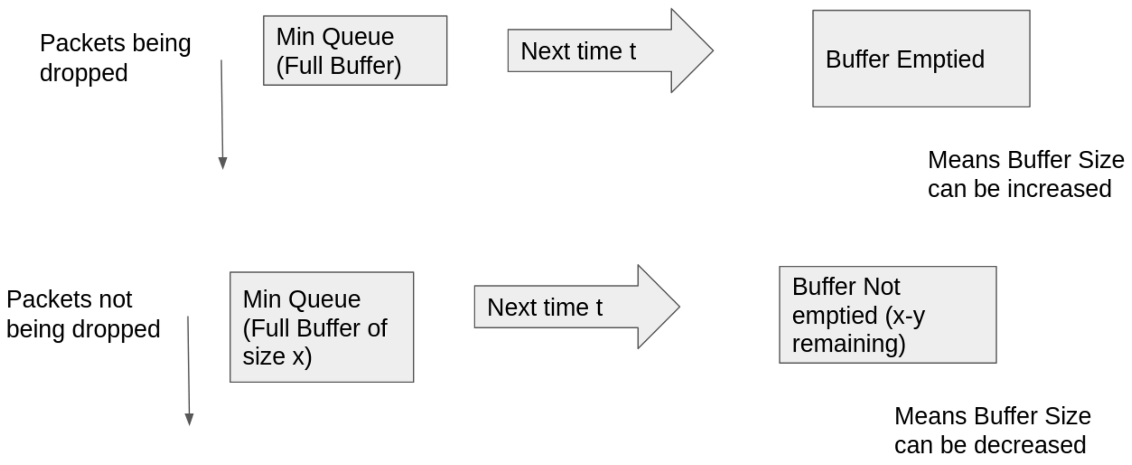
Since our aim was to reduce bufferbloat we introduced a router beyond P-GW and installed a queue on it. We called TC helper's BQL Algorithm with FQCodel on this queue.

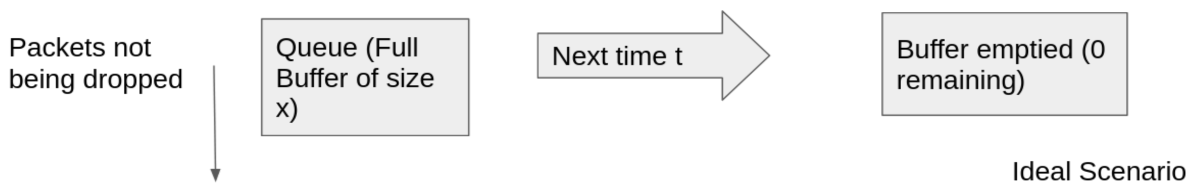
The code snippet is give below: (File: solution.cc)

```
  TrafficControlHelper tchBottleneck;
  tchBottleneck.SetQueueLimits ("ns3::DynamicQueueLimits");
  tchBottleneck.SetRootQueueDisc ("ns3::CoDelQueueDisc");

  Config::SetDefault ("ns3::CoDelQueueDisc::MaxSize",QueueSizeValue (QueueSize
(QueueSizeUnit::PACKETS, 1000)));

  Ptr<Node> pgw = epcHelper->GetPgwNode ();
  NodeContainer remoteHostContainer;
  remoteHostContainer.Create (1);
  Ptr<Node> remoteHost = remoteHostContainer.Get (0);
  InternetStackHelper internet;
  internet.Install (remoteHostContainer);
  //internet.Install(router);
```

Final Implementation

DRQL: Dynamic RLC Queue Limit pseudocode


Packets being dropped
Min Queue (Full Buffer)
Next time t
Buffer Emptied
Means Buffer Size can be increased
Packets not being dropped
Min Queue (Full Buffer of size x)
Next time t
Buffer Not emptied (x-y remaining)
Means Buffer Size can be decreased


Packets not being dropped
Queue (Full Buffer of size x)
Next time t
Buffer empty (0 remaining)
Ideal Scenario

```
1. $T \leftarrow$ TTI;
   $min\_val \leftarrow$ INF;
   $dequeued\_bytes \leftarrow 0$;

2. $limit \leftarrow$ MAX_VAL_LIMIT;
   $last\_time \leftarrow$ now;

3. procedure DEQUEUED($bytes$)

4. $dequeued\_bytes \leftarrow dequeued\_bytes + bytes$

5. update $limit\_reached$

6. update $remaining$

7. $buffer\_starved \leftarrow$ $no\ remaining$ AND $limit\_reached$

8. if $buffer\_starved$ then
         increase buffer's $limit$

9. else if $remaining$ then
         if $min\_val$ greater than $remaining$ then
               $min\_val \leftarrow$ remaining
         if $last\_time$ + T greater than $now$ then
               $last\_time \leftarrow now$
               $limit \leftarrow limit - min\_val$
               $min\_val \leftarrow$ INF
         else if $last\_time$ + T greater than $now$ then
               $last\_time \leftarrow now$
```

```
        $min\_val \leftarrow \text{INF}$
```

The above DRQL Algorithm consists of following variables:

Limit → queue limit in bytes
Dequeued_bytes → the bytes that were forwarded
Last_time → timestamp since last interval
T → interval during which the sojourn time of the packets is measured
Min_val → the minimum sojourn time during an interval

Explanation:

- T is set to the system Transmission Time Interval (TTI), min_val to the maximum value supported by the type of the system, dequeued_bytes to 0, limit to the maximum RLC queue capacity and last_time to the actual time.

- The DEQUEUED procedure first checks if the limit was reached during that interval of time and whether there are remaining packets at the queue it checks whether a queue starvation happened .

- If Yes, the buffer limit is increased as starvation happens and some transmission possibilities are wasted.

- If the buffer limit was reached, and no more data remains at the queue, the queue could have been provided with more data, and some bandwidth has been squandered. Therefore, the buffer limit is immediately increased.

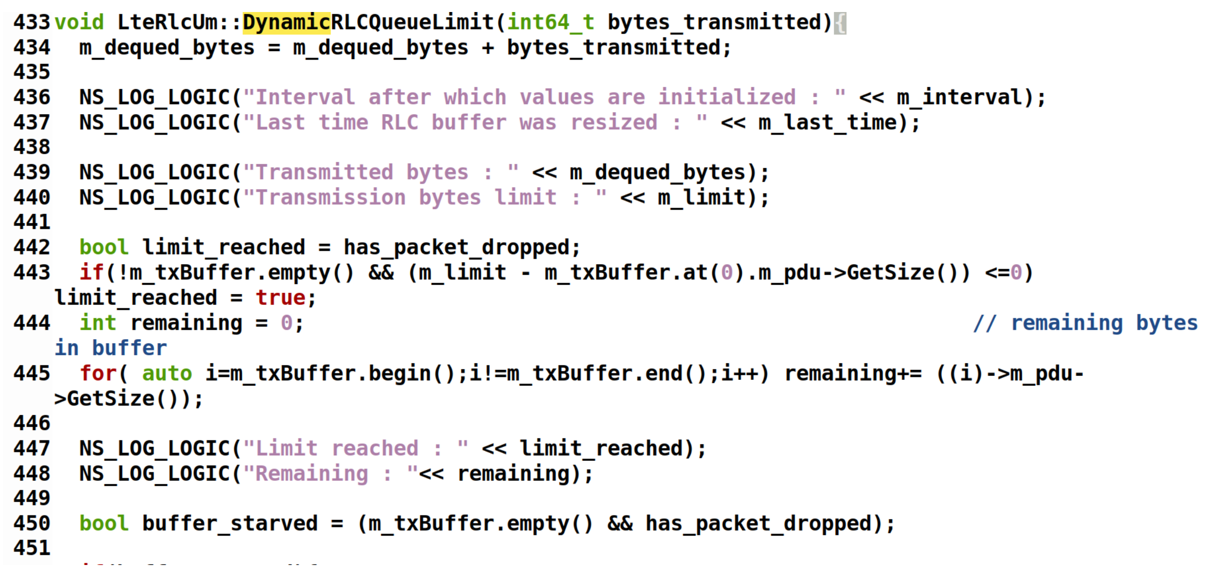
- If, on the contrary, some data still remains in the buffer, and it is smaller than the min_val, this value is assigned to min_val.

- If after an interval time, there has always been remaining data, the limit value is reduced by the lowest value observed during that interval, and kept at min_val.

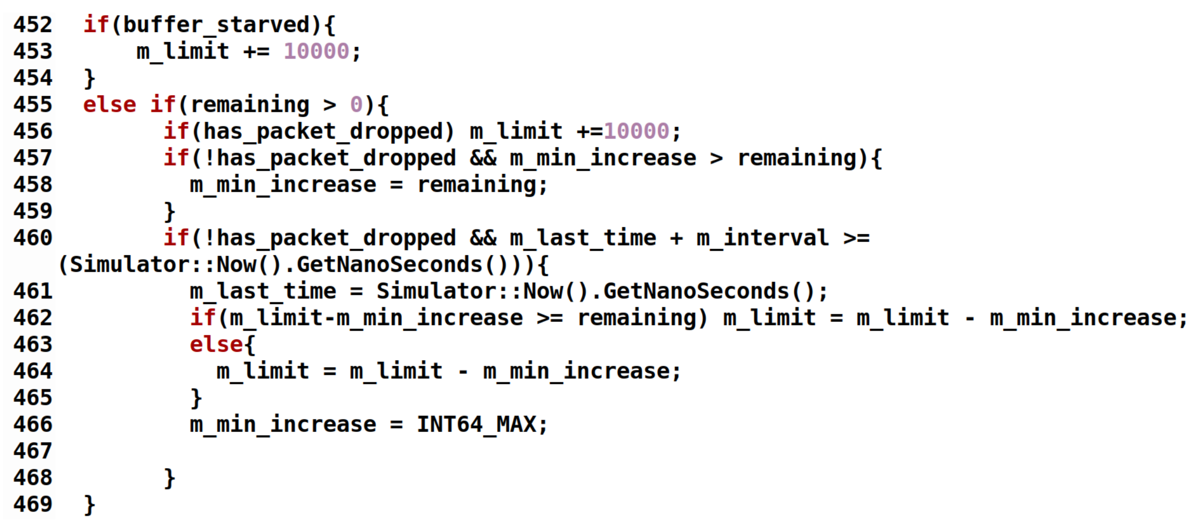
- If the limit is not reached and there is no remaining data the internal timer is simply reset and the

- min_val is set to INF as all the data that was forwarded into the buffer on that interval was successfully delivered to the PHY layer.

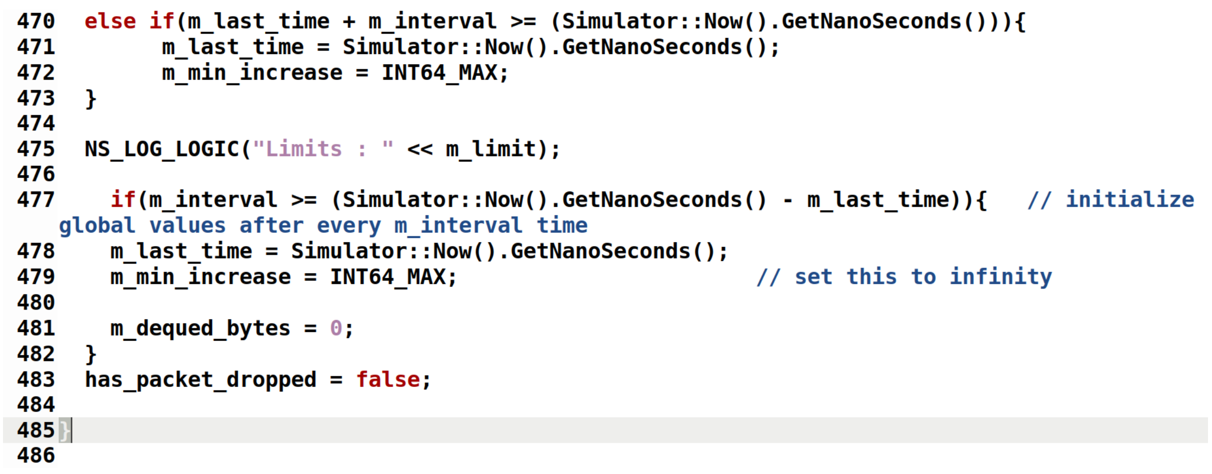
The NS-3-DEV core file “lte-rlc-um.cc” was modified. The function DynamicRLCQueueLimit which implements the above algorithms is show below (source file: dynamicRLCqueue.cc to be placed in scratch folder):



```
void LteRlcUm::DynamicRLCQueueLimit(int64_t bytes_transmitted){
  m_dequed_bytes = m_dequed_bytes + bytes_transmitted;

  NS_LOG_LOGIC("Interval after which values are initialized : " << m_interval);
  NS_LOG_LOGIC("Last time RLC buffer was resized : " << m_last_time);

  NS_LOG_LOGIC("Transmitted bytes : " << m_dequed_bytes);
  NS_LOG_LOGIC("Transmission bytes limit : " << m_limit);

  bool limit_reached = has_packet_dropped;
  if(!m_txBuffer.empty() && (m_limit - m_txBuffer.at(0).m_pdu->GetSize()) <=0) limit_reached = true;
  int remaining = 0;                                                    // remaining bytes in buffer
  for( auto i=m_txBuffer.begin();i!=m_txBuffer.end();i++) remaining+= ((i)->m_pdu->GetSize());

  NS_LOG_LOGIC("Limit reached : " << limit_reached);
  NS_LOG_LOGIC("Remaining : "<< remaining);

  bool buffer_starved = (m_txBuffer.empty() && has_packet_dropped);

```



```
  if(buffer_starved){
      m_limit += 10000;
  }
  else if(remaining > 0){
        if(has_packet_dropped) m_limit +=10000;
        if(!has_packet_dropped && m_min_increase > remaining){
          m_min_increase = remaining;
        }
        if(!has_packet_dropped && m_last_time + m_interval >= (Simulator::Now().GetNanoSeconds())){
          m_last_time = Simulator::Now().GetNanoSeconds();
          if(m_limit-m_min_increase >= remaining) m_limit = m_limit - m_min_increase;
          else{
            m_limit = m_limit - m_min_increase;
          }
          m_min_increase = INT64_MAX;

        }
  }
```



```
else if(m_last_time + m_interval >= (Simulator::Now().GetNanoSeconds())){
      m_last_time = Simulator::Now().GetNanoSeconds();
      m_min_increase = INT64_MAX;
}

NS_LOG_LOGIC("Limits : " << m_limit);

  if(m_interval >= (Simulator::Now().GetNanoSeconds() - m_last_time)){   // initialize
global values after every m_interval time
  m_last_time = Simulator::Now().GetNanoSeconds();
  m_min_increase = INT64_MAX;                       // set this to infinity

  m_dequed_bytes = 0;
}
has_packet_dropped = false;

}

```

```
void LteRlcUm::DynamicRLCQueueLimit(int64_t bytes_transmitted){
  m_dequed_bytes = m_dequed_bytes + bytes_transmitted;

  NS_LOG_LOGIC("Interval after which values are initialized : " << m_interval);
  NS_LOG_LOGIC("Last time RLC buffer was resized : " << m_last_time);

  NS_LOG_LOGIC("Transmitted bytes : " << m_dequed_bytes);
  NS_LOG_LOGIC("Transmission bytes limit : " << m_limit);

  bool limit_reached = has_packet_dropped;
  if(!m_txBuffer.empty() && (m_limit - m_txBuffer.at(0).m_pdu->GetSize()) <=0) limit_reached = true;
  int remaining = 0;                              // remaining bytes in buffer
  for( auto i=m_txBuffer.begin();i!=m_txBuffer.end();i++) remaining+= ((i)->m_pdu->GetSize());

  NS_LOG_LOGIC("Limit reached : " << limit_reached);
  NS_LOG_LOGIC("Remaining : "<< remaining);

  bool buffer_starved = (m_txBuffer.empty() && has_packet_dropped);

  if(buffer_starved){
        m_limit += 10000;
  }
  else if(remaining > 0){
        if(has_packet_dropped) m_limit +=10000;
        if(!has_packet_dropped && m_min_increase > remaining){
        m_min_increase = remaining;
        }
        if(!has_packet_dropped && m_last_time + m_interval >= (Simulator::Now().GetNanoSeconds())){
        m_last_time = Simulator::Now().GetNanoSeconds();
        if(m_limit-m_min_increase >= remaining) m_limit = m_limit - m_min_increase;
        else{
        m_limit = m_limit - m_min_increase;
        }
        m_min_increase = INT64_MAX;

        }
  }
  else if(m_last_time + m_interval >= (Simulator::Now().GetNanoSeconds())){
        m_last_time = Simulator::Now().GetNanoSeconds();
        m_min_increase = INT64_MAX;
  }

  NS_LOG_LOGIC("Limits : " << m_limit);

        if(m_interval >= (Simulator::Now().GetNanoSeconds() - m_last_time)){   // initialize global values after every
m_interval time
        m_last_time = Simulator::Now().GetNanoSeconds();
```

```
            m_min_increase = INT64_MAX;                    // set this to infinity

            m_dequed_bytes = 0;
  }
  has_packet_dropped = false;

}
```

Simulation

Test scenario created/used to conduct the experiments (Preliminary)

1. We set up one gNB.
2. We took one UEs, one with multiple Apps, one being mice flow receiving low latency packets from gNB (Downlink scenario) .
3. After some time we will activate the second App with elephant flow (bursty data) from gNB. This will cause a drop in throughput of the first UE simulating Bufferbloat.
4. We used TCP Flow.
5. Now we implemented our proposed algorithm and noted down changes in throughput and delay.

Simulation parameters in tabular form

Simulation Parameter	Value
Number of UEs	1; 1 Downlink UDP Flow per UE from the Remote Host.
Number of gNBs	1
Locations of UE (in meters)	(30,0)
Base Station position	(0,0)
gNB Tx Power	23 dBm
S1-U Link Delay between gNodeB and P-GW	1 ms

P2P link between P-GW and Remote Host	Data Rate: 10 Gbps Link Delay: 5 ms
Channel bandwidth	20 MHz
Central frequency	28 GHz
Scenario	UMi_StreetCanyon
Shadowing	disabled
Numerologies	0
Application Type	TCP onoff application and TCP bulk Sender.
Antennas for all the UEs	NumRows: 2 NumColumns: 4 AntennaElement: IsotropicAntennaModel
Antennas for all the gNbs	NumRows: 4 NumColumns: 8 AntennaElement: IsotropicAntennaModel

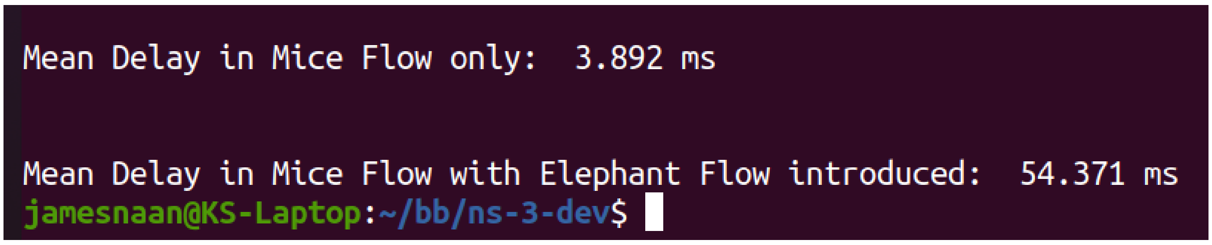
Performance metrics

Flow's	Delay in Mice Flow
Mice flow	3.892 ms
Mice flow with Elephant flow	54.371 ms

Results

Test Setup (Pre-liminary)

We successfully recreated a scenario to observe Bufferbloat in NS-3.
As we can see from the above table, the low latency packet gets significantly delayed on the introduction of the Bursty packet flow, as the RLC queue experiences bufferbloat.

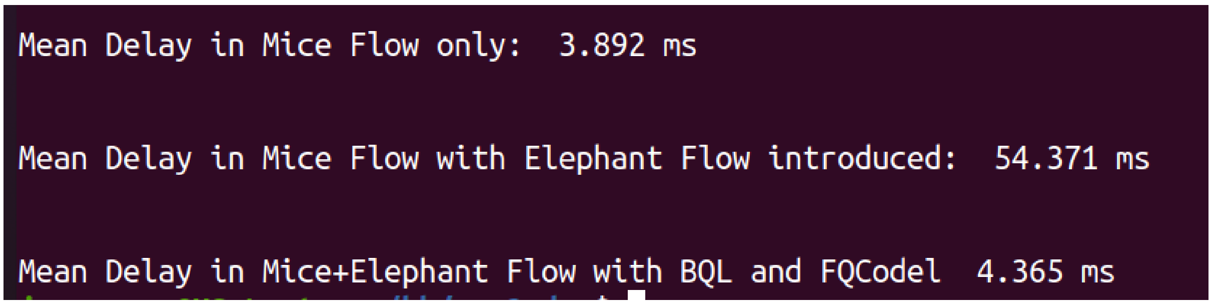

```
Mean Delay in Mice Flow only:  3.892 ms

Mean Delay in Mice Flow with Elephant Flow introduced:  54.371 ms
jamesnaan@KS-Laptop:~/bb/ns-3-dev$
```

To counter this we tried various ways and the different challenges have been described above. We finally discuss our successful implementations.

Preliminary Implementation

To counter this bufferbloat, we introduced BQL Algorithm with FQCodel at a router before the remote host side of gNB.

Overall the best way to counter Bufferbloat was BQL + FQCodel. This is the same result found in the paper and we verified it once again.

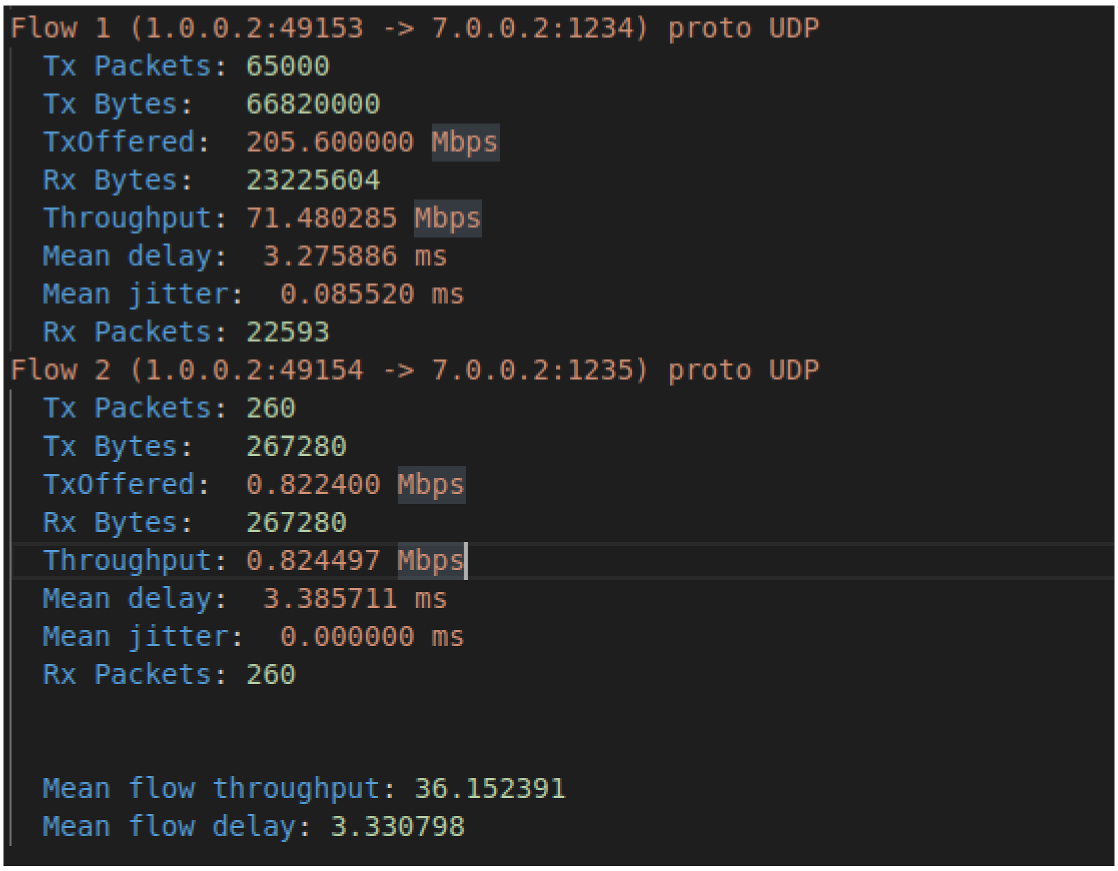

```
Mean Delay in Mice Flow only:  3.892 ms

Mean Delay in Mice Flow with Elephant Flow introduced:  54.371 ms

Mean Delay in Mice+Elephant Flow with BQL and FQCodel  4.365 ms
```

Final Implementation (DRQL Algorithm)
For testing DRQL algorithm we are using a test setup as shown below:

Simulation Parameter	Value
Number of UEs	1; 1 Downlink UDP Flow per UE from the Remote Host.
Number of gNBs	1
Locations of UE (in meters)	(30,0)
Base Station position	(0,0)
gNB Tx Power	14 dBm
S1-U Link Delay between gNodeB and P-GW	1 ms
P2P link between P-GW and	Data Rate: 100 Gbps

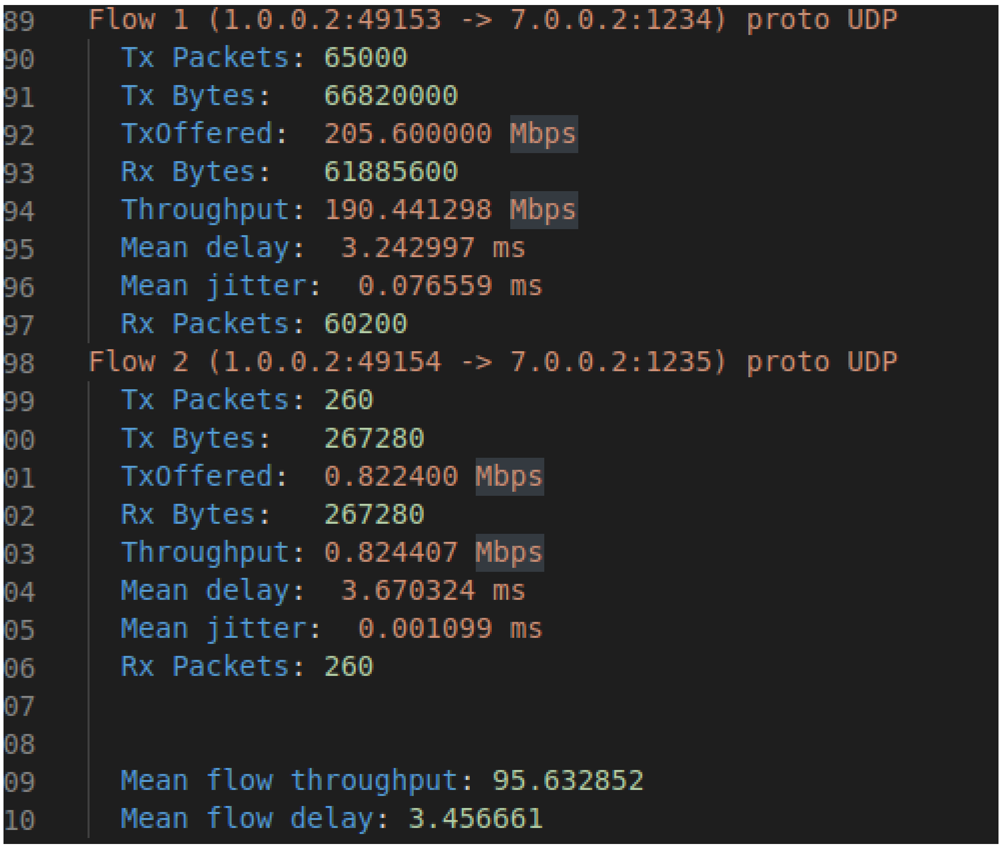
Remote Host	Link Delay: 5 ms
Channel bandwidth	100 MHz
Central frequency	7 GHz
Scenario	UMi_StreetCanyon
Shadowing	disabled
Numerologies	0
Application Type	2 UDP applications: 1 with lambda 25000 packets/sec Other with lambda 100 packets/sec
Size of RLC buffer	10KB

<u>Flow stats without DRQL algorithm</u>



```
Flow 1 (1.0.0.2:49153 -> 7.0.0.2:1234) proto UDP
  Tx Packets: 65000
  Tx Bytes:   66820000
  TxOffered:  205.600000 Mbps
  Rx Bytes:   23225604
  Throughput: 71.480285 Mbps
  Mean delay:  3.275886 ms
  Mean jitter:  0.085520 ms
  Rx Packets: 22593
Flow 2 (1.0.0.2:49154 -> 7.0.0.2:1235) proto UDP
  Tx Packets: 260
  Tx Bytes:   267280
  TxOffered:  0.822400 Mbps
  Rx Bytes:   267280
  Throughput: 0.824497 Mbps
  Mean delay:  3.385711 ms
  Mean jitter:  0.000000 ms
  Rx Packets: 260


  Mean flow throughput: 36.152391
  Mean flow delay: 3.330798
```

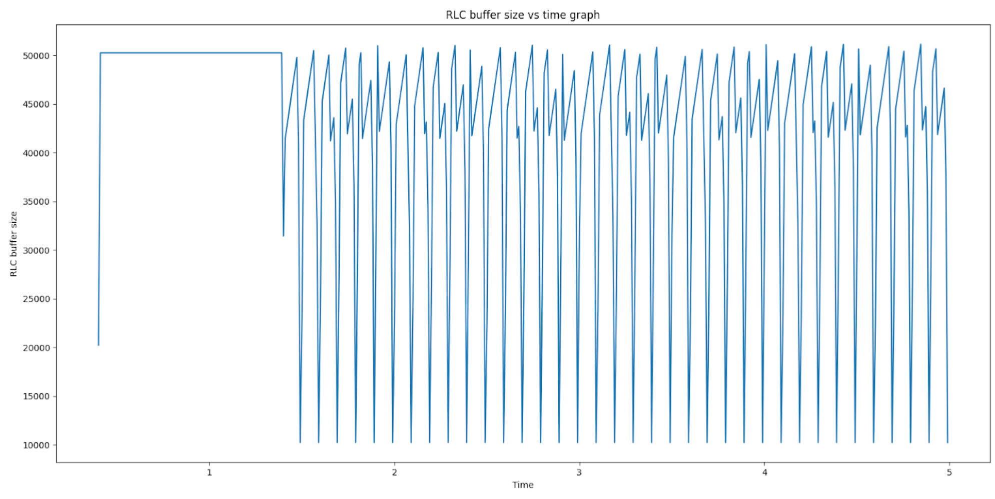
<u>Flow stats with DRQL algorithm</u>

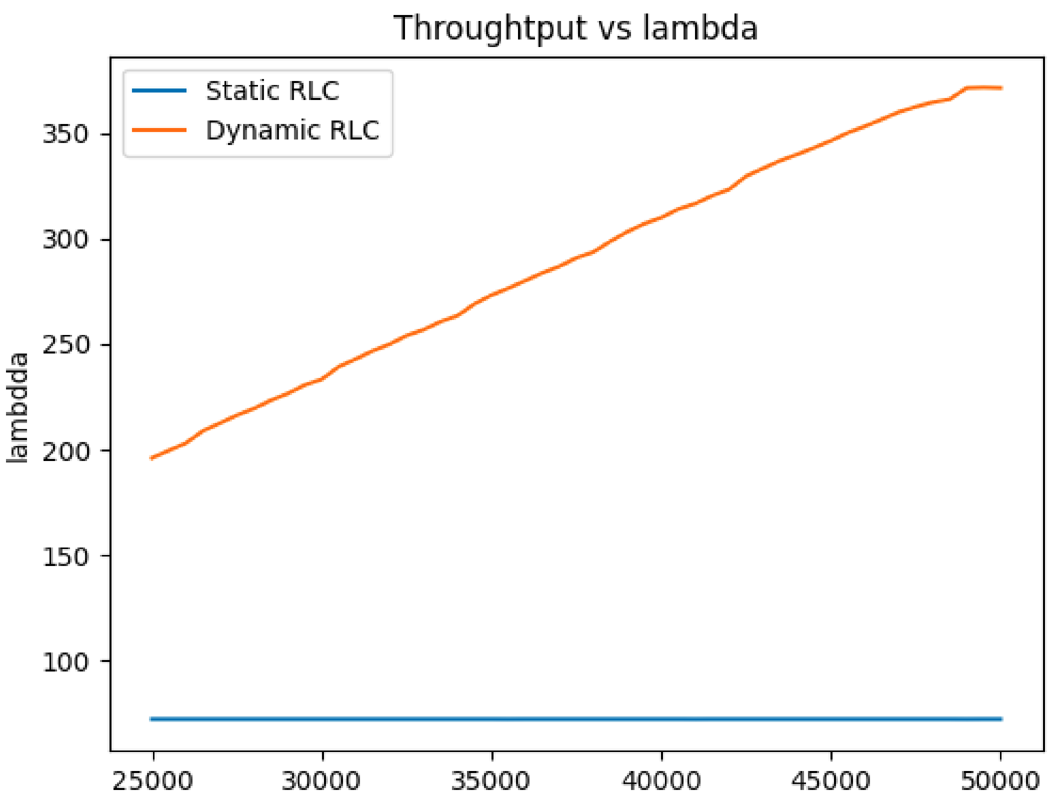


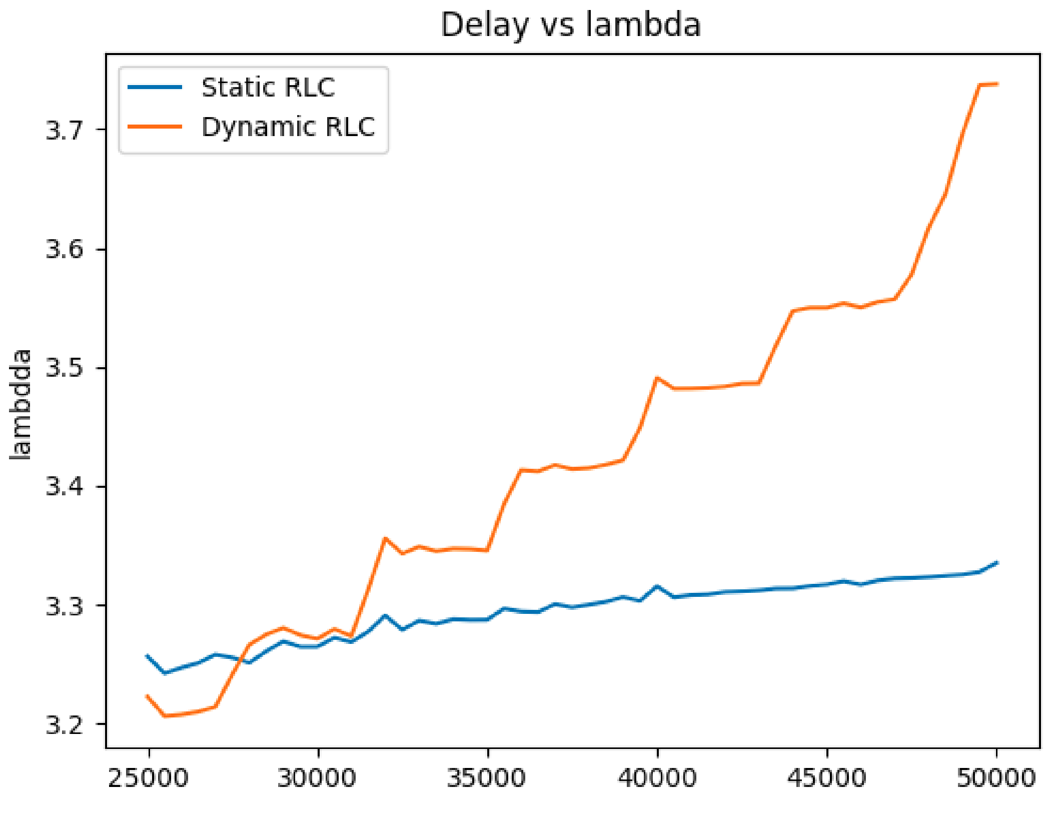
```
Flow 1 (1.0.0.2:49153 -> 7.0.0.2:1234) proto UDP
    Tx Packets: 65000
    Tx Bytes:   66820000
    TxOffered:  205.600000 Mbps
    Rx Bytes:   61885600
    Throughput: 190.441298 Mbps
    Mean delay:  3.242997 ms
    Mean jitter:  0.076559 ms
    Rx Packets: 60200
Flow 2 (1.0.0.2:49154 -> 7.0.0.2:1235) proto UDP
    Tx Packets: 260
    Tx Bytes:   267280
    TxOffered:  0.822400 Mbps
    Rx Bytes:   267280
    Throughput: 0.824407 Mbps
    Mean delay:  3.670324 ms
    Mean jitter:  0.001099 ms
    Rx Packets: 260


    Mean flow throughput: 95.632852
    Mean flow delay: 3.456661
```

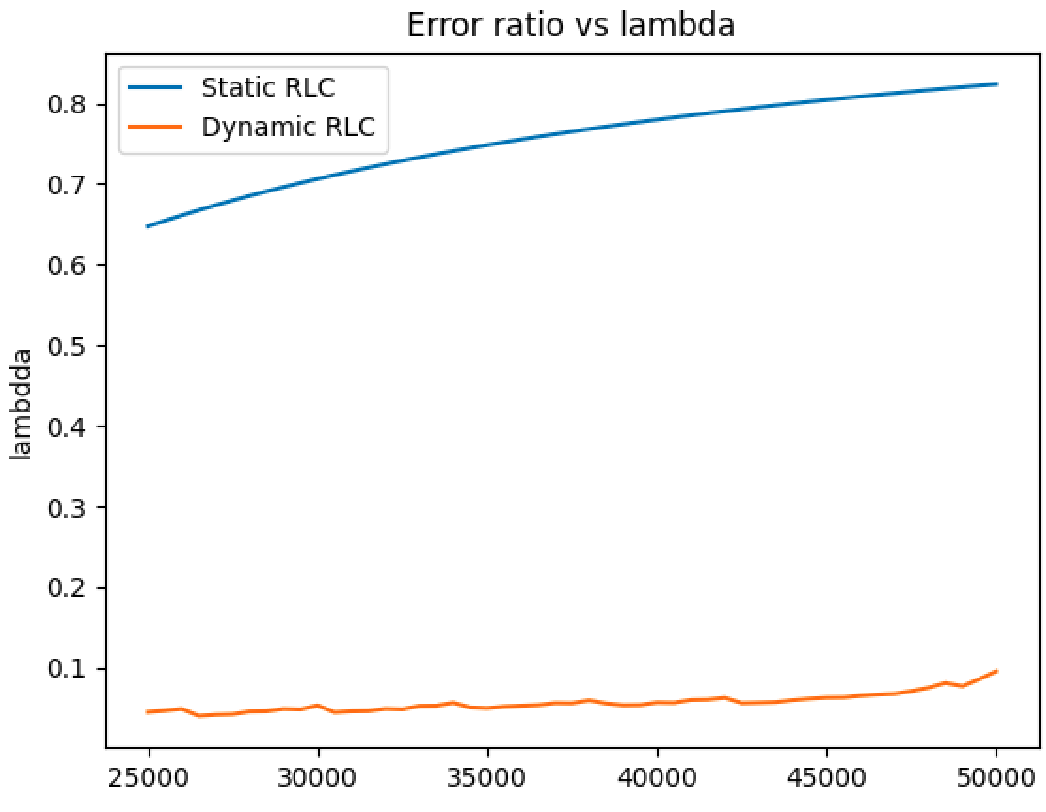
Flow 1 is elephant flow and flow 2 is mice flow . We can see that in 1st case the RLC buffer size is very less and fixed which is why most of the packets are lost even before they are sent to the MAC layer. But in the second case the RLC buffer size adjusts itself over time and increases or decreases the RLC buffer depending on how bursty traffic is. We can observe that elephant flow throughput increases significantly but delay of mice flow increases slightly, this happens because as the size of RLC buffer increases, delay in RLC takes more time. Here the algorithm prioritises safe delivery of data over average delay of packets.

GRAPHS


RLC buffer size vs time graph
RLC buffer size
Time
50000
45000
40000
35000
30000
25000
20000
15000
10000
1
2
3
4
5


Throughtput vs lambda
Static RLC
Dynamic RLC
lambdda
350
300
250
200
150
100
25000
30000
35000
40000
45000
50000


Delay vs lambda
Static RLC
Dynamic RLC
lambdda
3.2
3.3
3.4
3.5
3.6
3.7
25000
30000
35000
40000
45000
50000


Error ratio vs lambda
Static RLC
Dynamic RLC
lambdda
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
25000
30000
35000
40000
45000
50000

As we can see The packet loss ratio is significantly improved in our algorithm and is kind of a guaranteed flow generated. On closer observation it can be seen that because of this packet guaranteed there is a slight increase in delay (We want to decrease delay and hence decrease bufferbloat). Again this is because of the packet guarantee that has been achieved! We have concluded and further commented on this below

Conclusion & Future Work

So now if we were to pair this Algorithm with Codel (either at RLC BUffer itself or at P-GW, or at higher layers) Then we can achieve potentially better or same results than that achieved at Preliminary Solution (presented above).

To Conclude our DRQL Algorithm implementation does solve the packet loss problem of AQM ALgorithms and all it needs is to be paired with AQM Algorithm to solve bufferbloat too, which should be the next step for any further development on this project.

Our Implementation can be significantly improved to get performance levels on par with Wired Network Implementations which would require much more rigorous research.

References

1) Bufferbloat: Dark Buffers on the Internet. Networks without effective AQM may again be vulnerable to congestion collapse. https://dl.acm.org/doi/10.1145/206in3166.2071893
2) P. Imputato, N. Patriciello, S. Avallone and J. Mangues-Bafalluy, "Smart Backlog Management to Fight Bufferbloat in 3GPP Protocol Stacks," *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2019, pp. 1-6, doi: 10.1109/CCNC.2019.8651727. https://ieeexplore.ieee.org/document/8651727
3) **Dynamic Buffer Sizing and Pacing as Enablers of 5G Low-Latency Services** https://ieeexplore.ieee.org/document/9169837
4) NS-3 manual for 5G NR Module (version 2.2) https://cttc-lena.gitlab.io/nr/nrmodule.pdf
5) Traffic Control Layer https://www.nsnam.org/docs/release/3.37/models/html/traffic-control.html

Work Division

Neel:

1. Implement DRQL ALgorithm in NS-3
2. Lead Programmer for all NS-3 implementations

Nikhil Kori:

3. Implement Phase I with multiple devices installed in one UE.
4. Compiling Report, scripts, identifying challenges in implementing all the ideas presented in the paper

K Saravanan:

5. Implement Phase II Traffic Control using Traffic Controller
6. Research and analyse appropriate logs and traces to verify the working of traffic control in NS-3.

Combined Work:

1. Study the paper to understand the Traffic Control presented.
2. Study the paper and understand the DRQL Algorithm presented
3. Studying various Reference Materials

Anti-Plagiarism Statement

I certify that this assignment/report is our own work, based on our personal study and/or research on our personal/lab equipment and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, We understand our responsibility to report honour violations by other students if we become aware of it.

Names and Roll Nos:

Neel Yogendra Kansagra (CS22MTECH11002),

Nikhil Kori (CS22MTECH11014),

K Saravanan (CS22MTECH12007)

Date: 9/12/2022

Signatures: NYK, NK, KS