NS3 PROJECT REPORT

CSE 322

February 24, 2022

Md. Zarif Ul Alam (1705010)

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Contents

TASK A

Wireless high-rate (mobile)

Topology

We use a dumbbell topology for simulating Wireless high-rate (mobile) network in task A using ns3[2]

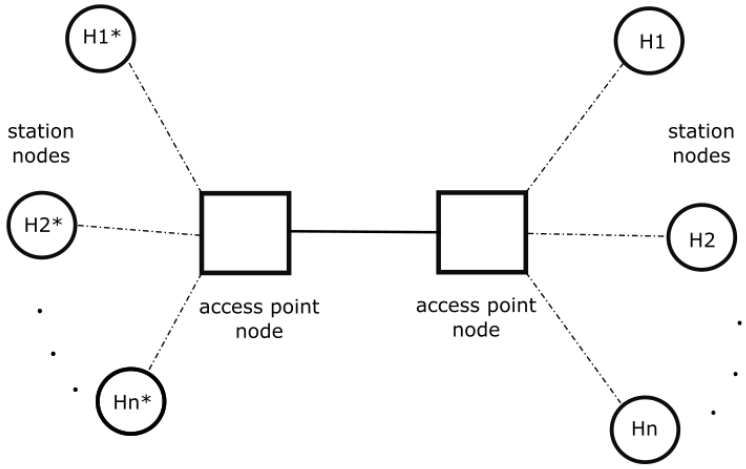

Figure 1: Wireless high-rate Simulation Topology

Network Specification

We have two routers connected through a point to point link (5Mbit/sec) with a link propagation delay of 50 msec. In the basic configuration, Host H1 sends packets to Host H1*, H2 to H2*, etc. and destinations send back acknowledgement packets as required by TCP. The links between the hosts and routers are also point to point links. Finally we assign the IP as follows,

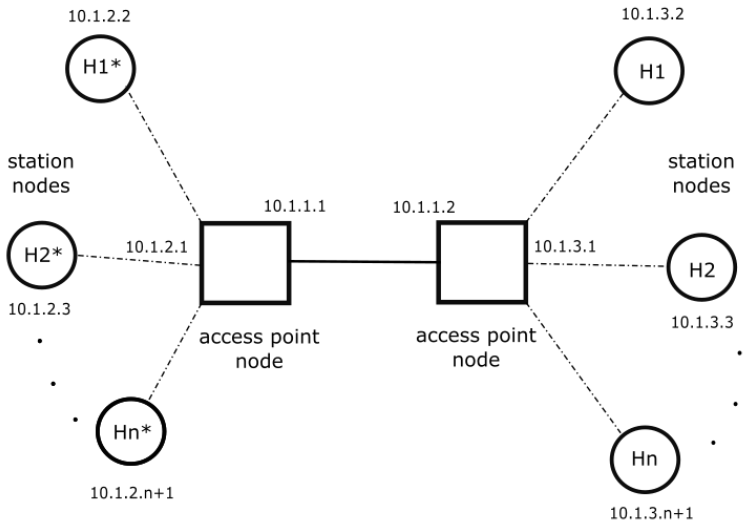

Figure 2: Wireless high-rate IP Assignment

Variation of Parameters (Results)

Vary nodes

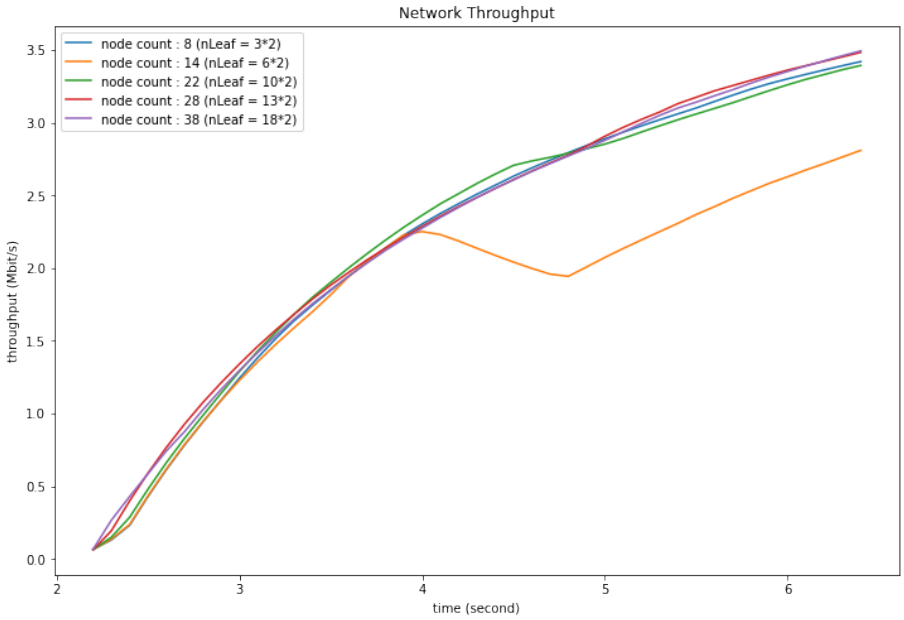

Figure 3: Network Throughput

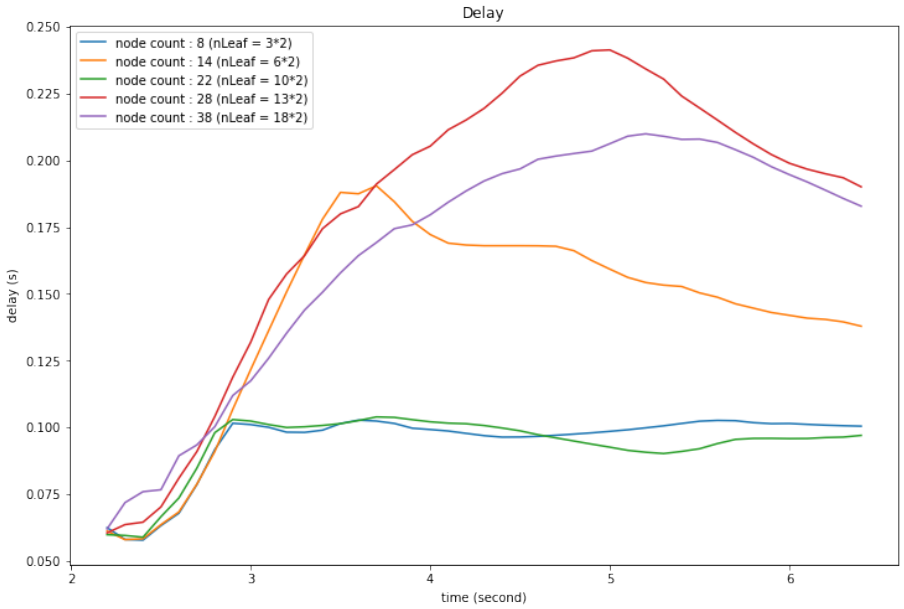

Figure 4: End to End Delay

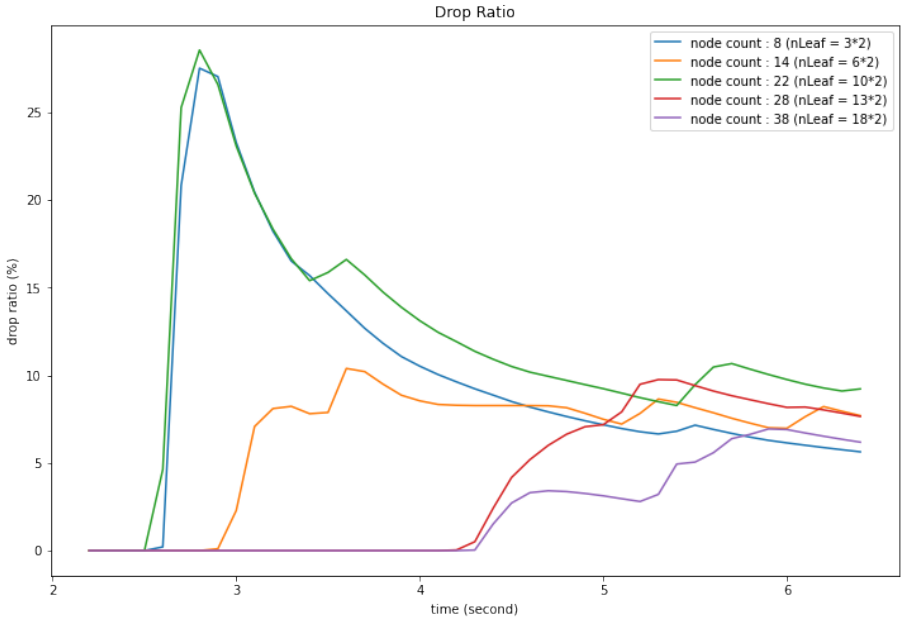

Figure 5: Packet Drop Ratio

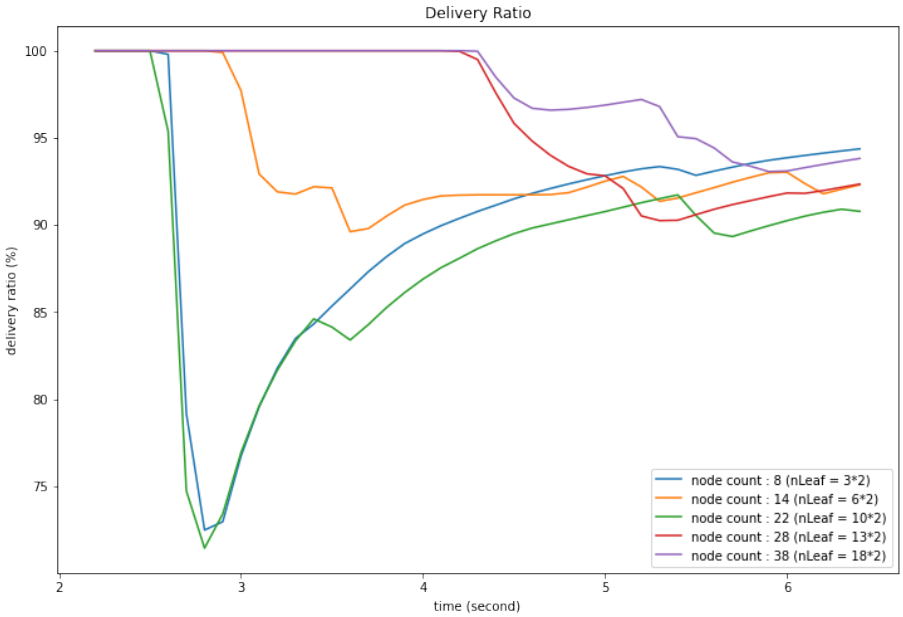

Figure 6: Packet Delivery Ratio

Vary flow

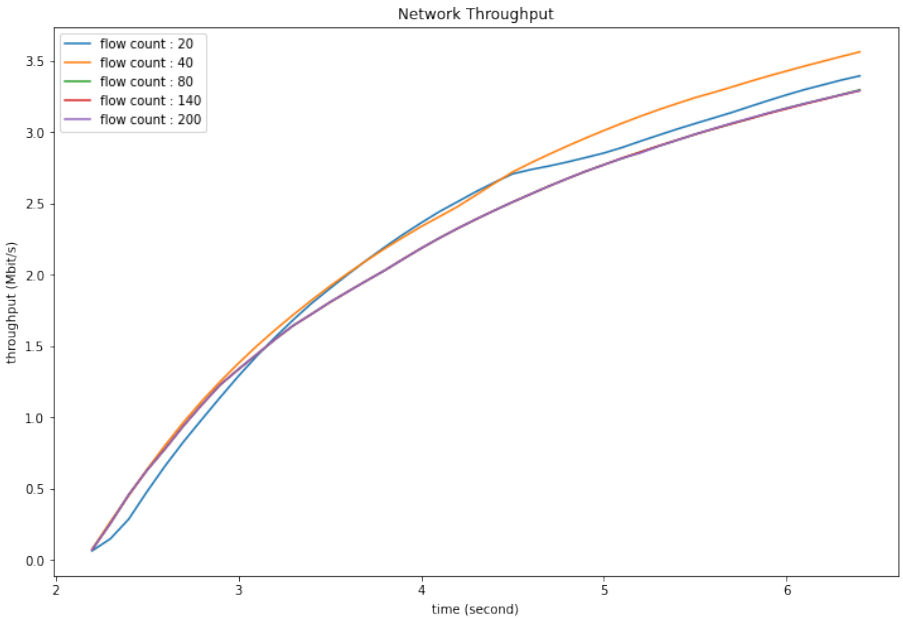

Figure 7: Network Throughput

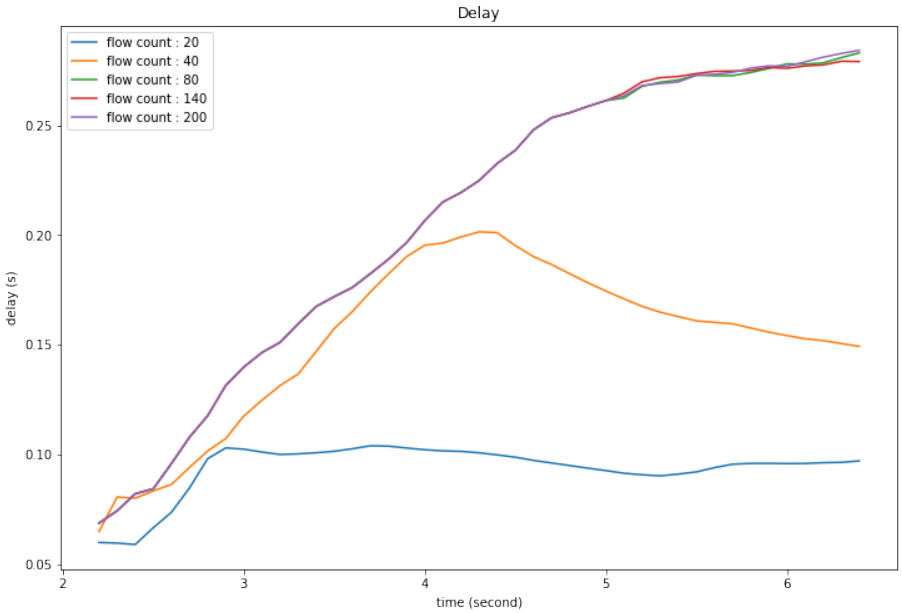

Figure 8: End to End Delay

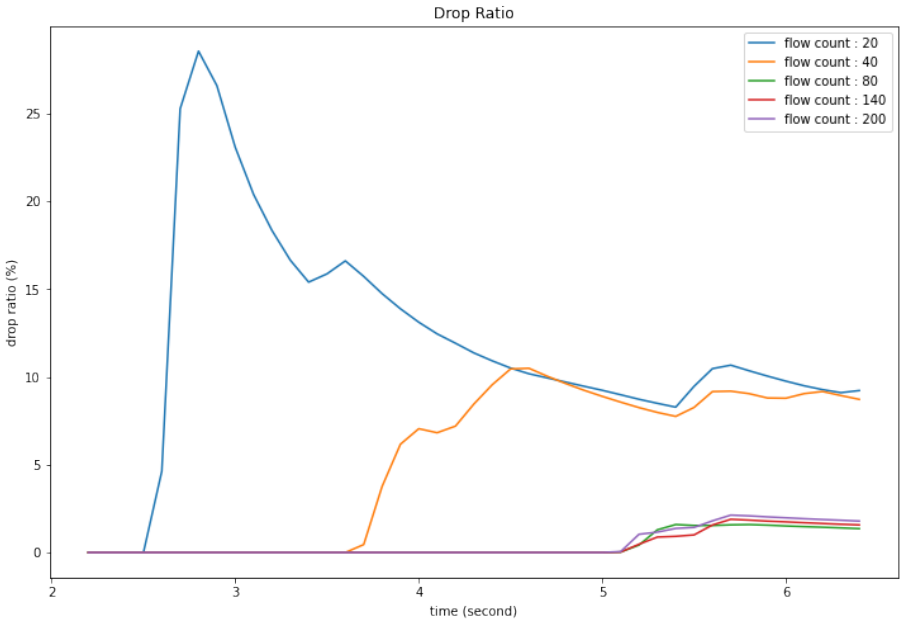

Figure 9: Packet Drop Ratio

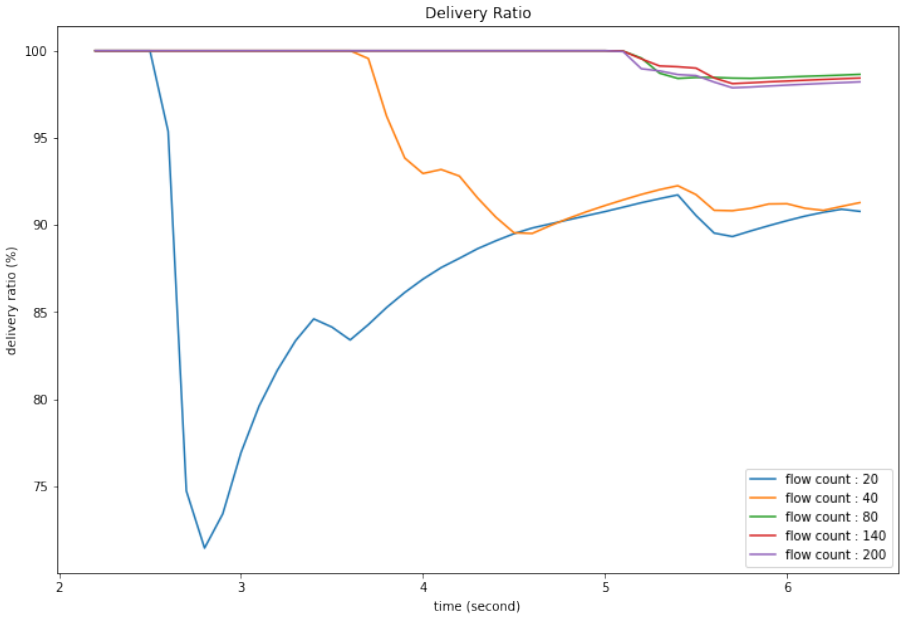

Figure 10: Packet Delivery Ratio

Vary packets per second

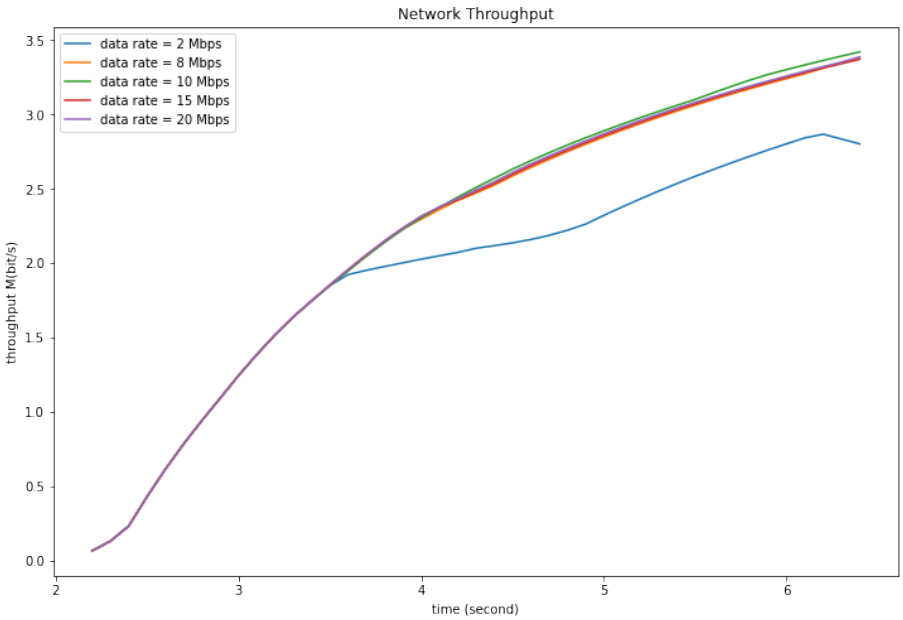

Figure 11: Network Throughput

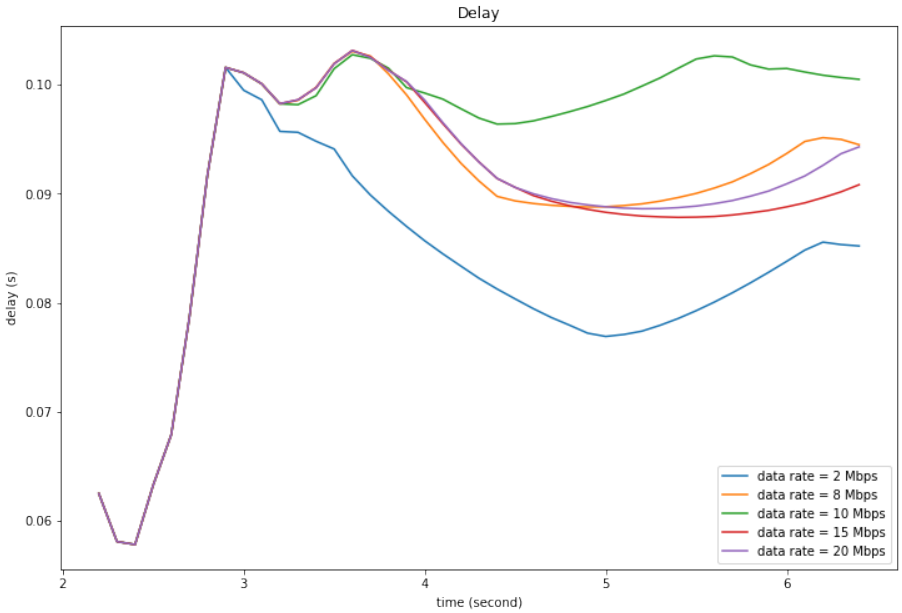

Figure 12: End to End Delay

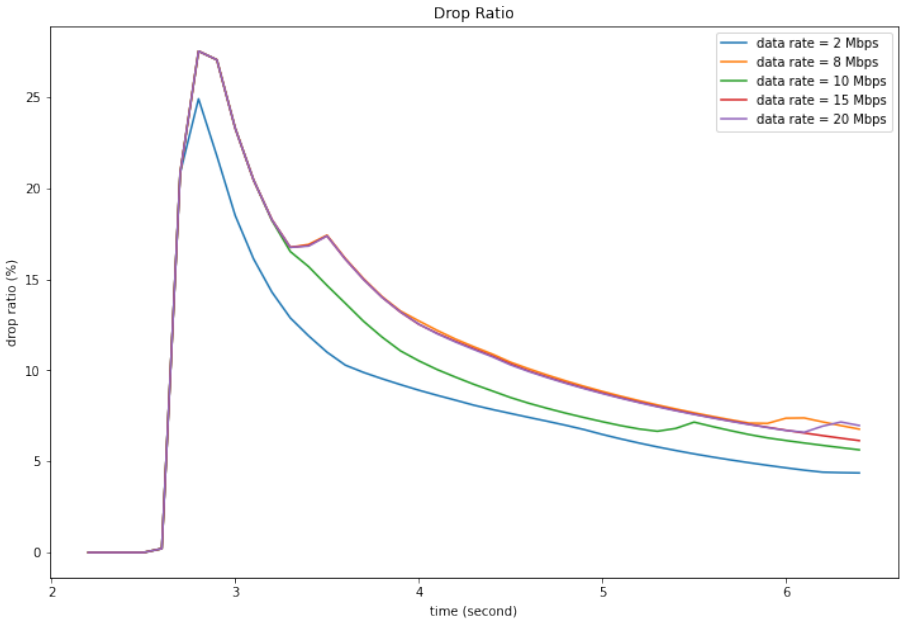

Figure 13: Packet Drop Ratio

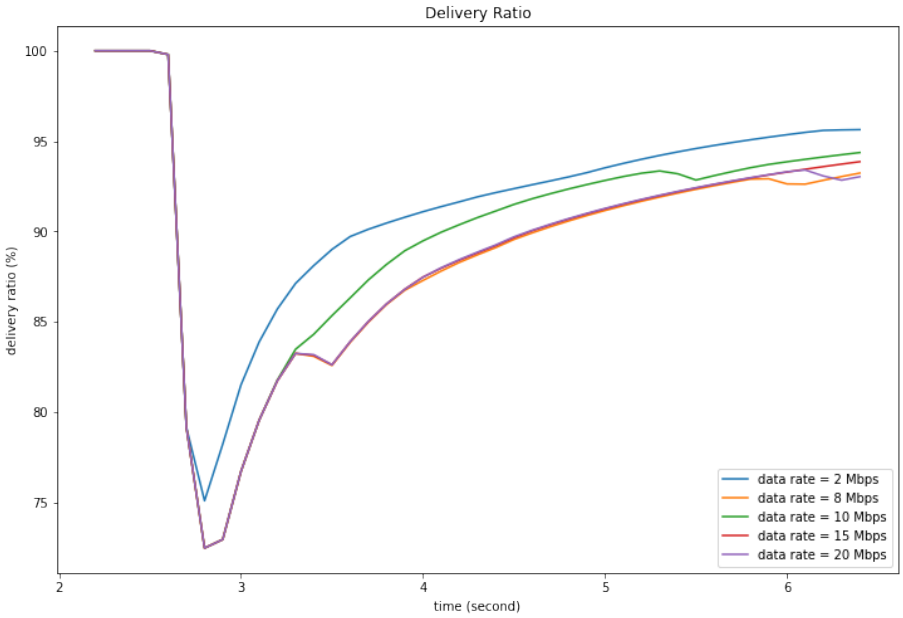

Figure 14: Packet Delivery Ratio

Vary speed

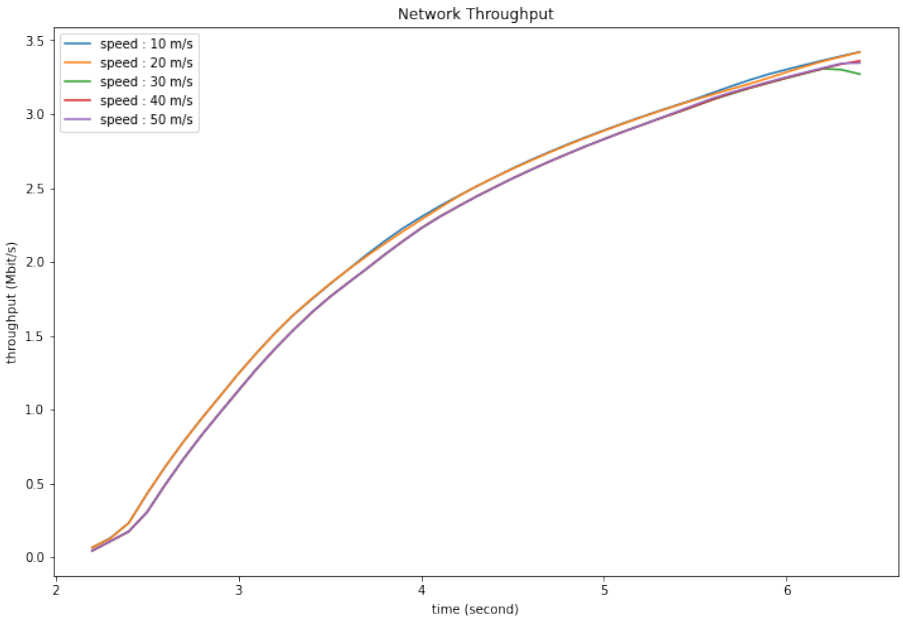

Figure 15: Network Throughput

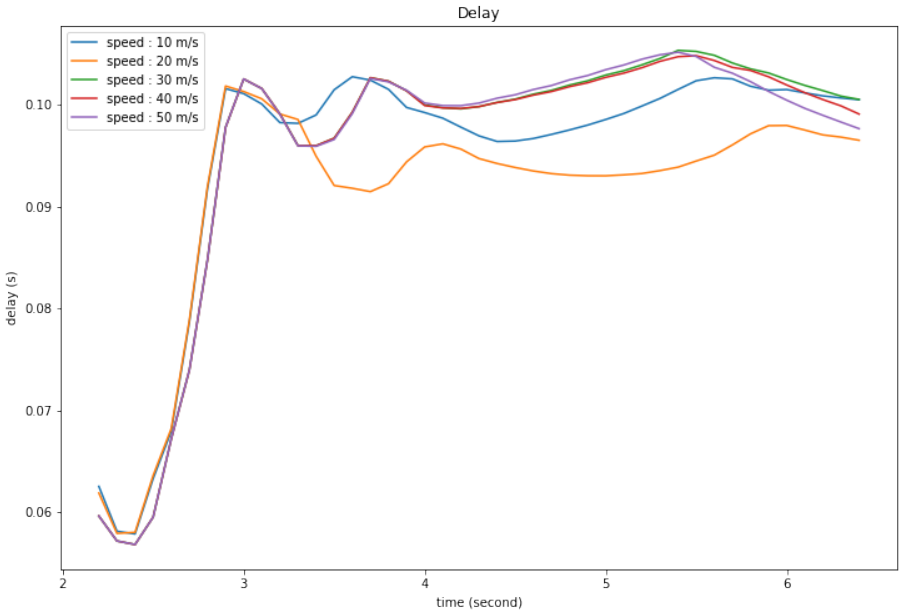

Figure 16: End to End Delay

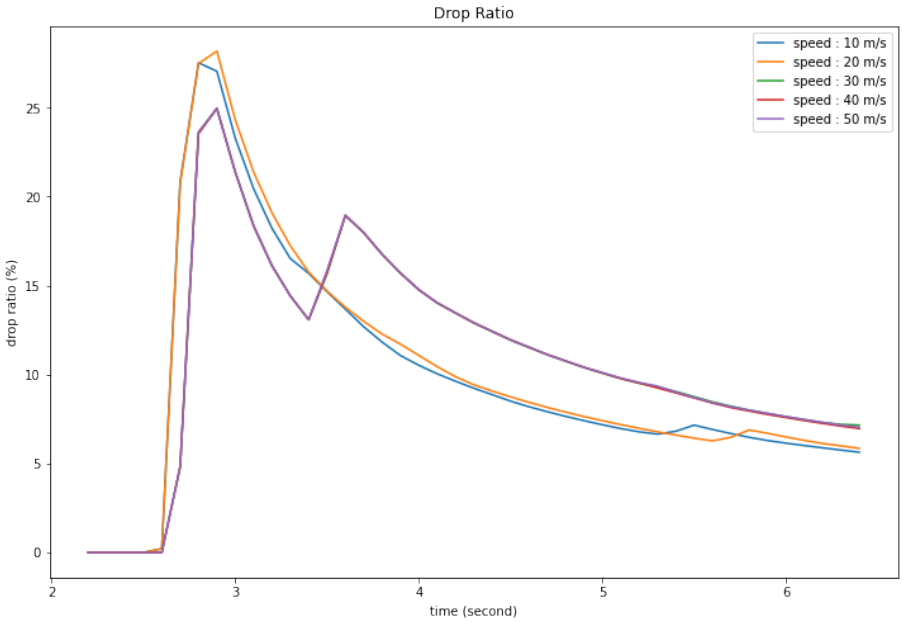

Figure 17: Packet Drop Ratio

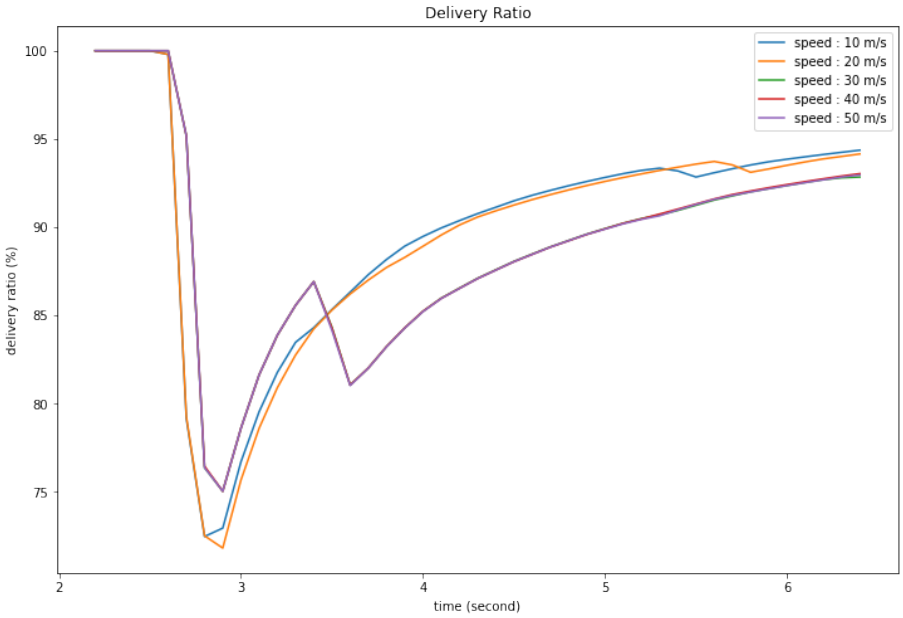

Figure 18: Packet Delivery Ratio

Variation of Parameters (Discussion)

No matter which parameter is varied, we see that the **throughput** doesn't vary much. This can be explained with the existence of our bottleneck link in our network. No matter how much we increase our leaf nodes, or increase flow, or packets per second or even speed, throughput is almost always same.

In case of **end to end delay**, the general trend is that it increases with the number of nodes, flow, packets per second and speed. This is intuitive since we can't transfer more than the bottleneck bandwidth and that is why we get higher end to end delays.

In case of **packet drop ratio**, it starts with 0% and as time increases the drop rate also increases rapidly at first. After the initial surge of packet drop, it slowly decreases because of congestion control mechanism. As the mentioned parameters are increased, packet drop almost always increase.

For **packet delivery ratio**, the trend is exactly as opposite as expected. Initially, the delivery rate is 100%. As the drop rate increases, the delivery rate decreases. But after a few seconds, when the network traffic becomes stable, the delivery rate quickly recovers and reaches above 90%. As the parameters are increased, packet delivery rate almost always decrease.

TASK A

Wireless low-rate (static)

Topology

We use a dumbbell topology for simulating Wireless low-rate (static) network in part 2 of task A.

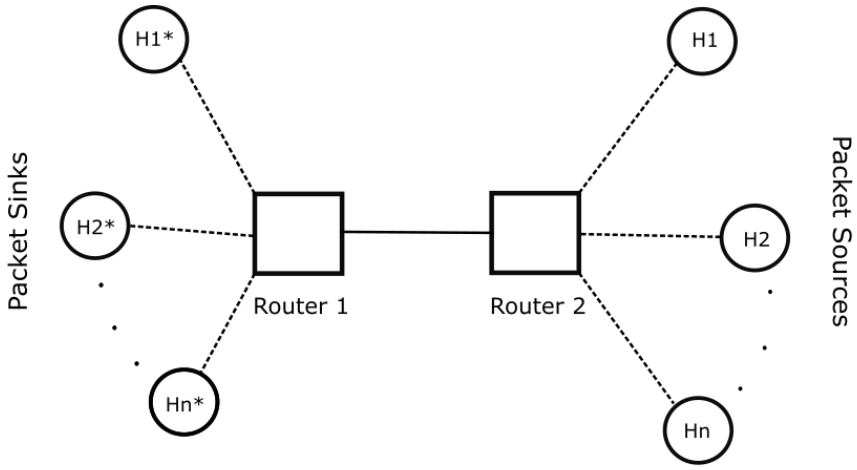

Figure 19: Wireless low-rate Simulation Topology

It is worth noting that for this simulation we have increased the bandwidth between router 1 and router 2, so this link is not bottleneck anymore. We do this to observe performance if there was not a bottleneck link as opposed to what we have done in wireless high rate.

Network Specification

We have two routers connected through a point to point link (50kbit/sec). In the basic configuration, Host H1 sends packets to Host H1*, H2 to H2*, etc. and destinations send back acknowledgement packets as required by TCP. The app data rate is set to 6000 bps and packet size used is 50 bit. The links between the hosts and routers are also point to point links.

Variation of Parameters (Result)

Vary nodes

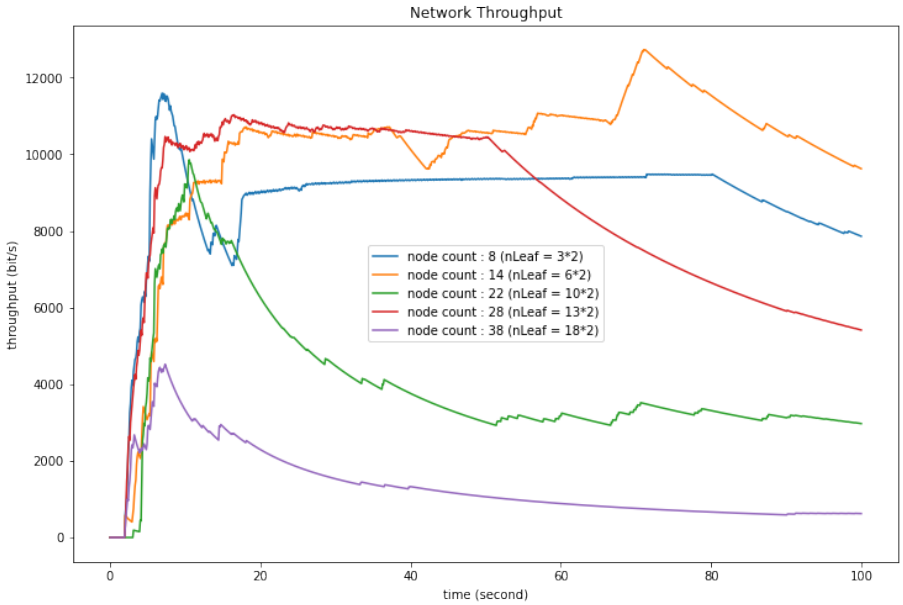

Figure 20: Network Throughput

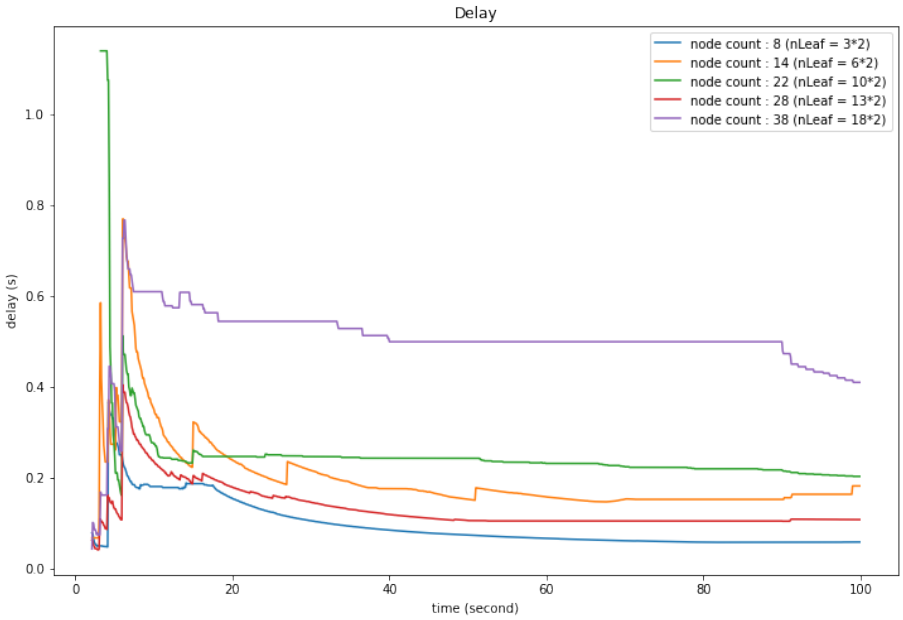

Figure 21: End to End Delay

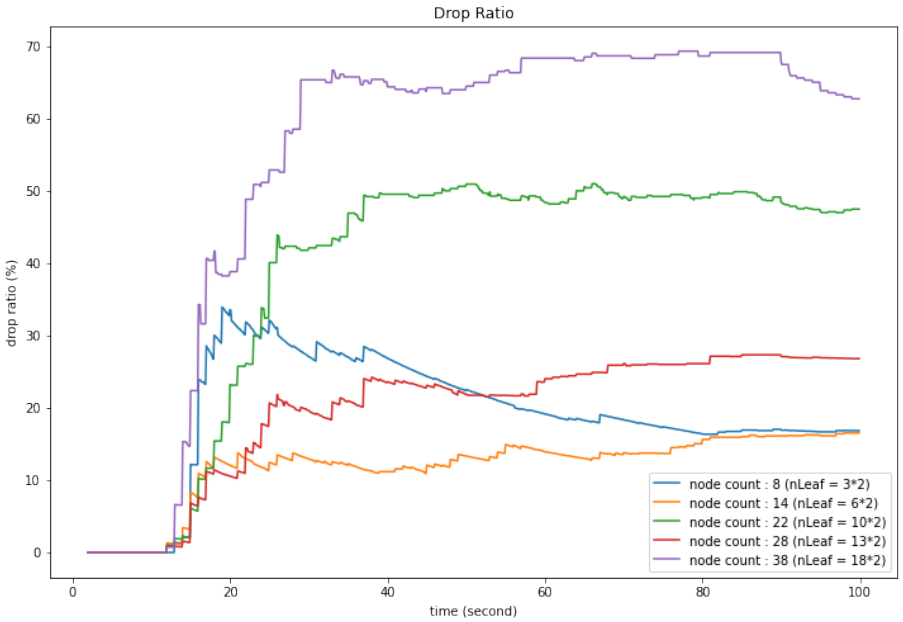

Figure 22: Packet Drop Ratio

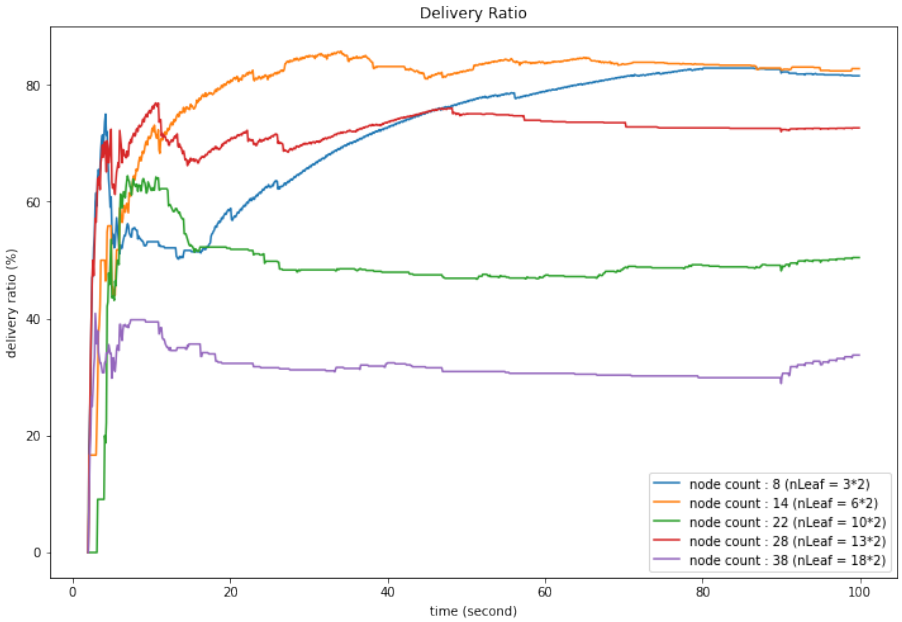

Figure 23: Packet Delivery Ratio

Vary flow

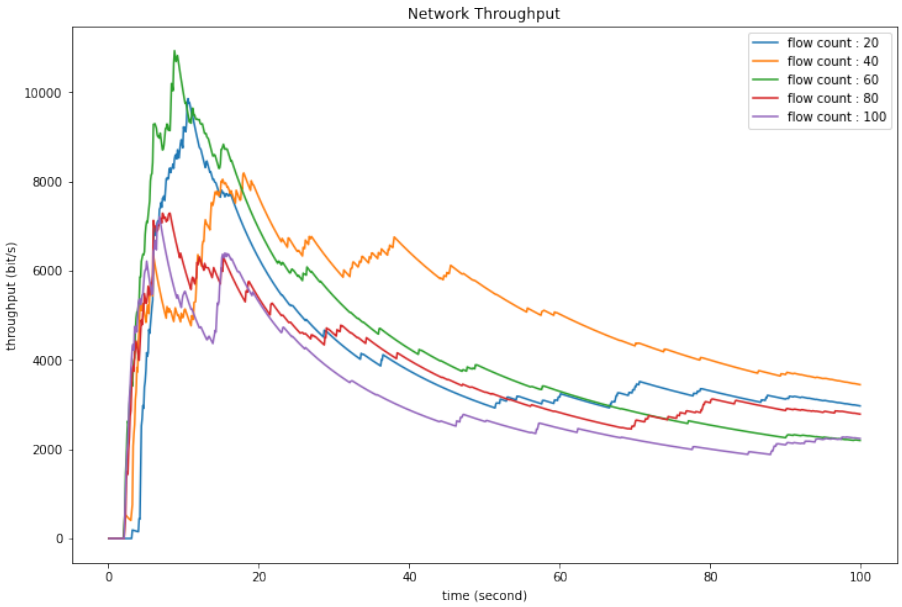

Figure 24: Network Throughput

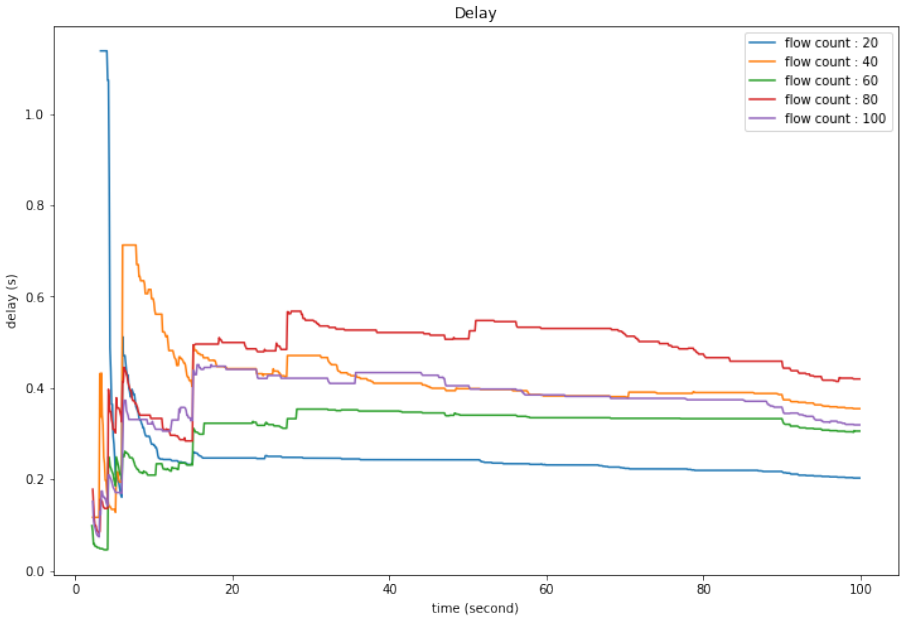

Figure 25: End to End Delay

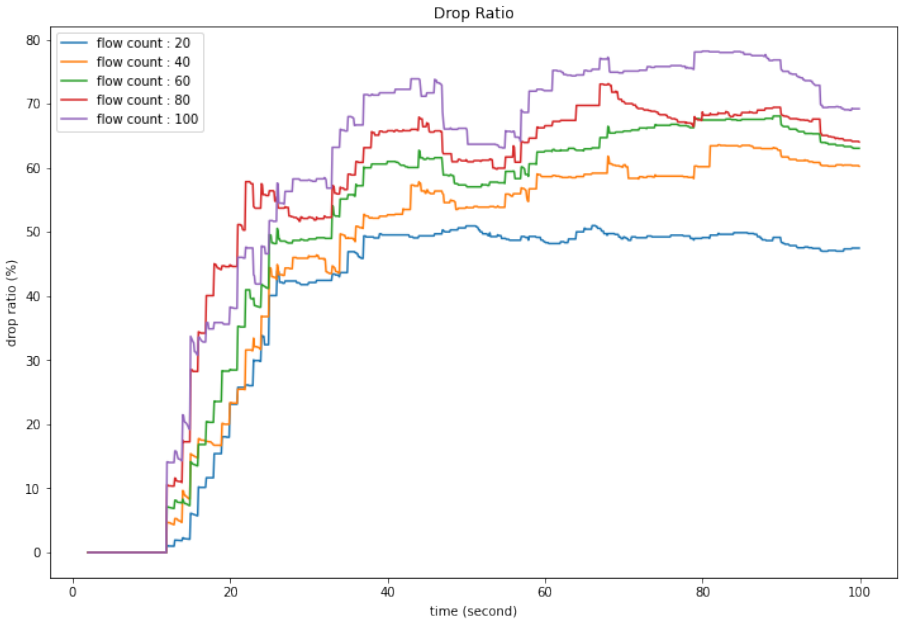

Figure 26: Packet Drop Ratio

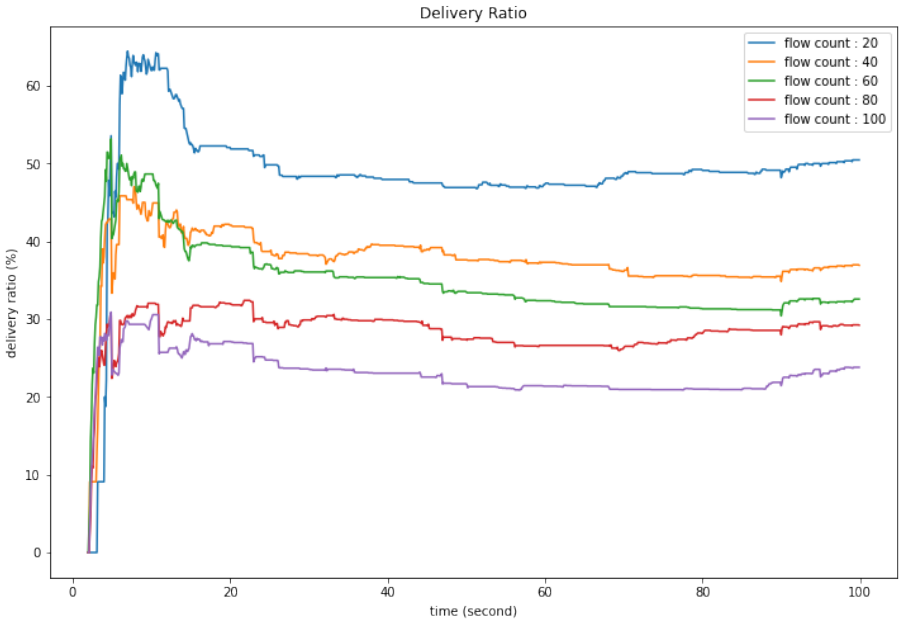

Figure 27: Packet Delivery Ratio

Vary packets per second

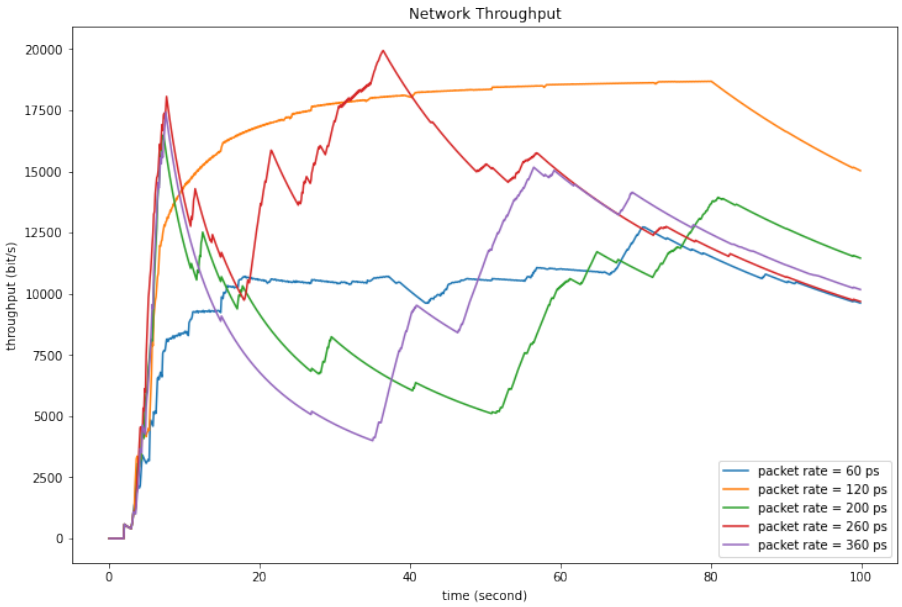

Figure 28: Network Throughput

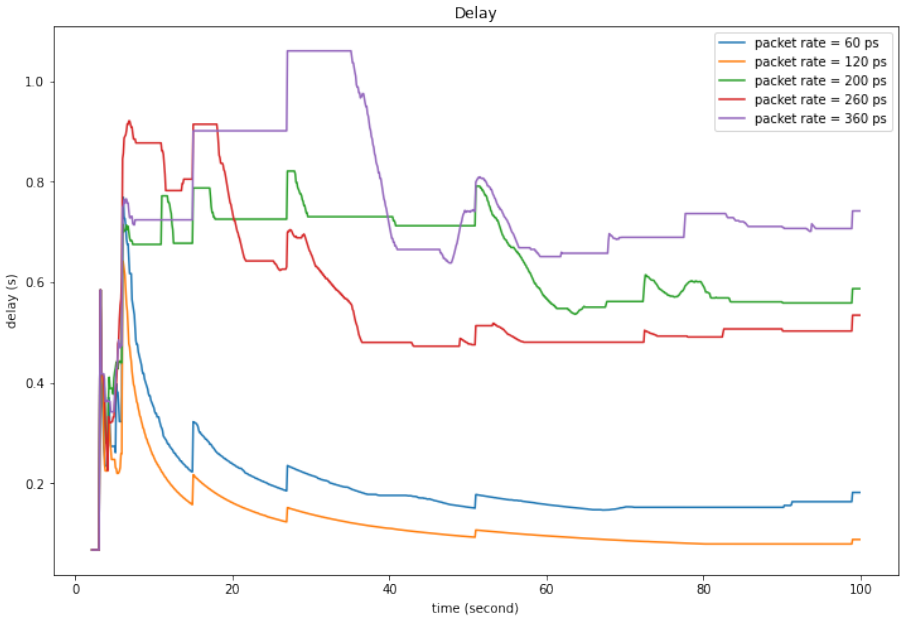

Figure 29: End to End Delay

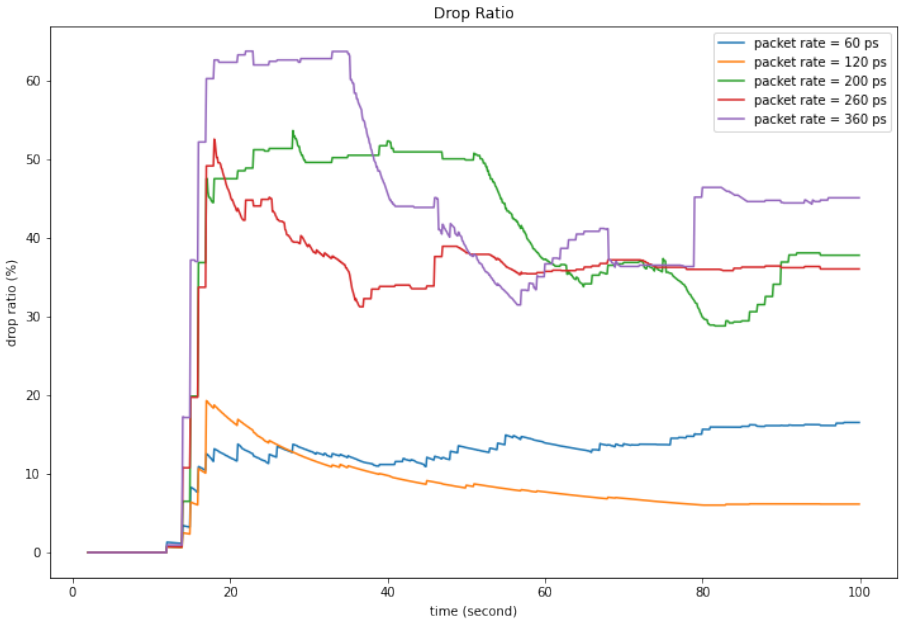

Figure 30: Packet Drop Ratio

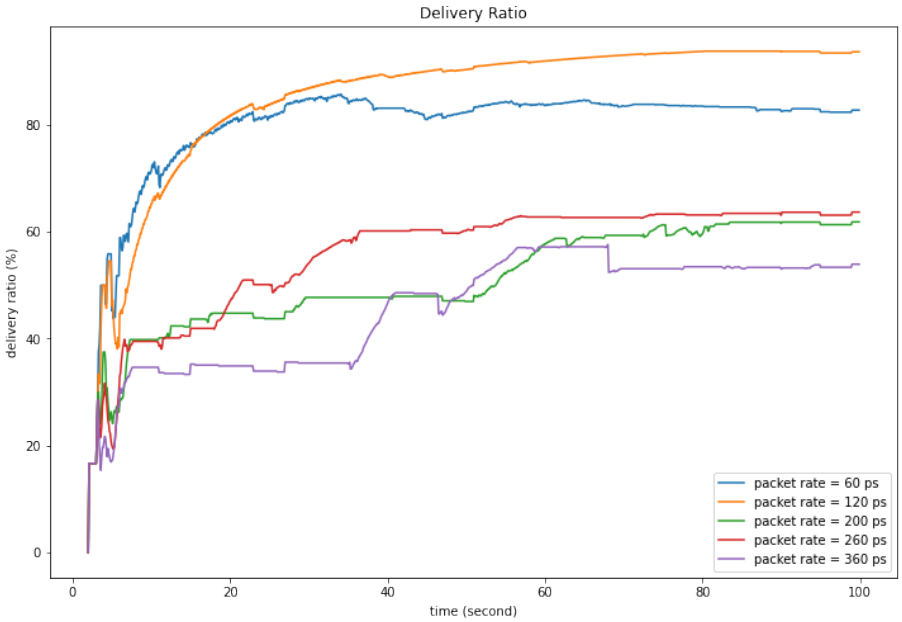

Figure 31: Packet Delivery Ratio

Vary coverage

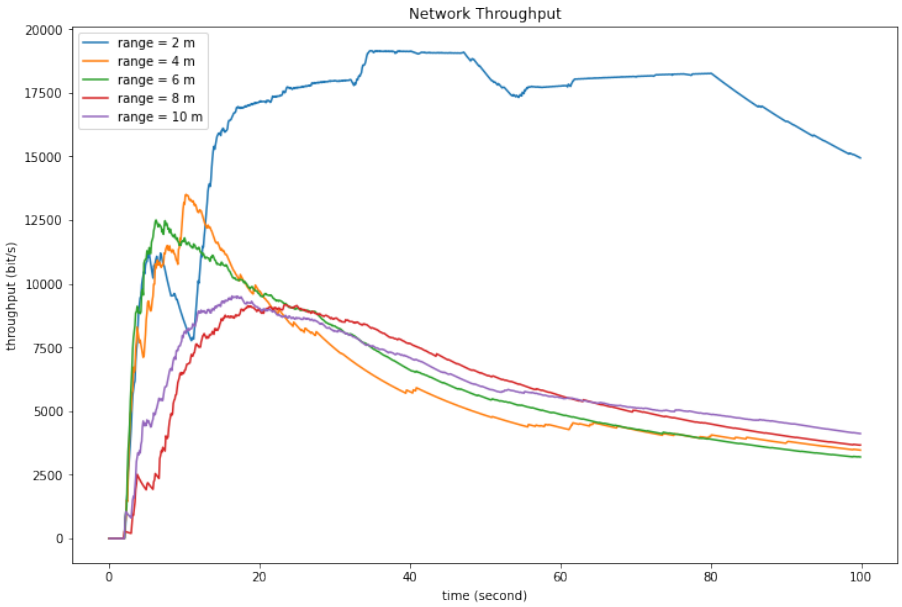

Figure 32: Network Throughput

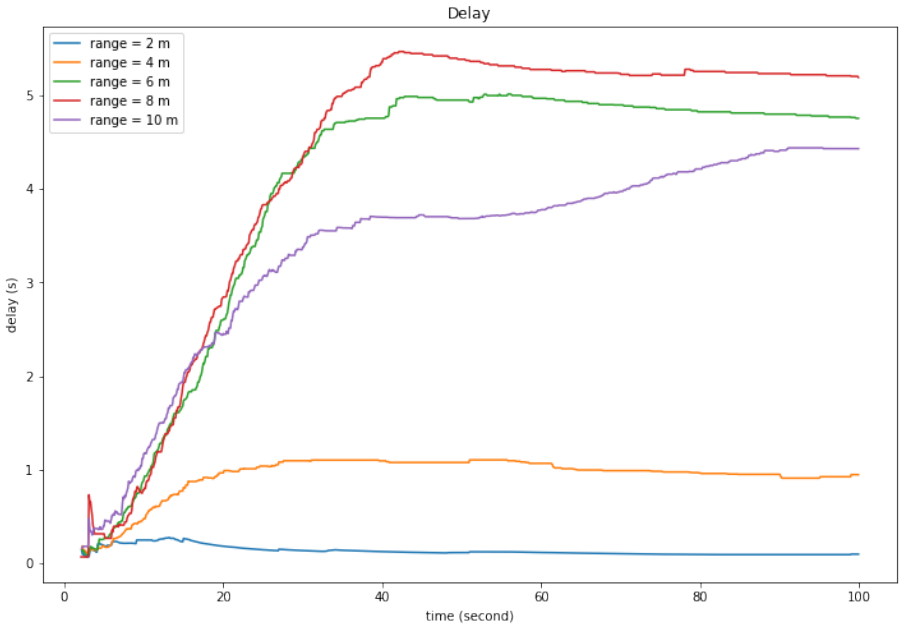

Figure 33: End to End Delay

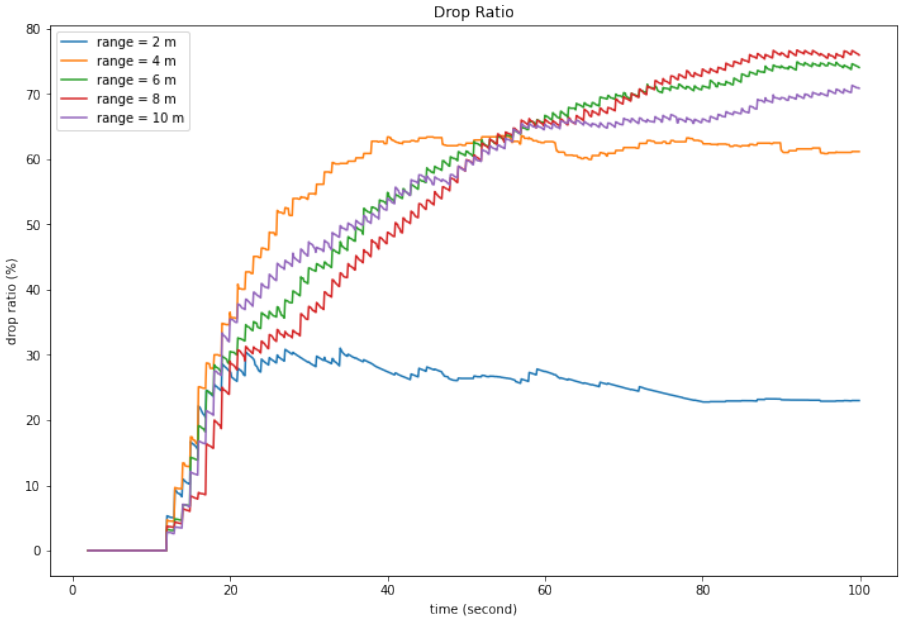

Figure 34: Packet Drop Ratio

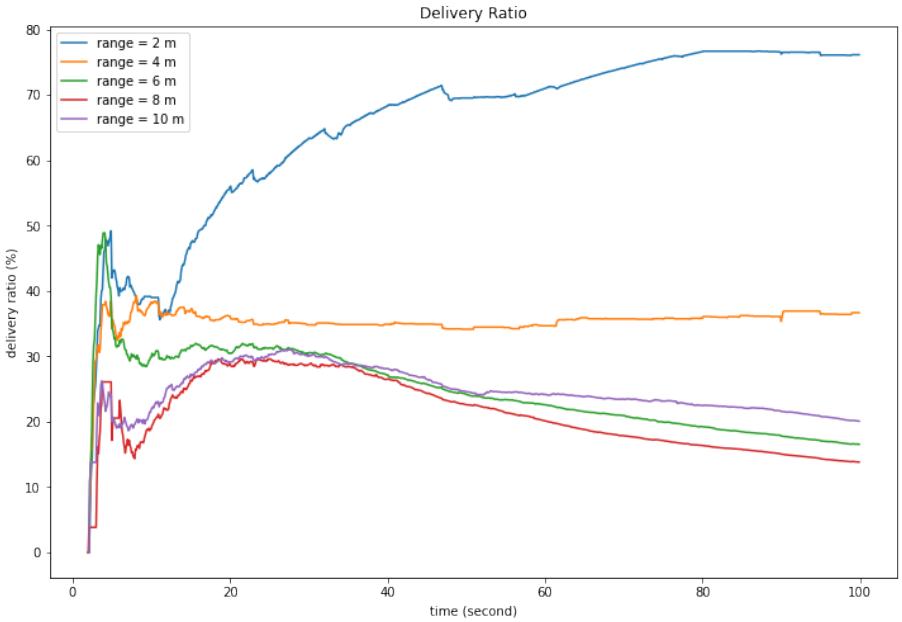

Figure 35: Packet Delivery Ratio

Variation of Parameters (Discussion)

We see that wireless low rate behaves very differently from wireless low rate. This is partly because we increased the capacity of our bottleneck link and also because large number of packets are lost (dropped) when simulating using LRWPAN.

When node numbers are increased, **throughput** increases initially but decreases later which is due to the high network traffic. Same thing also happens when flow is increased. When packets per second is increased within proper range, network throughput always increase.

In case of **end to end delay**, the general trend is that it increases with the number of nodes, flow, packets per second and coverage. This is intuitive since the traffic is larger now. And in case of this is because more nodes are in consideration, and they are at a larger distance.

In case of **packet drop ratio**, it starts with 0% and as time increases the drop rate also increases. As the number of nodes, flow, packets per second and coverage is increased, packet drop almost always increase because of the larger network traffic.

For **packet delivery ratio**, the trend is exactly as opposite as expected. Initially, the delivery rate is 100%. As the drop rate increases, the delivery rate decreases. As the parameters are increased, packet delivery rate almost always decrease because of the larger network traffic.

TASK B

Implementing Stabilized RED (SRED) [1]
and Extended Stabilized RED (ESRED) * using ns3

*An extended idea from SRED which we call it ESRED, implemented as a bonus part

Abstract

This report describes the methodology to implement Stabilized RED (SRED) Algorithm in ns3. Like RED (Random Early Detection) SRED pre-emptively discards packets with a load-dependent probability when a buffer in a router in the Internet or an Intranet seem congested. SRED has an additional feature that over a wide range of load levels helps it stabilize its buffer occupation at a level independent of the number of active connections.

We also implement an extended version of SRED where we also consider timestamp of the incoming packets in our algorithm and adjust the probability to overwrite accordingly. We call this Extended Stabilized RED or ESRED.

Algorithm Overview

A simple way of comparing an arriving packet with a recent other packet is to compare it with a packet still in the buffer. This makes it impossible to compare packets more than one buffer drain time apart. To give the system longer memory, we augment the information in the buffer with a `zombie list`. We can think of this as a list of `M` recently seen flows, with the following extra information for each flow in the list: a `count` and a `timestamp`. Note that this zombie list or flow cache is small and maintaining this list is not the same as maintaining per-flow state. We call the flows in the zombie list `zombies`.

The zombie list starts out empty. As packets arrive, as long as the list is not full, for every arriving packet the packet flow identifier (`source address`, `destination address`, etc.) is added to the list, the `count` of that zombie is set to zero, and its `timestamp` is set to the arrival time of the packet.

Once the `zombie list` is full it works as follows: Whenever a packet arrives, it is compared with a randomly chosen `zombie` in the `zombie list`.

- **Hit** : If the arriving packet's flow matches the zombie we declare a "hit". In that case, the `count` of the zombie is increased by one, and the `timestamp` is reset to the arrival time of the packet in the buffer.
- **No Hit** : If the two are not of the same flow, we declare a "no hit". In that case, with probability $p_{overwrite}$ the flow identifier of the packet is overwritten over the zombie chosen for comparison. The `count` of the zombie is set to 0, and the `timestamp` is set to the arrival time at the buffer. With probability $1 - p_{overwrite}$ there is no change to the zombie list.

Irrespective of whether there was a hit or not, the packet may be dropped if the buffer occupancy is such that the system is in random drop mode i.e it will

depend on p_{zap}. The drop probability p_{zap} may depend on whether there was a hit or not.

We maintain an estimate $P_{hit}(t)$ for the hit frequency around the time of the arrival of the t-th packet at the buffer. For the t-th packet, let

$$\text{Hit}(t) = \begin{cases} 0 & \text{if no hit} \\ 1 & \text{if hit} \end{cases}$$

and let

$$P_{hit}(t) = (1-\alpha)P_{hit}(t-1) + \alpha \text{ Hit } (t)$$

where,

$$\alpha \sim \frac{p_{overwrite}}{M}.$$

Here, $0 < \alpha < 1$, and M is the size of `zombie list`

We have a buffer of capacity B bytes. A function $p_{\text{sred}}\ (q)$ is defined as follows:

$$p_{\text{sred}}\ (q) = \begin{cases} p_{\max} & \text{if } \frac{1}{3}B \leq q < B \\ \frac{1}{4} \times p_{\max} & \text{if } \frac{1}{6}B \leq q < \frac{1}{3}B \\ 0 & \text{if } 0 \leq q < \frac{1}{6}B \end{cases}$$

For **Simple SRED** mode, p_{zap} is calculated is as follows

$$p_{zap} = p_{sred}(q) \times \min\left(1, \frac{1}{(256 \times P_{hit}(t))^2}\right)$$

and for **Full SRED** mode,

$$p_{zap} = p_{\text{sred}}\ \times \min\left(1, \frac{1}{(256 \times P_{hit}(t))^2}\right) \times \left(1 + \frac{Hit(t)}{P_{hit}(t)}\right)$$

The pseudocode of the described algorithm is given below.

```
function processPackets():
  zombie_list = []
  for each packet p:
    hash_value = hash(p)
    if zombie_list not full:
      zombie.flowId = hash_value
      zombie.count = 0
      zombie_list.insert(zombie)
    else:
      idx = random index from zombie_list
      if zombie_list[idx].flowId == hash_value :
        hit = 1
        zombie_list[idx].count++
      else:
        r = random probability
        if r < p_overwrite:
          zombie_list[idx].flowId = hash_value
          zombie_list[idx].count = 0

      p_hitFreq = (1 - alpha) * p_hitFreq + alpha * hit

  p_sred = calculatePSred()
  p_zap = calculatePZap(p_sred,hit,p_hitFreq,mode)

  r = random probability
  if r < p_zap:
    drop(p)
  else:
    if p does not overflow buffer capacity:
      enqueue(p)
    else:
      drop(p)
```

Listing 1: SRED Pseudocode

Implementation in ns3

To implement SRED in ns3 we need to create a new queue disc class.

Every queue disc collects statistics about the total number of packets/bytes received from the upper layers (in case of root queue disc) or from the parent queue disc (in case of child queue disc), enqueued, dequeued, requeued, dropped, dropped before enqueue, dropped after dequeue, marked, and stored in the queue disc and sent to the netdevice or to the parent queue disc. Note that packets that are dequeued may be requeued, i.e., retained by the traffic control infrastructure, if the netdevice is not ready to receive them. Requeued packets are not part of the queue disc. The following identities hold:

- dropped = dropped before enqueue + dropped after dequeue
- received = dropped before enqueue + enqueued
- queued = enqueued - dequeued
- sent = dequeued - dropped after dequeue (- 1 if there is a requeued packet)

A C++ abstract base class, class QueueDisc, is subclassed to implement a specific queue disc. A subclass is required to implement the following methods:

- `bool DoEnqueue(Ptr<QueueDiscItem>item)`: Enqueue a packet
- `Ptr<QueueDiscItem> DoDequeue (void)`: Dequeue a packet
- `bool CheckConfig (void) const`: Check if the configuration is correct
- `void InitializeParams (void)`: Initialize queue disc parameters

We implement all of these functions for SRED. A brief overview of the implementation is given below.

Full source code can be found at https://github.com/zarif98sjs/SRED-ESRED-ActiveQueueManagement-ns3.

In ns3 traffic-control module, we add a header class `stabilized-red-queue-disc.h` and model `stabilized-red-queue-disc.cc` in the `/src/traffic-control/model` directory

We declare a zombie `struct` in the header file.

```
struct Zombie
{
  int32_t flowID;
  int32_t count;
};
```

We now write a new derived class `StabilizedRedQueueDisc`

```
class StabilizedRedQueueDisc : public QueueDisc
{
public:
  static TypeId GetTypeId (void);
  StabilizedRedQueueDisc ();
  virtual ~StabilizedRedQueueDisc ();
  int64_t AssignStreams (int64_t stream);

protected:
  virtual void DoDispose (void);

private:
  virtual bool DoEnqueue (Ptr<QueueDiscItem> item);
  virtual Ptr<QueueDiscItem> DoDequeue (void);
  virtual bool CheckConfig (void);
  virtual void InitializeParams (void);

virtual Ptr<const QueueDiscItem> DoPeek (void);

  double calculateProbabilityStabilizedRed();
  double calculateProbabilityZapSimple(double
    probabilityStabilizedRed);
  double calculateProbabilityZap(double probabilityStabilizedRed,
    int32_t hit);

  vector<Zombie> zombies;
  double p_hitFreq;
  double p_overwrite;
  double p_max;
  double alpha;
```

```
  int32_t stabilizedRedMode;

  Ptr<UniformRandomVariable> m_uv;
};
```

Now we move on to implement the functions that are necessary to implement SRED.

First we need to register the object subclass that we are creating with the TypeId system. This can be done using

```
NS_OBJECT_ENSURE_REGISTERED (StabilizedRedQueueDisc);
```

and after that define the log component as follows -

```
NS_LOG_COMPONENT_DEFINE ("StabilizedRedQueueDisc");
```

First thing to do after that is add the attributes associated with SRED for example we set the attribute for $p_{overwrite}$, give it with 0.25 default value. In the same way, we set the value for p_{max} or maximum drop probability as 0.15. Consequently we limit the value of maximum number of packets that the queue can accept using the $MaxSize$ attribute. Finally, we set the default mode to be Simple SRED. All of the default values can be changed later.

```
TypeId
StabilizedRedQueueDisc::GetTypeId (void)
{
  static TypeId tid =
  TypeId ("ns3::StabilizedRedQueueDisc")
  .SetParent<QueueDisc> ()

  .SetGroupName ("TrafficControl")

  .AddConstructor<StabilizedRedQueueDisc> ()

  .AddAttribute ("OverwriteProbability", "Probability of
    overwriting zombie list",
     DoubleValue (0.25),
     MakeDoubleAccessor (&StabilizedRedQueueDisc::p_overwrite),
     MakeDoubleChecker<double> (0, 1))

  .AddAttribute ("MaximumDropProbability", "maximum probability of
    dropping a packet",
     DoubleValue (0.15), MakeDoubleAccessor (&
    StabilizedRedQueueDisc::p_max),
     MakeDoubleChecker<double> (0, 1))

  .AddAttribute ("MaxSize",
    "The maximum number of packets accepted by this queue disc",
    QueueSizeValue (QueueSize ("25p")),
    MakeQueueSizeAccessor (&QueueDisc::SetMaxSize,&QueueDisc::
    GetMaxSize),
    MakeQueueSizeChecker ())

  .AddAttribute ("StabilizedRedMode", "Simple  or Full",
    IntegerValue (1),
    MakeIntegerAccessor (&StabilizedRedQueueDisc::stabilizedRedMode
    ),
     MakeIntegerChecker<int32_t> ());
  return tid;
}
```

We initialize rest of the variables associated with SRED that can't be direct changed from the user end.

```
void
StabilizedRedQueueDisc::InitializeParams (void)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_INFO ("Initializing Stabilized RED params.");
  zombies = vector<Zombie> ();
  p_hitFreq = 0;
  alpha = p_overwrite / MAX_ZOMBIE_LIST_SIZE;
}
```

We also implement the `CheckConfig` function to check if there are no queue size

mismatch and also the number of queue disc classes is not greater than 0

```
bool
StabilizedRedQueueDisc::CheckConfig (void)
{
  NS_LOG_FUNCTION (this);
  if (GetNQueueDiscClasses () > 0)
    {
      cout<<"StabilizedRedQueueDisc cannot have classes"<<endl;
      NS_LOG_ERROR ("StabilizedRedQueueDisc cannot have classes");
      return false;
    }
  if (GetNInternalQueues () == 0)
    {
      // add a DropTail queue
      AddInternalQueue (CreateObjectWithAttributes<DropTailQueue<
    QueueDiscItem>> (
            "MaxSize", QueueSizeValue (GetMaxSize ())));
    }

  if (GetNInternalQueues () != 1)
    {
      NS_LOG_ERROR ("StabilizedRedQueueDisc needs 1 internal queue"
    );
      return false;
    }
  return true;
}
```

Using the equation for p_{sred} defined above, we write a function to calculate that

```
// calculate p_sred
double
StabilizedRedQueueDisc::calculateProbabilityStabilizedRed()
{
    uint32_t bufferCapacity = GetInternalQueue (0)->GetMaxSize ().
    GetValue ();
    uint32_t q = GetInternalQueue (0)->GetNBytes ();

    double bufferCapacity_3 = (double) bufferCapacity / 3.0;
    double bufferCapacity_6 = (double) bufferCapacity / 6.0;

    if(bufferCapacity_3 <= q && q <= bufferCapacity){
        return p_max;
    }
    else if(bufferCapacity_6 <= q && q < bufferCapacity_3){
        return p_max/4;
    }
    else{
        return 0;
    }
}
```

and subsequently we calculate for p_{zap} for both Simple and Full SRED

```
// calculate p_zap simple
double
StabilizedRedQueueDisc::calculateProbabilityZapSimple(double
    probabilityStabilizedRed)
{
    double multiply = 1.0/(256*p_hitFreq*p_hitFreq);

    if(multiply < 1)
    {
        return probabilityStabilizedRed * multiply;
    }
    else
    {
        return probabilityStabilizedRed;
    }
}
```

```
// calculate p_zap full
double
StabilizedRedQueueDisc::calculateProbabilityZap(double
    probabilityStabilizedRed,int32_t hit)
{
    double multiply1 = 1.0/(256*p_hitFreq*p_hitFreq);
    double multiply2 = 1 + (hit/p_hitFreq);

    if(multiply1 < 1)
    {
        return probabilityStabilizedRed * multiply1 * multiply2;
    }
    else
    {
        return probabilityStabilizedRed * multiply2;
    }
}
```

Finally, the most important part, the `Enqueue` function. In this function, we implement the main algorithm of SRED. First we classify the packet or calculate its hash value. If it doesn't match any filter, we drop it. Otherwise we proceed with the SRED algorithm.

```
bool
StabilizedRedQueueDisc::DoEnqueue (Ptr<QueueDiscItem> item)
{
  NS_LOG_FUNCTION (this << item);

  uint32_t flowHash;
  if (GetNPacketFilters () == 0)
  {
    flowHash = item->Hash (0);
    // cout<<"flowHash: "<<flowHash<<endl;
  }
  else
  {
    int32_t ret = Classify (item);

    if (ret != PacketFilter::PF_NO_MATCH) flowHash = static_cast<
    uint32_t> (ret);
    else{
     NS_LOG_ERROR ("No filter has been able to classify this packet
    , drop it.");
     DropBeforeEnqueue (item, "PAcket filter no match");
     return false;
    }
  }

  int32_t flowID = flowHash;
  uint32_t nQueued = GetInternalQueue (0)->GetCurrentSize ().
    GetValue ();
  uint32_t capacity = GetInternalQueue (0)->GetMaxSize ().GetValue
    ();

  if (flowID == -1) DropBeforeEnqueue (item, "InvalidFlowID");

  int32_t curZombieSize = zombies.size ();

  if (curZombieSize < MAX_ZOMBIE_LIST_SIZE){ // still has space in
    zombie list
    Zombie zombie;
    zombie.flowID = flowID;
    zombie.count = 0;
    zombies.push_back (zombie);

  if (nQueued + item->GetSize () <= capacity){
      // if adding packet size doesn't exceed capacity of queue,
    then enqueue
      NS_LOG_DEBUG("Enqueueing "<<item->GetPacket()->GetUid() <<"
    to queue "<<(uint32_t)flowID);
      return GetInternalQueue (0)->Enqueue (item);
  }
  else{
    // if adding packet size exceeds capacity of queue, then drop
    packet
    NS_LOG_DEBUG("Dropping "<<item->GetPacket()->GetUid()<<" due to
     queue overflow");
    DropBeforeEnqueue (item, "QUEUE_OVERFLOW");
    return false;
  }
 }
```

```
  else { // no more space in zombie list
  // choose a random zombie
  uint32_t index = m_uv->GetInteger (0, curZombieSize - 1);
  int32_t hit = 0;

  if (zombies[index].flowID == flowID) { // HIT
      hit = 1;
      zombies[index].count++;
  }
  else{ // not HIT
    // get a random value and compare with p_overwrite
    double rand = m_uv->GetValue ();
    if (rand < p_overwrite){
      zombies[index].flowID = flowID;
      zombies[index].count = 0;
    }
  }
  // update hit frequency
  p_hitFreq = (1 - alpha) * p_hitFreq + alpha * hit;
}

// calculate p_sred
double probabilityStabilizedRed = calculateProbabilityStabilizedRed
    ();

// calculate p_zap
double probabilityZap = 0;
if(stabilizedRedMode == 1)
{
    probabilityZap = calculateProbabilityZapSimple(
    probabilityStabilizedRed);
}
else if(stabilizedRedMode == 2)
{
    probabilityZap = calculateProbabilityZap(
    probabilityStabilizedRed,1);
}

// get a random value and compare with p_zap
double rand = m_uv->GetValue ();
if (rand < probabilityZap)
 {
    DropBeforeEnqueue (item, "Zap");
    return false;
 }
else
  {
    if(nQueued < capacity)
    {
        return GetInternalQueue (0)->Enqueue (item);
    }
    else
    {
        DropBeforeEnqueue (item, "Overflow");
        return false;
    }
  }
}
```

Lastly we implement the DoDequeue and DoPeek functions

```
Ptr<QueueDiscItem>
StabilizedRedQueueDisc::DoDequeue (void)
{
  NS_LOG_FUNCTION (this);

  if (GetInternalQueue (0)->IsEmpty ()){
    NS_LOG_LOGIC ("Queue empty");
    return 0;
  }
  else{
    Ptr<QueueDiscItem> item = GetInternalQueue (0)->Dequeue ();

    NS_LOG_LOGIC ("Popped " << item);

    NS_LOG_LOGIC ("Number packets " << GetInternalQueue (0)->
    GetNPackets ());
    NS_LOG_LOGIC ("Number bytes " << GetInternalQueue (0)->
    GetNBytes ());

    return item;
  }
}
```

```
Ptr<const QueueDiscItem>
StabilizedRedQueueDisc::DoPeek (void)
{
  NS_LOG_FUNCTION (this);
  if (GetInternalQueue (0)->IsEmpty ()){
    NS_LOG_LOGIC ("Queue empty");
    return 0;
  }

  Ptr<const QueueDiscItem> item = GetInternalQueue (0)->Peek ();

  NS_LOG_LOGIC ("Number packets " << GetInternalQueue (0)->
    GetNPackets ());
  NS_LOG_LOGIC ("Number bytes " << GetInternalQueue (0)->GetNBytes
    ());

  return item;
}
```

Building Topology

We simulate the SRED algorithm and compare it with RED in ns3 using a dumbbell topology.

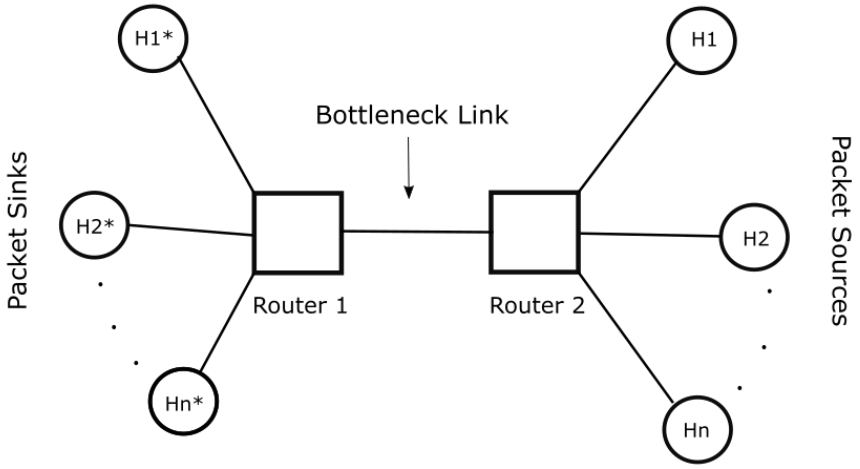

Figure 36: Simulation Topology

Network Specification

We have two routers connected through a point to point link (45Mbit/sec) with a link propagation delay of 1 msec. Host H1 sends packets to Host H1*, H2 to H2*, etc. and destinations send back acknowledgement packets as required by TCP. The bottleneck link in Router 2 has a buffer of capacity .5 Mbytes. The links between the hosts and routers are also point to point links. Finally we assign the IP as follows,

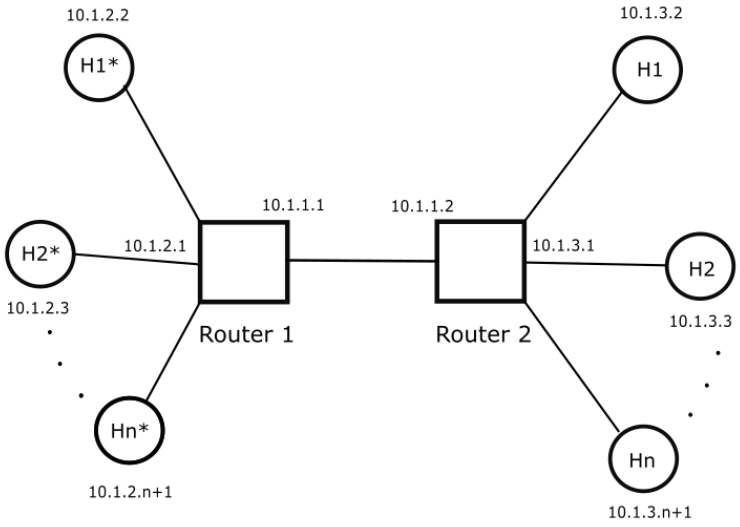

Figure 37: IP Assignment

Variation of Parameters (Results)

We vary number of leaf nodes in our dumbbell network to figure out how it affects **network throughput**, **end to end delay**, **packet drop ratio** and **packet delivery ratio**. We also look at SRED queue performance by measuring **queue packet drop rate** and **buffer occupancy**.

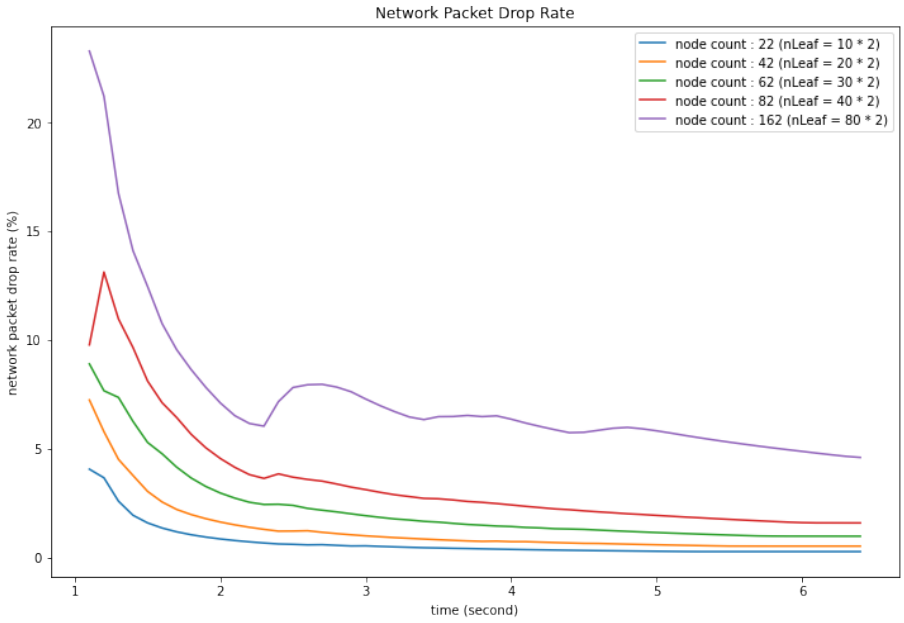

Figure 38: Network Packet Drop Ratio

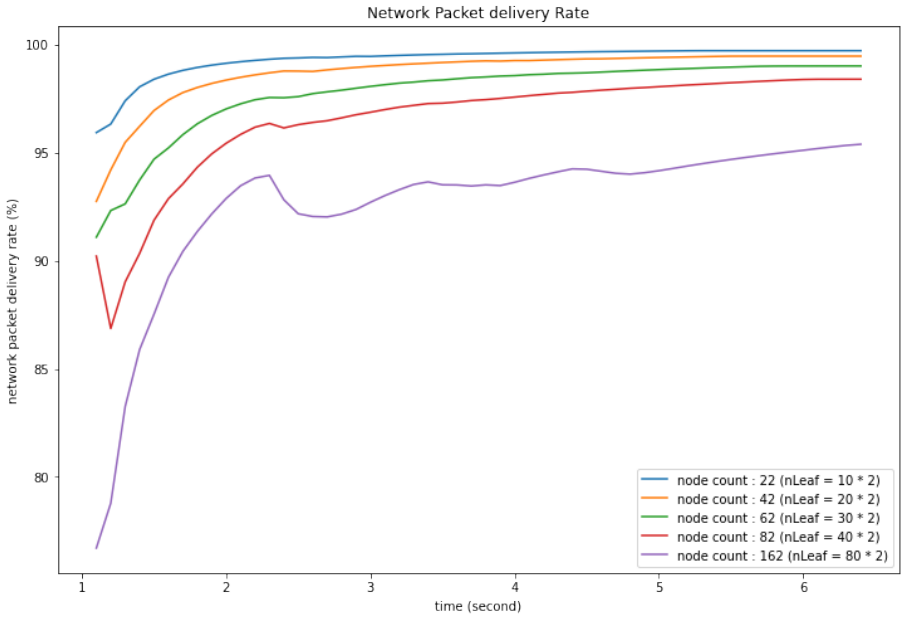

Figure 39: Network Packet Delivery Ratio

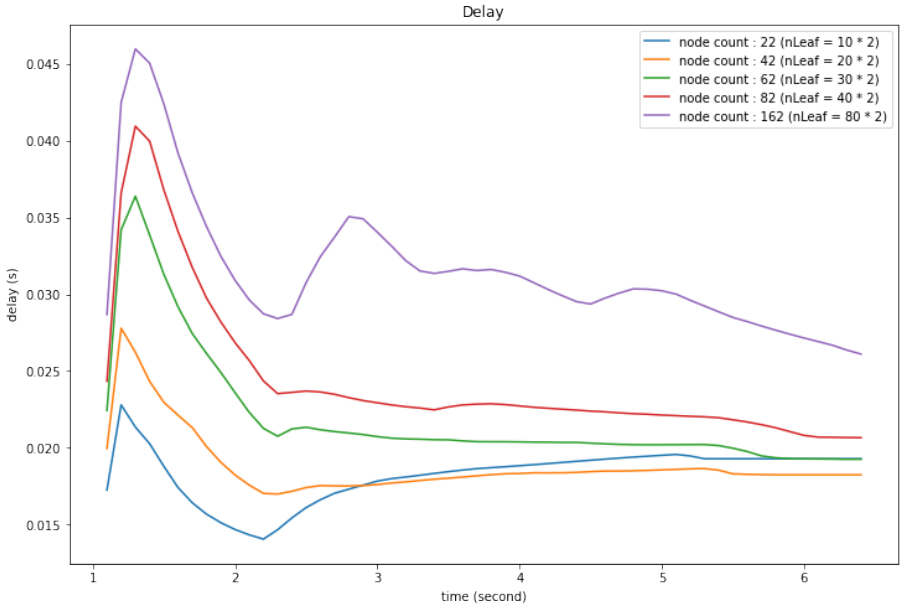

Figure 40: End to End Delay

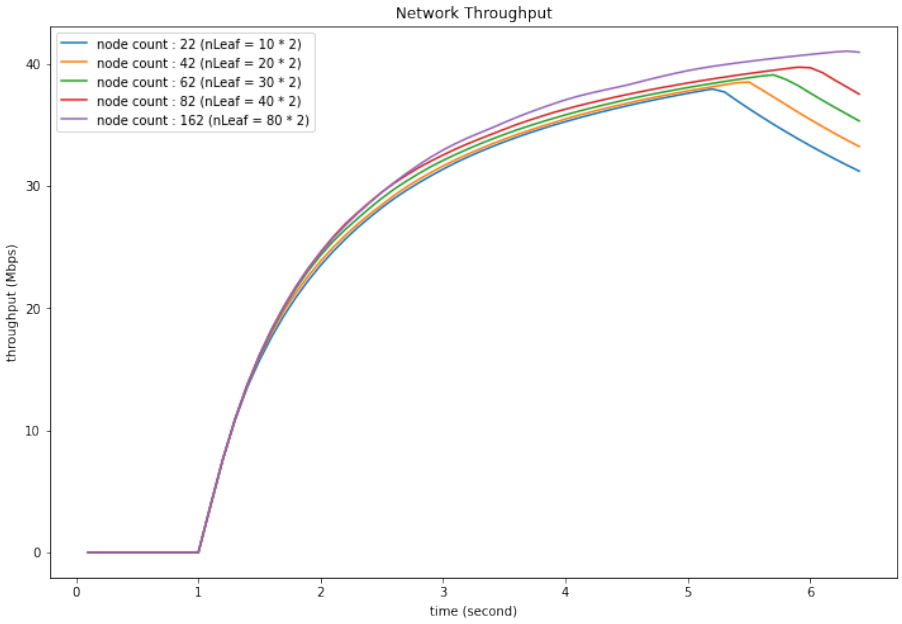

Figure 41: Network Throughput

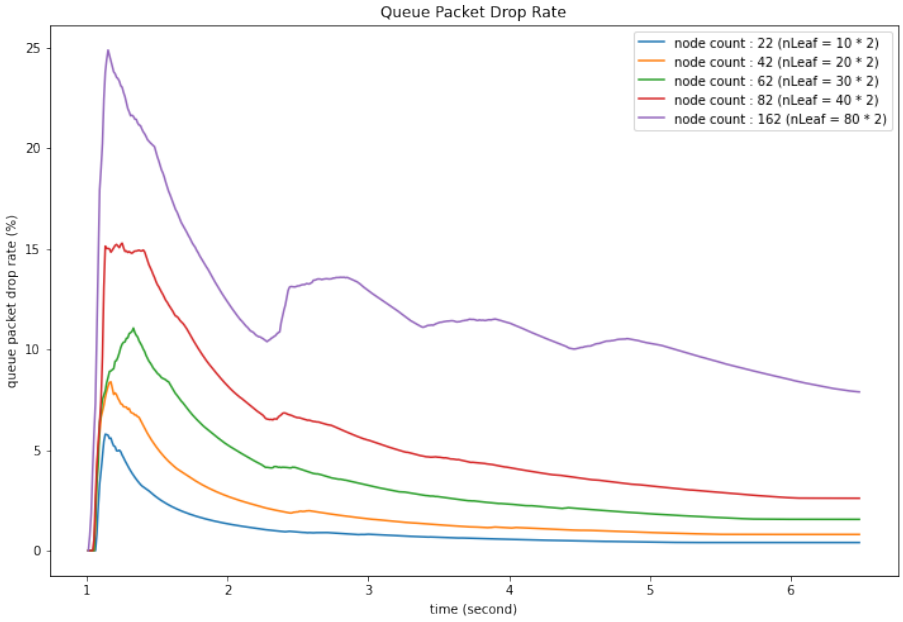

Figure 42: Queue Packet Drop Ratio

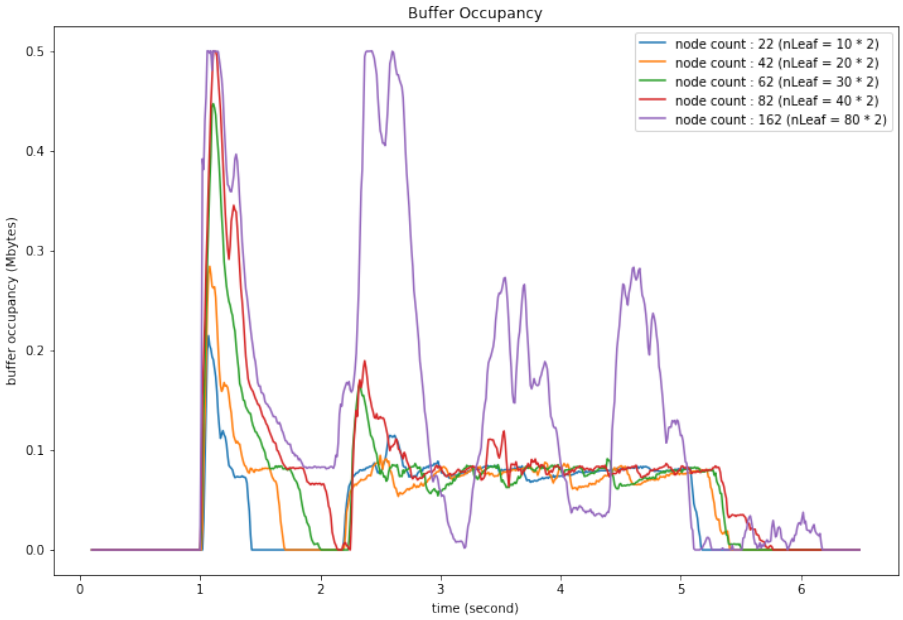

Figure 43: Buffer Occupancy

Variation of Parameters (Discussion)

In case of **packet drop ratio**, when the number of leaf nodes is increased in our dumbbell network, it results in higher network packet drop. This downward trend is expected because of the increase of network traffic.

For similar reason, we also see that the trend for **packet delivery ratio** is exactly the opposite to that of packet drop ratio. Packet delivery ratio decreases as the number of leaf nodes is increased from 10 to 80.

In case of **end to end delay**, the general trend is that it increases with the number of nodes. This is intuitive since we can't transfer more than the bottleneck bandwidth and that is why we get higher end to end delays.

The only exception is that **throughput** doesn't vary much. Throughput doesn't vary that much when number of nodes is changed. This can be explained with the existence of our bottleneck link in our network. No matter how much we increase our leaf nodes, or increase flow, or packets per second or even speed, throughput is almost always same.

When we look at some metric specific to queue, we see that **packet queue drop** also increases with number of nodes. The same holds true for **buffer occupancy**.

Comparison with RED

We compare the SRED algorithm with RED by measuring a few metrics like **network packet drop rate**, **packet delivery rate**, **throughput**, **buffer occupancy** and **queue drop rate**. Here the number of leaf node is 10 for both sides of our dumbbell network and rest of the configuration is the same as basic configuration mentioned above in the SRED section.

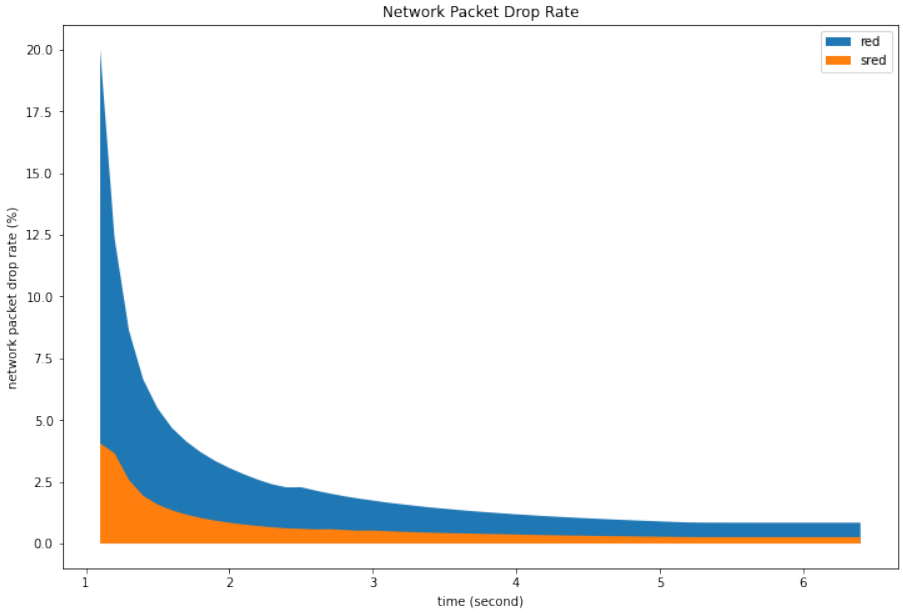

Figure 44: Network Packet Drop Ratio

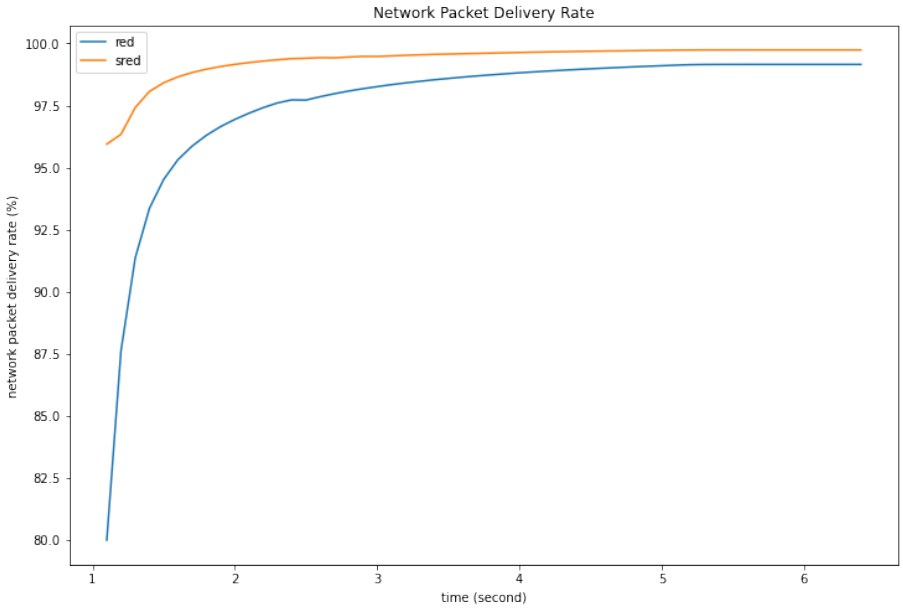

Figure 45: Network Packet Delivery Ratio

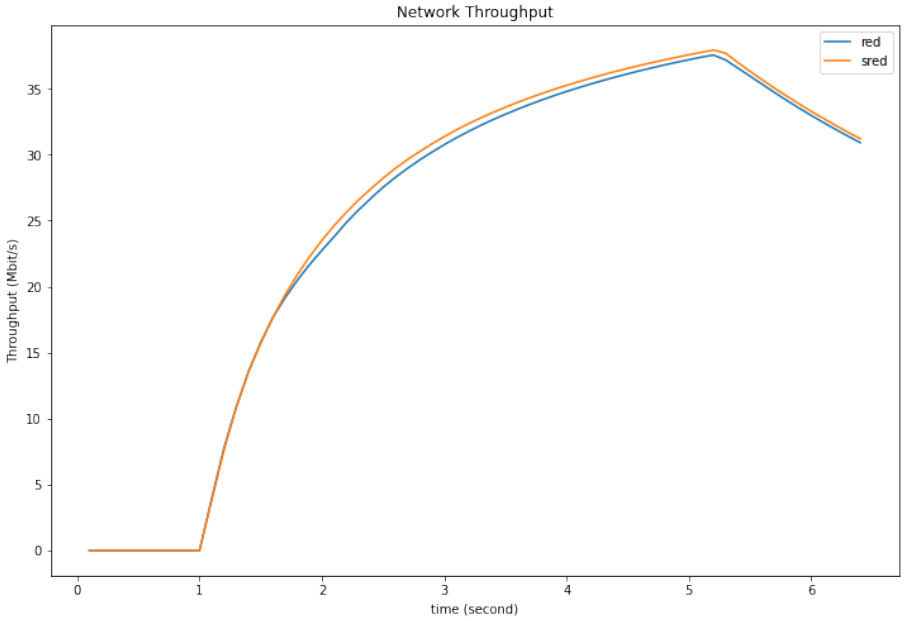

Figure 46: Network Throughput

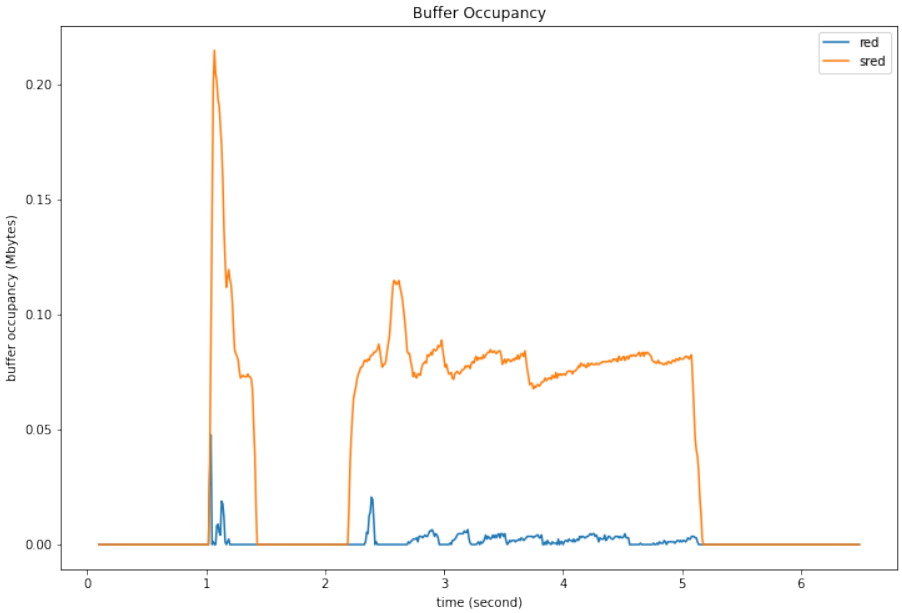

Figure 47: Buffer Occupancy

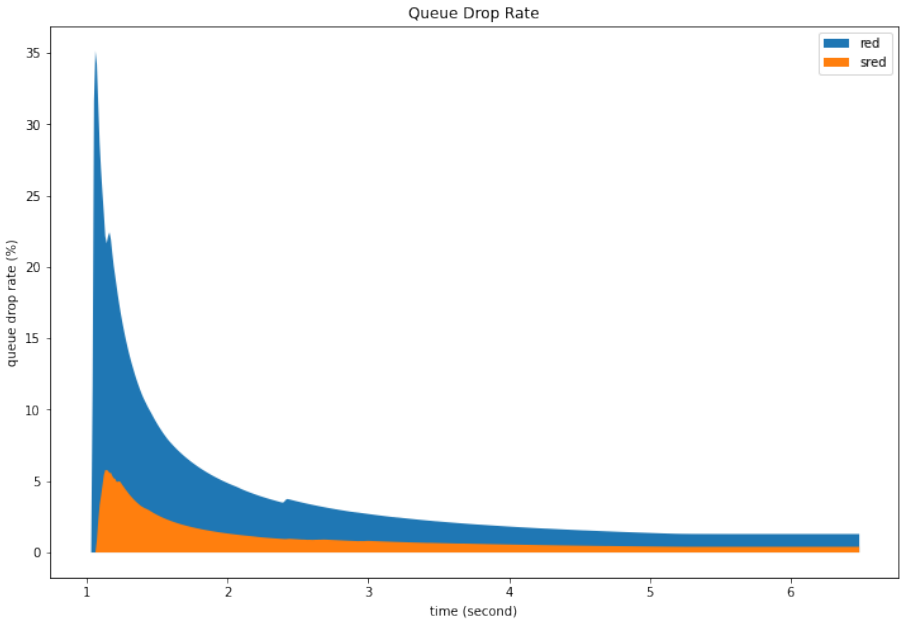

Figure 48: Queue Packet Drop Ratio

We see that SRED performs much better than its RED counterpart. The **queue drop rate** decreases from 1.2925% to 0.4% in SRED. In case of **network packet drop rate**, it decreases from 0.84% to 0.26%. This results in the increase of **delivery rate** from 99.15% to 99.73%.

SRED is more efficient in maintaining a queue within its capacity. It also maintains a higher packet delivery rate and lower packet drop rate while doing so. As the queue packet drop rate is low, we can see that the buffer occupancy is higher in case of SRED compared to RED.

Finally, if we look at network throughput, we can see that it increases from 37 Mbps to 38 Mbps at its peak. We don't see a large improvement because of the bottleneck present.

TASK B (BONUS PART)

Extended Stabilized RED (ESRED)[†] using ns3

[†]An extended idea from SRED which we call it ESRED, implemented as a bonus part

Idea

In the SRED paper, timestamp is not used as part of the experiments that have been conducted. Only the timestamps were stored for the zombie flows. We extend the SRED algorithm utilizing the timestamp of the zombie flows stored in our zombie list. We call this the Extended Stabilized RED algorithm or ESRED in short.

In the original SRED algorithm, a fixed $p_{overwrite}$ is used to decide if a zombie flow needs to be overridden when there is no hit. Instead of using a fixed $p_{overwrite}$ when there is no hit, we dynamically change it according to the flow timestamp we are comparing with. We modify this $p_{overwrite}$ so that it is higher for flows with older timestamp.

Comparison with SRED

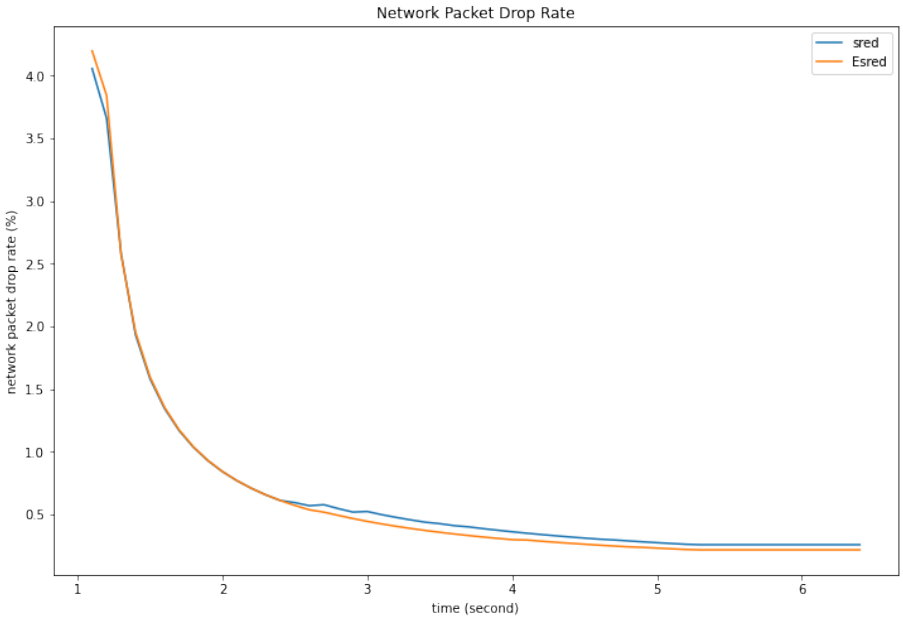

Figure 49: Network Packet Drop Ratio

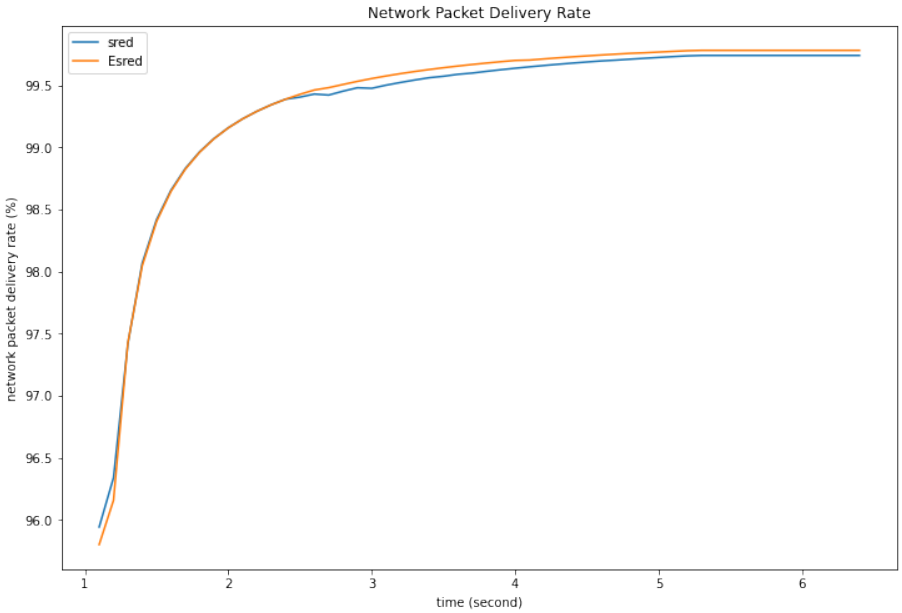

Figure 50: Network Packet Delivery Ratio

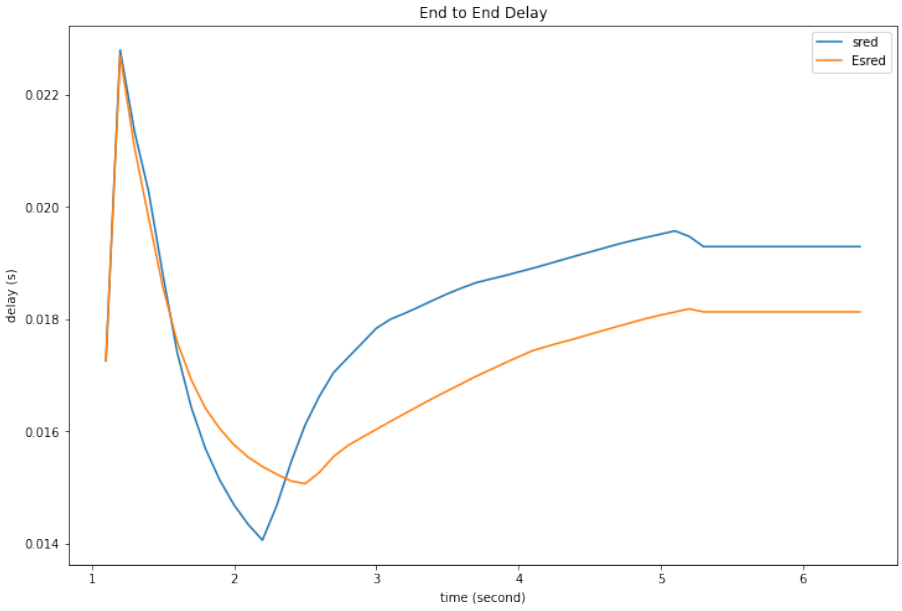

Figure 51: End to End Delay

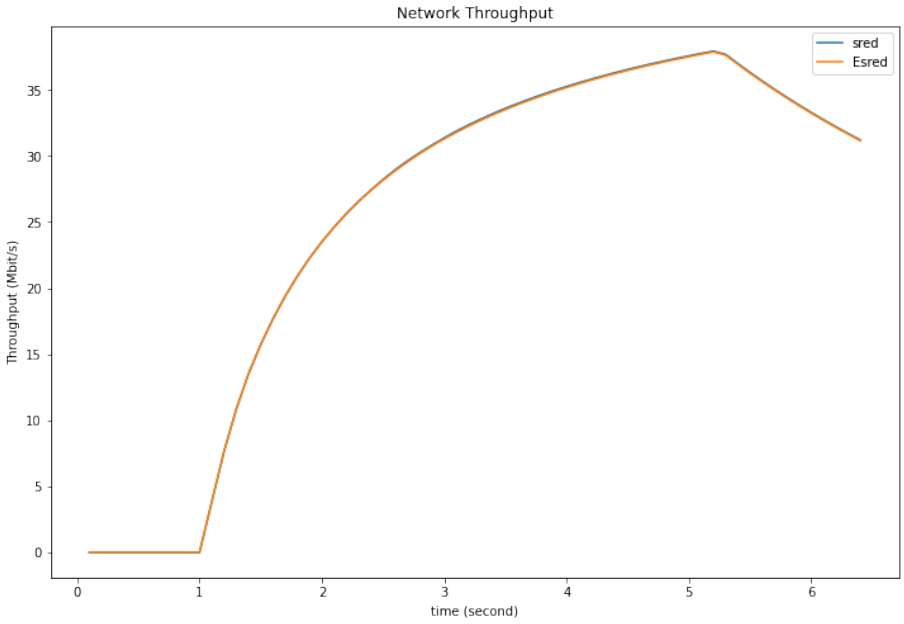

Figure 52: Network Throughput

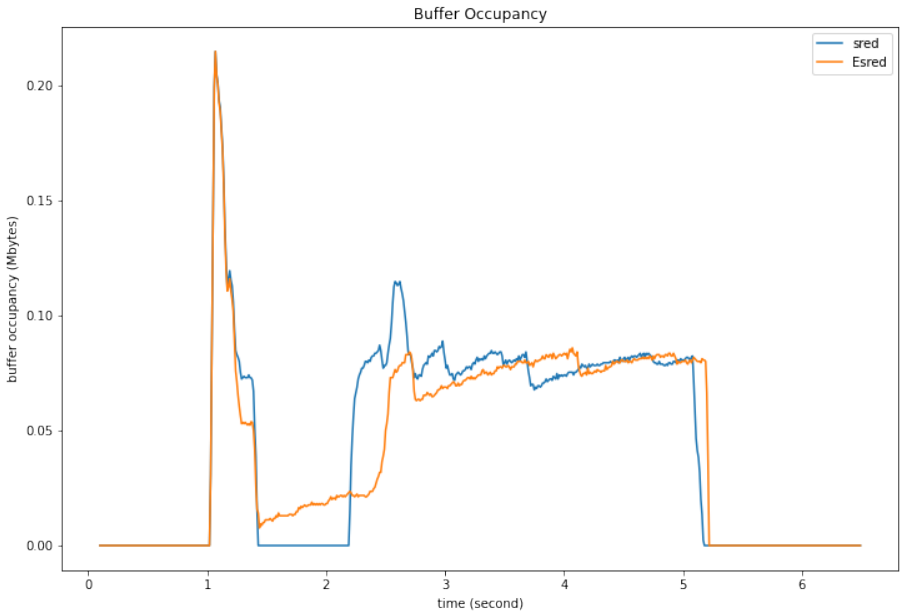

Figure 53: Buffer Occupancy

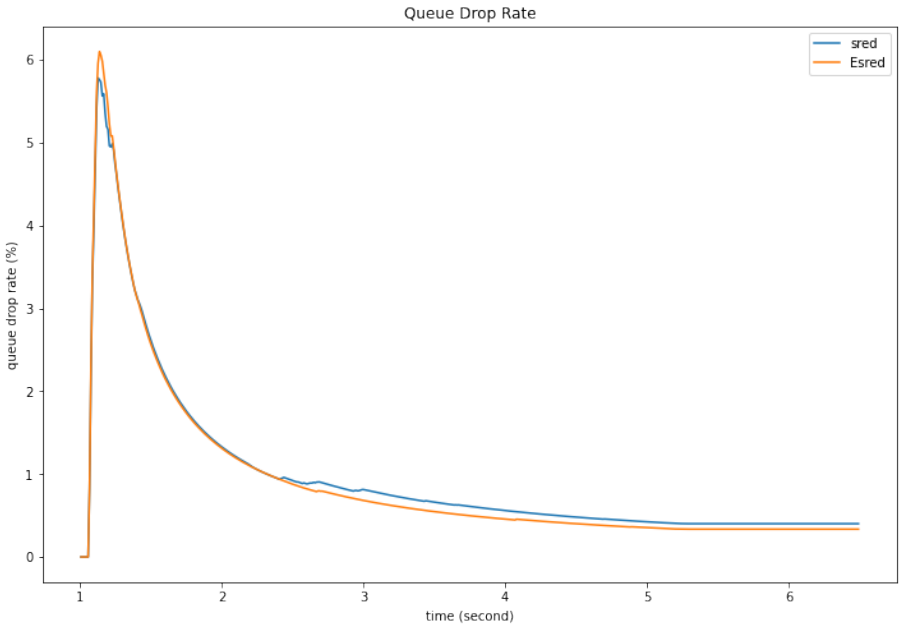

Figure 54: Queue Packet Drop Ratio

We see slight improvement in **network packet drop rate**, **network packet delivery rate** and also **queue drop rate**. **Throughput** and **buffer occupancy** remains almost same and we get a slightly larger improvement in case of end to end delay.

Reference

[1] Teunis J Ott, TV Lakshman, and Larry H Wong. "Sred: stabilized red". In: *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*. Vol. 3. IEEE. 1999, pp. 1346–1355.

[2] George F Riley and Thomas R Henderson. "The ns-3 network simulator". In: *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.