A report on

ns-3 Project

by

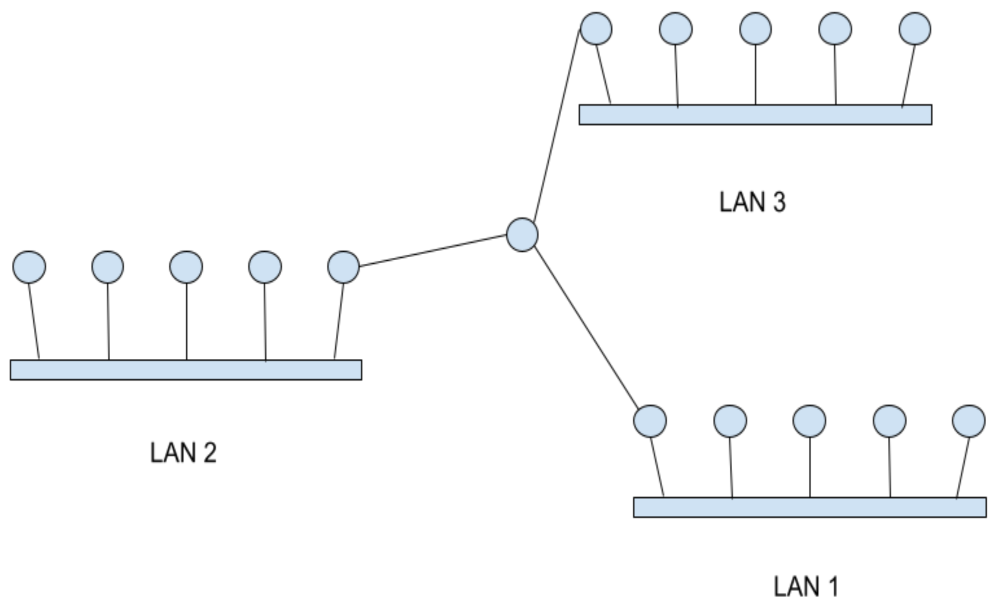
Nishat Farhana Purbasha
1705067

Network Topology Description

Task A (Part-1)

Wired :

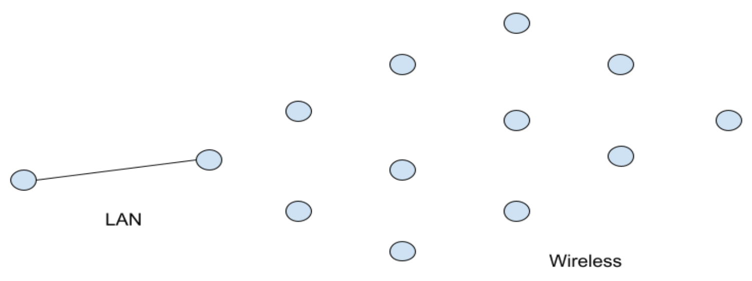
The topology used in the wired network is shown below. The number of nodes in LANs varied and the major flows in the network were from LAN1 to LAN2 and LAN2 to LAN3. There are a total six networks in this topology.



Task A (Part-2)

Wireless Low Rate (802.15.4) (static) :

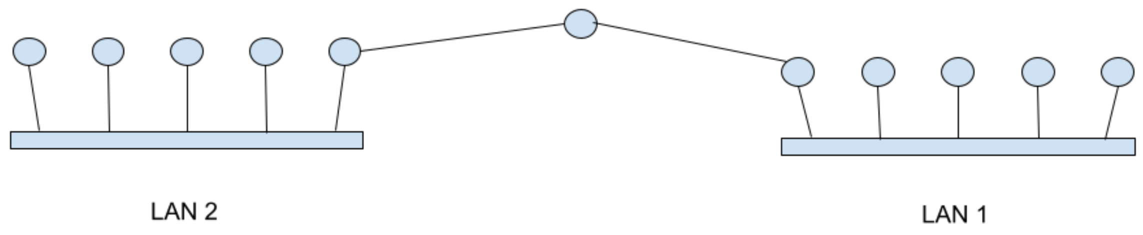
The topology used in the wireless low rate network is shown below. The number of nodes in LAN is two and The number of nodes in the wireless network was varied. The major flows in the network were from wireless network to LAN. There are a total two networks in this topology. The nodes in the wireless network were allocated using the grid position allocator.



Task B

Wired :

The topology used in the wired network is shown below. The number of nodes in LANs varied and the major flows in the network were from LAN1 to LAN2. There are a total four networks in this topology.



Variation of Parameters

In Wired -

- Number of Nodes
- Number of Flows in the Network
- Number of Packets sent Per second

In Wireless Low Rate(802.15.4) (static)-

- Number of Nodes
- Number of Flows in the Network
- Number of Packets sent Per second
- Coverage Area (Static Nodes)

In Modified Algorithm (Wireled) -

- Number of Nodes

Overview of the proposed Algorithm

The proposed algorithm is a congestion control algorithm titled “RR-TCP: A Reordering-Robust TCP with DSACK”. It is a modified Congestion Control algorithm for TCP with misinterpretation of out-of-order packets as packet loss. It offers significantly enhanced throughput on reordering paths.

With the help of the algorithm, detection and recovery from false fast retransmits using DSACK Information is done. To avoid false fast retransmits, the algorithm adaptively balances increasing dupthresh. This algorithm also limits the growth of dupthresh to avoid unnecessary timeouts. These enhancements improve the performance of TCP over paths that reorder or delay packets.

Measuring Reordering Length, r

```
$c$ = greatest contiguously ACKed packet number
$m$ = greatest ACK or SACK number received so far
$n = 0$
foreach packet $k$ such that $c < k \leq i$
    if current ACK newly ACKs or SACKs $k$
    then
        $h = k$        // found a hole
        $n = n + 1$
    endif
end
if $n == 1$ then
    $r = m - h$
endif
```

False Fast Retransmit Avoidance

The FA ratio is denoted by percentage of reordering. The value of dupthresh is measured using the percentile value in the reordering length cumulative distribution. Increasing the FA ratio increases the dupthresh and vice versa.

Various Cost Function

RTO: $C(\text{true timeout}) = W(\frac{T}{R} + \log_2 W - k - 2) + 1$ packets

False Fast Retransmits: $C(\text{false fast retransmit}) \leq \sum_{i=0}^{k-1} [\frac{W}{2} - i] = \frac{k(W-k+1)}{2}$ packets

Limited Transmit: $C(\text{limited transmit}) = \frac{I}{R}W - d.$ packets

Where, W = steady state window size, R = smoothed RTT, T = retransmission timeout period,

I = idle period, d = number of further duplicate ACKs that return.

Adapting the FA Ratio

S of 0.01 is set.

Upon every false fast retransmit detected, increase the FA ratio by S.

Upon every timeout, decrease the FA ratio by

$$\frac{C(\text{timeout})}{C(\text{false fast retransmit})} S$$

Upon every limited-transmit-induced idle period, provided $C(\text{limited transmit}) > C(\text{false fast retransmit})$, decrease the FA ratio by

$$\frac{C(\text{limited transmit})}{C(\text{false fast retransmit})} S$$

Modifications in Simulator to Implement Algorithm

In the ns-3 simulator, a number of modifications were made in the hope of implementing the TCP-RR algorithm. The details are as follows:

src/internet/model/tcp-rr.h

It is similar to the header part of TCP New Reno of tcp-congestion-op.h. This class inherits the TCP New Reno class.

src/internet/model/tcp-rr.cc

It is similar to the code part of TCP New Reno of tcp-congestion-op.cc with trivial modification.

src/internet/model/tcp-socket-state.h

Variable:

- **SequenceNumber32 m_cumulativeAck:** The cumulative ack sequence
- **uint32_t m_reorderingLength:** The reordering length of a packet

src/internet/model/tcp-socket-base.h

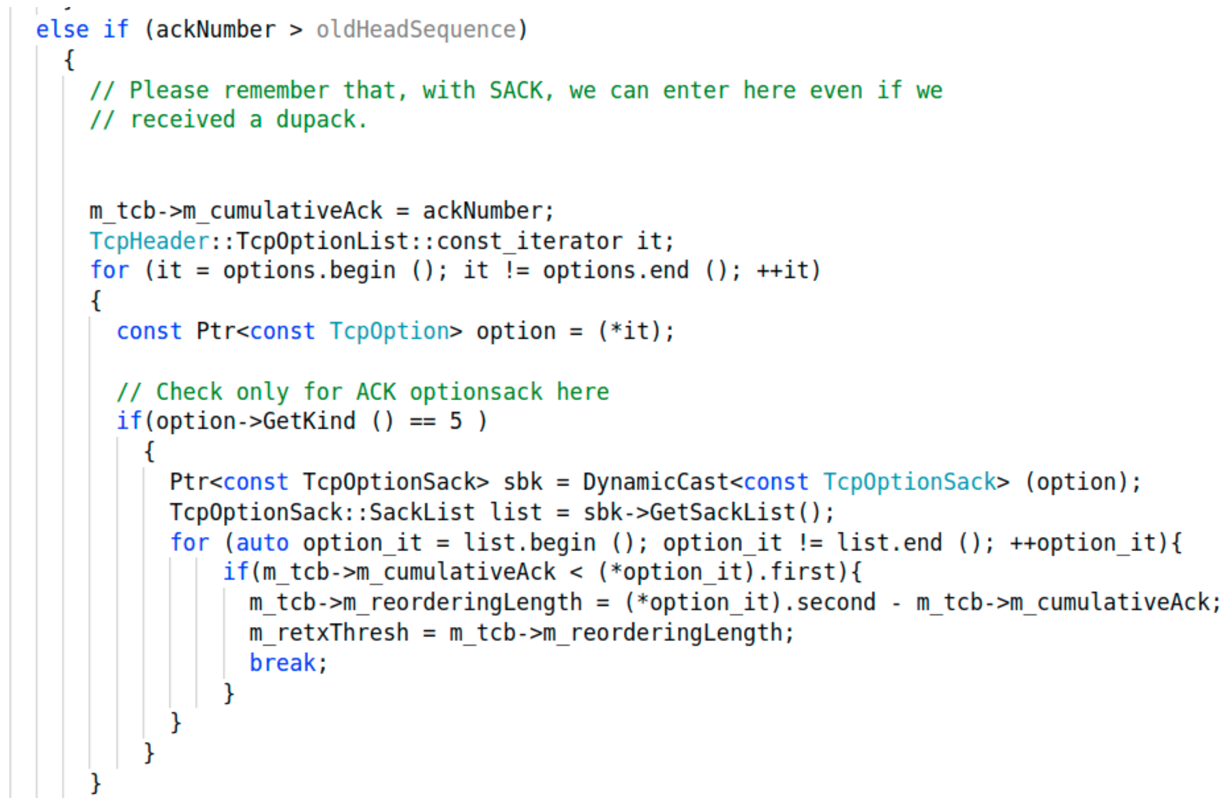
Functions:

- **Void TcpSocketBase::ProcessAck():** The 1st parameter of this function is changed to const TcpHeader &tcpHeader from const SequenceNumber32 &ackNumber. This declaration of the function is updated in this file.

src/internet/model/tcp-socket-base.cc

Functions:

- **Void TcpSocketBase::ProcessAck():** The 1st parameter of this function is changed to const TcpHeader &tcpHeader from const SequenceNumber32 &ackNumber. Also, to store the value of cumulative Ack and reordering length and to set the value of dupthresh the following part was added in this function when the sequence number of the received acknowledgement is greater than the first unacknowledged sequence number in the transmission buffer.



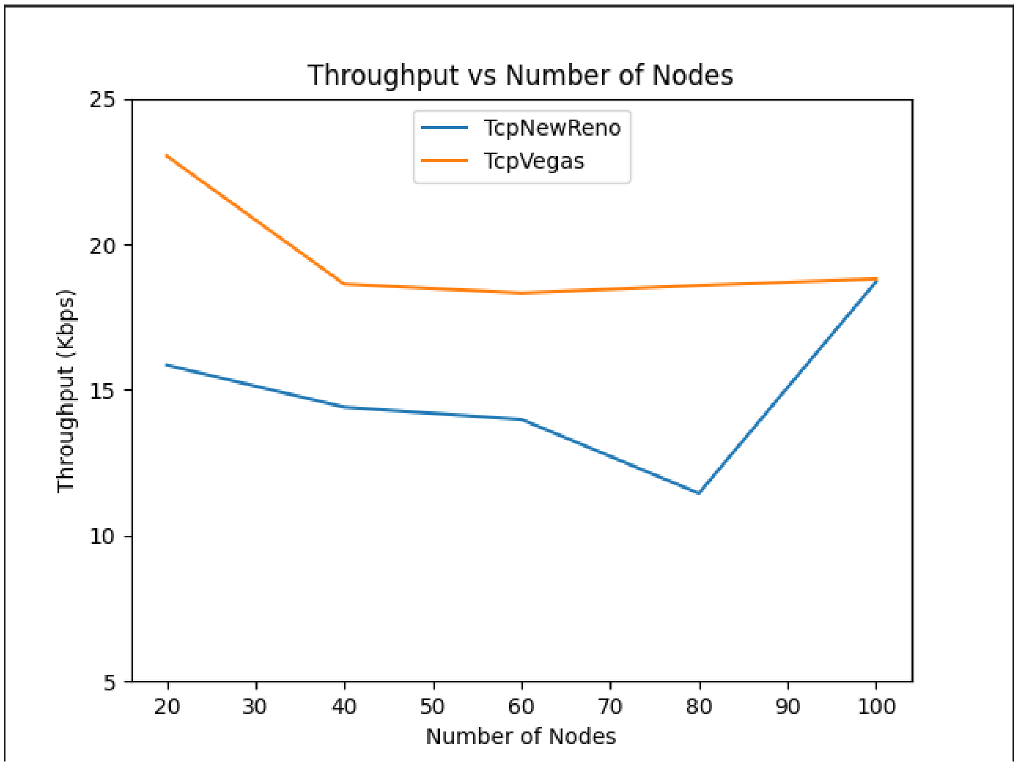
```
else if (ackNumber > oldHeadSequence)
  {
    // Please remember that, with SACK, we can enter here even if we
    // received a dupack.


    m_tcb->m_cumulativeAck = ackNumber;
    TcpHeader::TcpOptionList::const_iterator it;
    for (it = options.begin (); it != options.end (); ++it)
    {
      const Ptr<const TcpOption> option = (*it);

      // Check only for ACK optionsack here
      if(option->GetKind () == 5 )
        {
          Ptr<const TcpOptionSack> sbk = DynamicCast<const TcpOptionSack> (option);
          TcpOptionSack::SackList list = sbk->GetSackList();
          for (auto option_it = list.begin (); option_it != list.end (); ++option_it){
              if(m_tcb->m_cumulativeAck < (*option_it).first){
                m_tcb->m_reorderingLength = (*option_it).second - m_tcb->m_cumulativeAck;
                m_retxThresh = m_tcb->m_reorderingLength;
                break;
              }
          }
        }
    }
```

Results with Graph

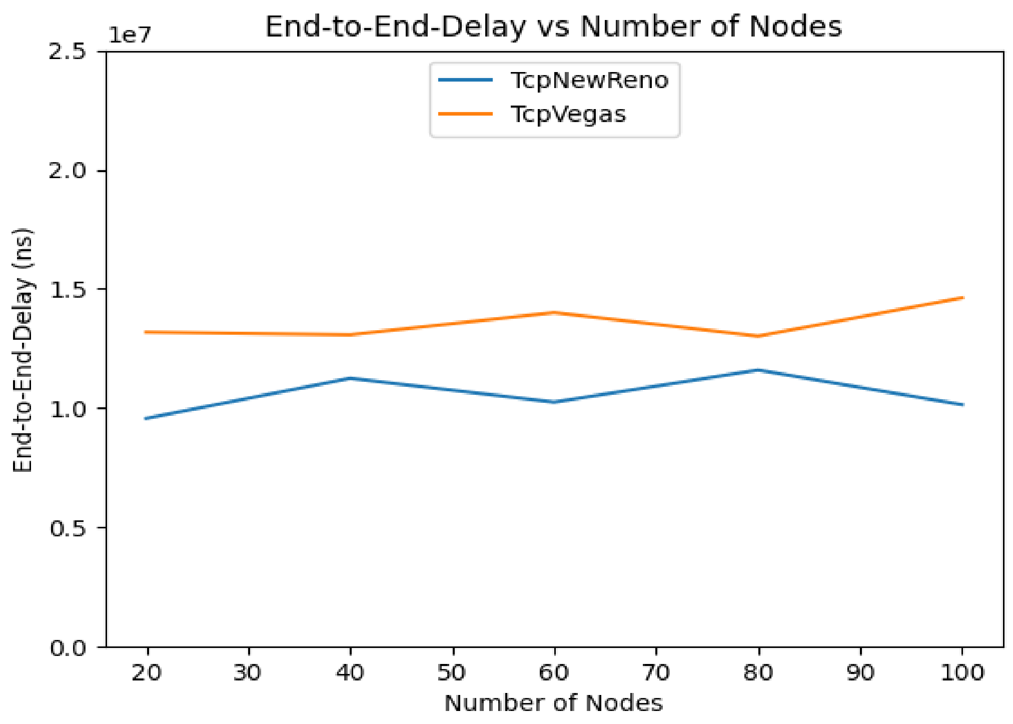
Task A (Wired):

- **Varying Number of Nodes (20, 40, 60, 80, 100):**As the distance between the source node and the sink node increases due to the increase of the number of nodes, the throughput decreases in both TCP New Reno and TCP Vegas. But the throughput again increases for TCP New Reno when the number of nodes is 100. The packet delivery ratio was changed a little throughout the variation of the number of nodes. The packet drop ratio was decreased.

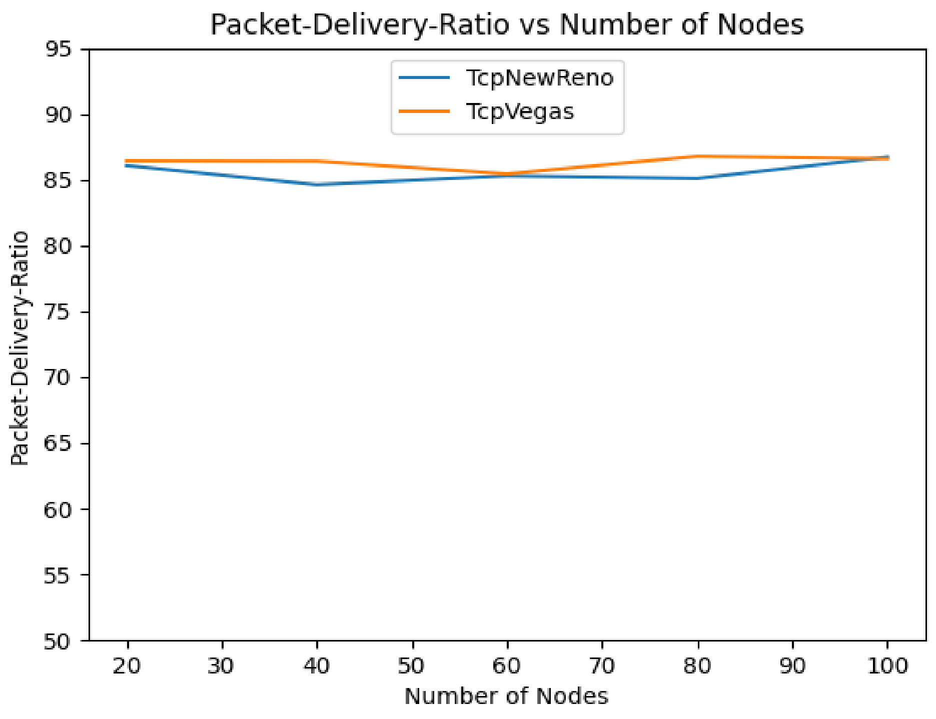
Throughput:



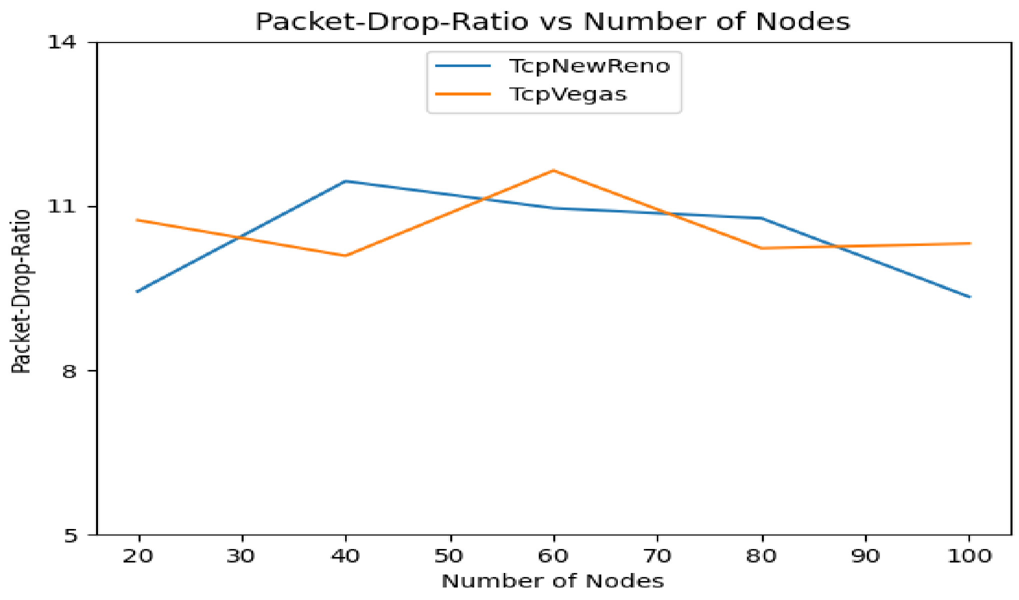
End to end delay:



Packet delivery Ratio:

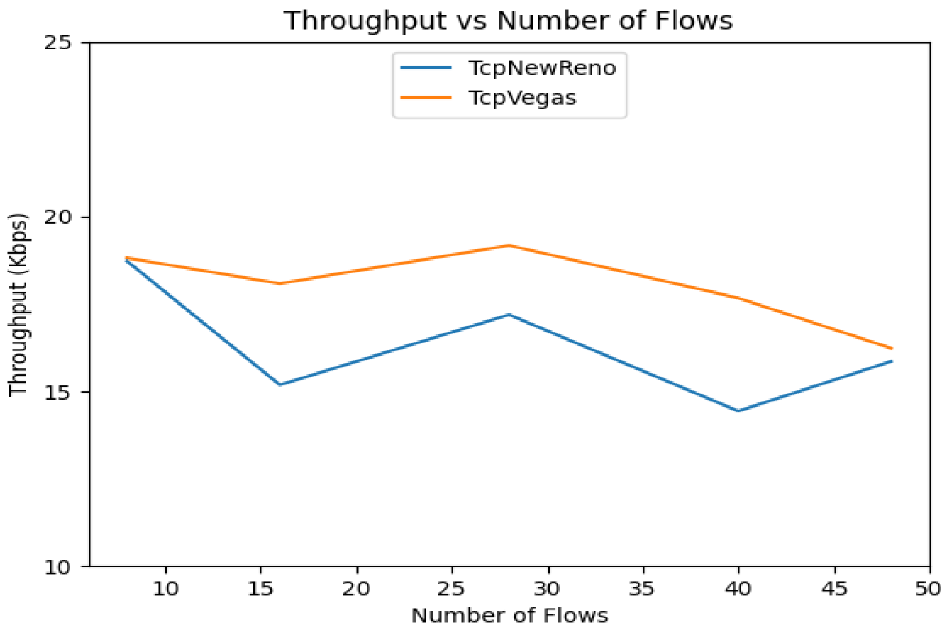


Packet drop Ratio:

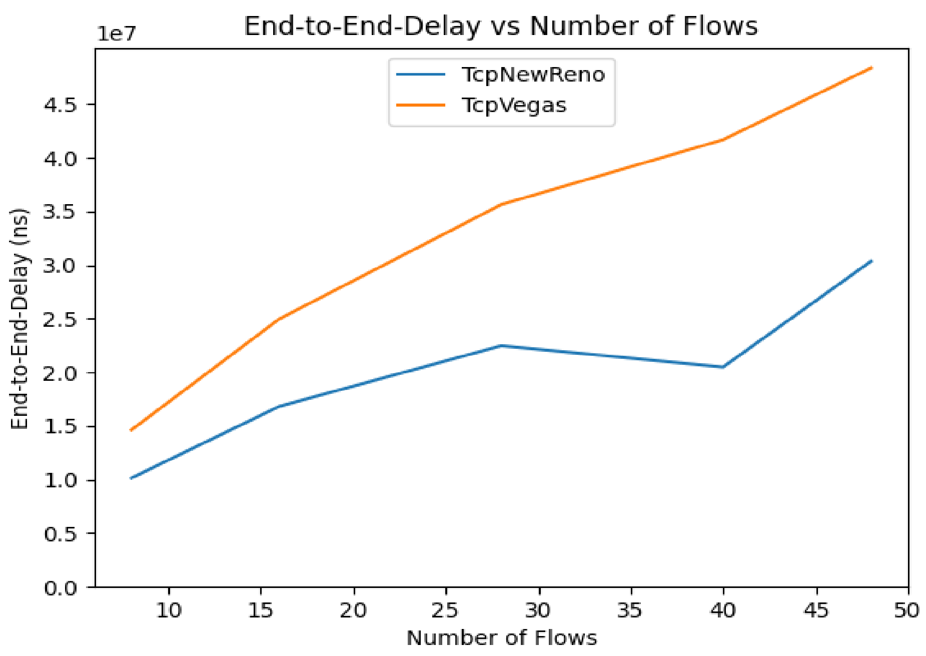


- **Varying Number of Flows (8, 16, 28, 40, 48):** As the number of flows increases in the network, the throughput decreases in both TCP New Reno and TCP Vegas due to more traffic in the path. The packet delivery ratio was changed a little throughout the variation of the number of flows. The packet drop ratio was increased.

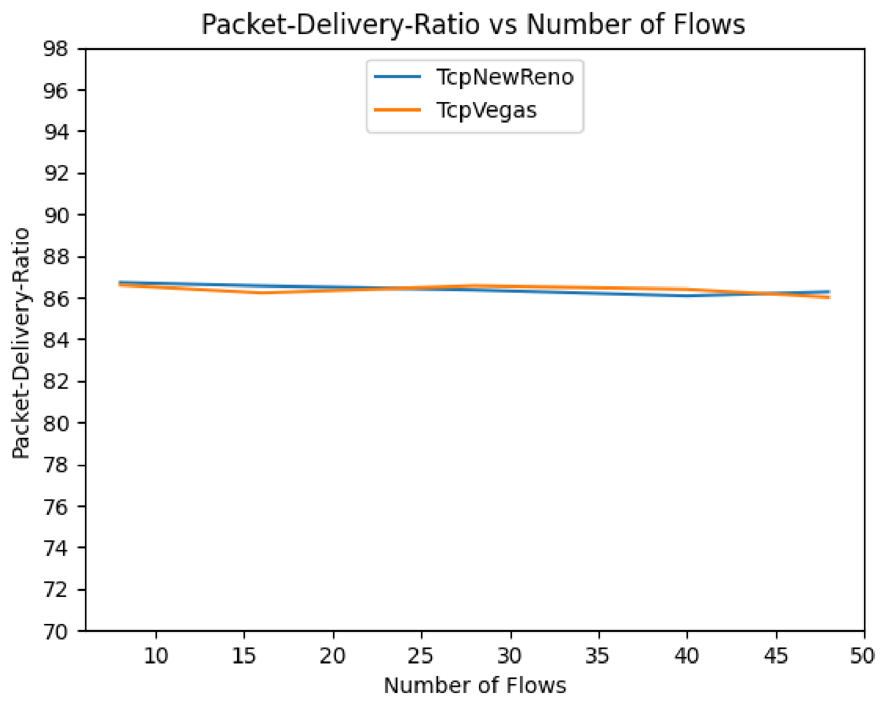
Throughput:



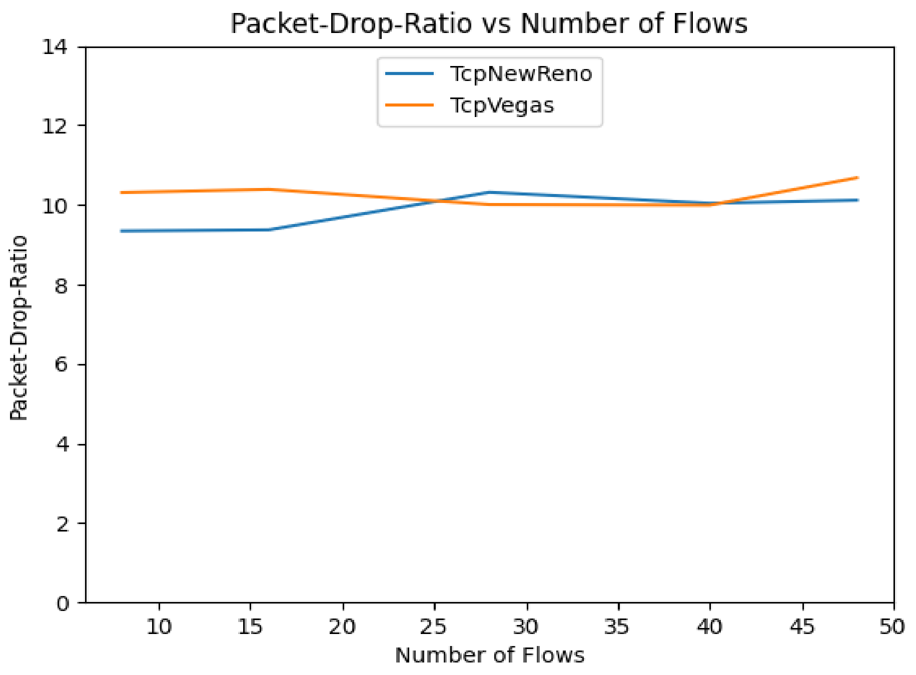
End to end delay:



Packet delivery Ratio:

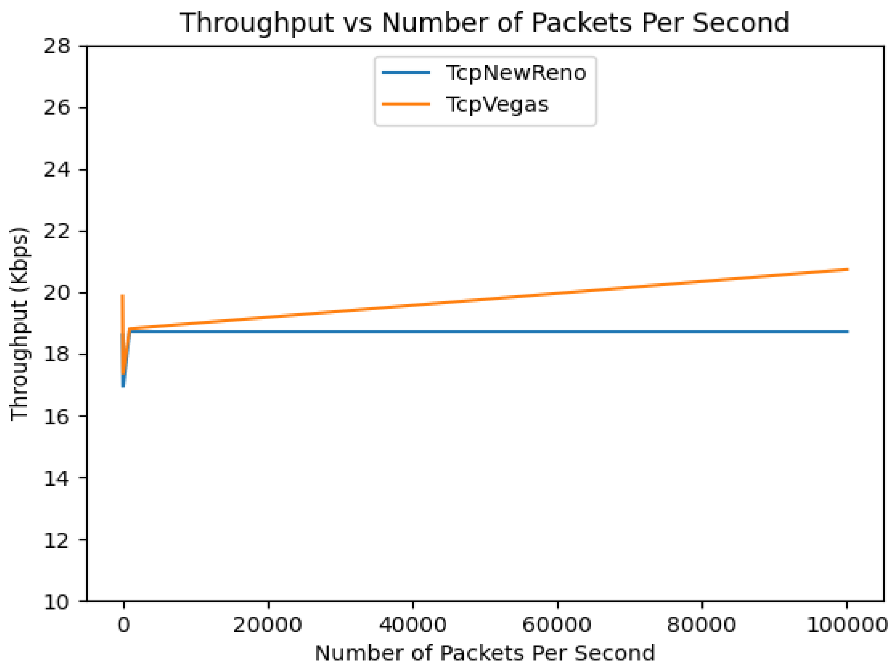


Packet drop Ratio:

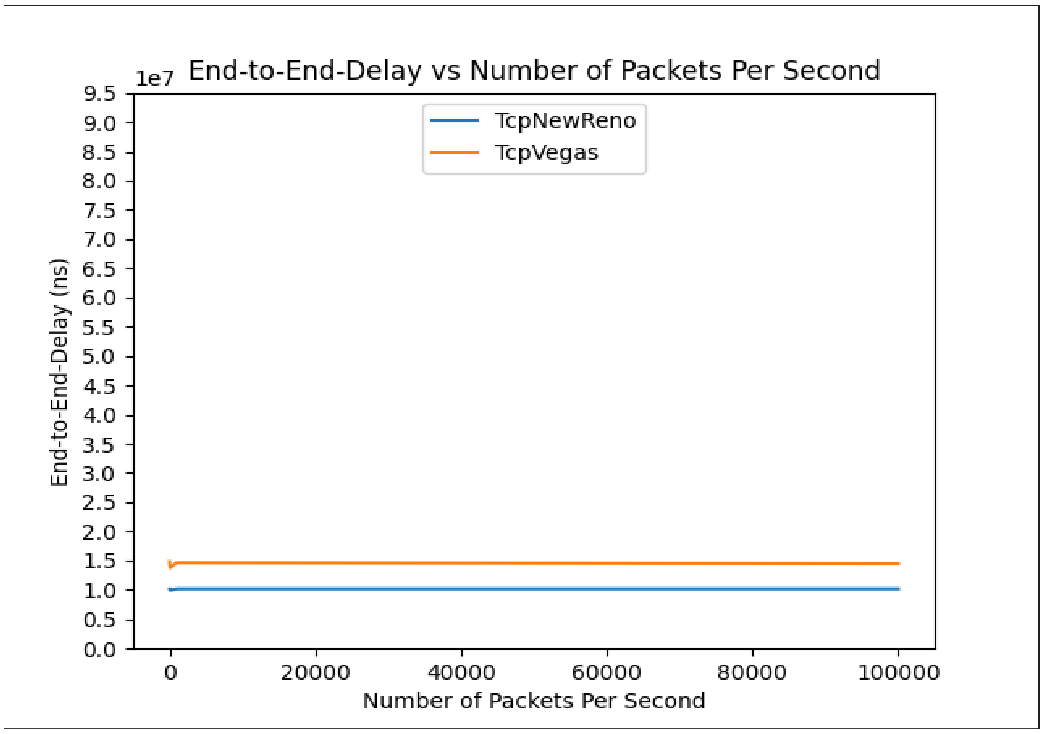


- **Varying Number of Packets Per Seconds (10, 100, 1000, 100000):**
  All the metrics barely changed by varying the number of packets per second.

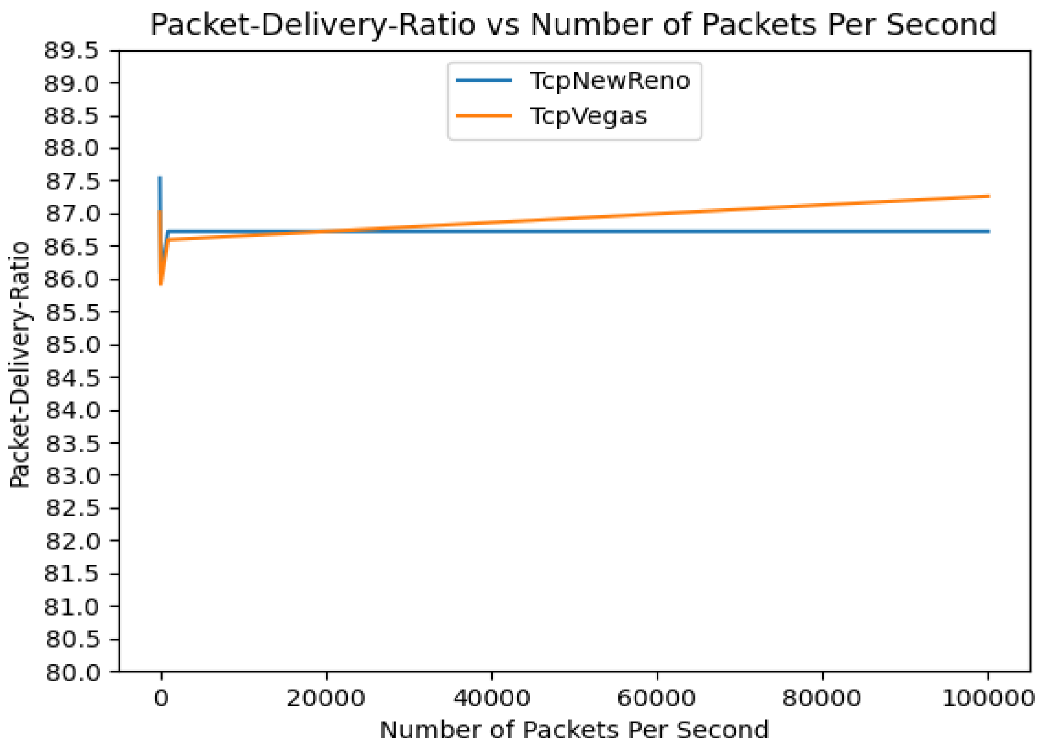
Throughput:



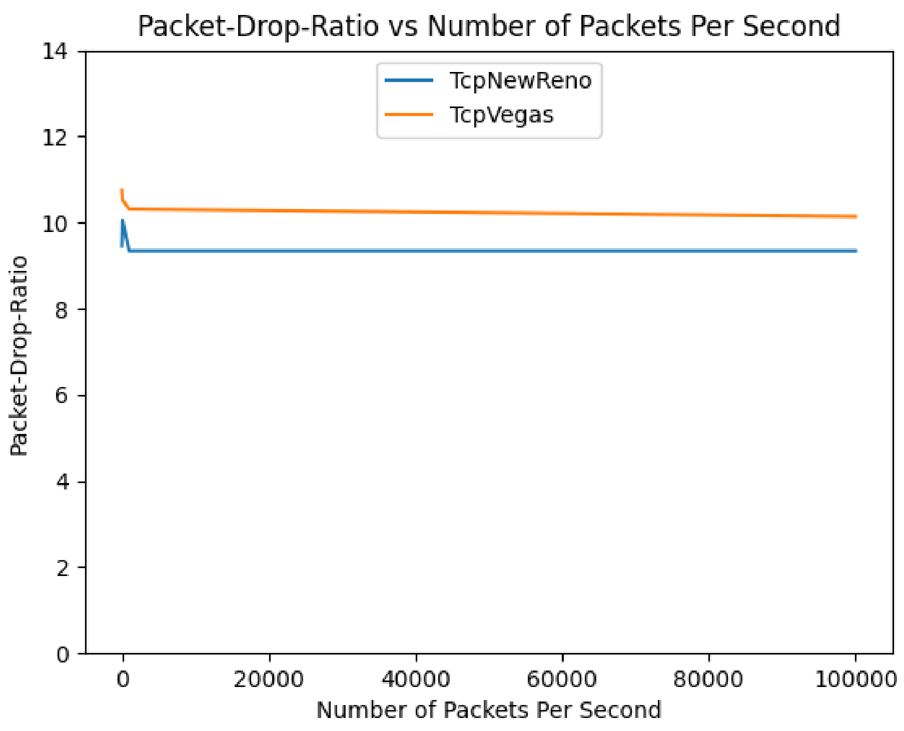
End to end delay:



Packet delivery ratio:



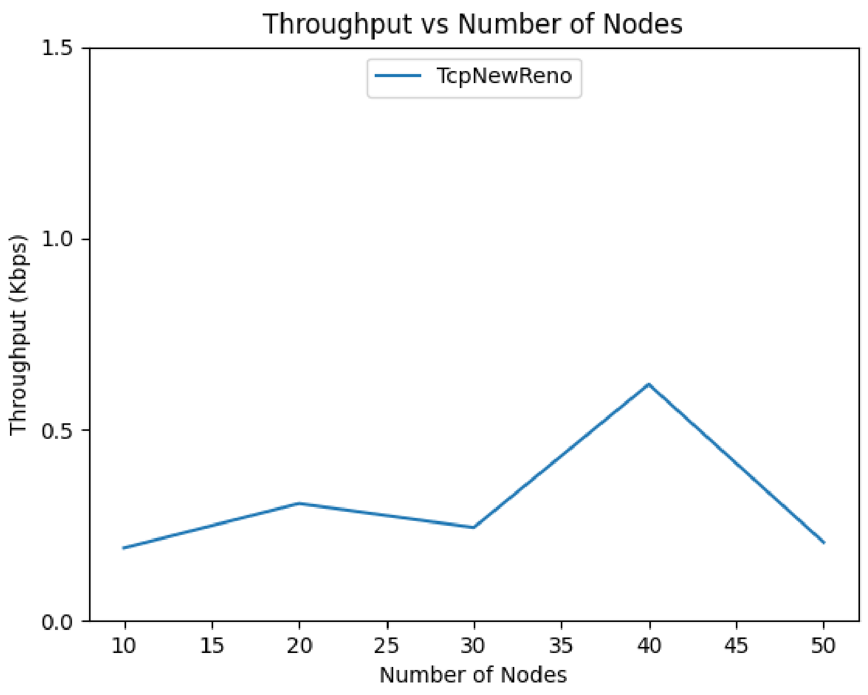
Packet drop ratio:



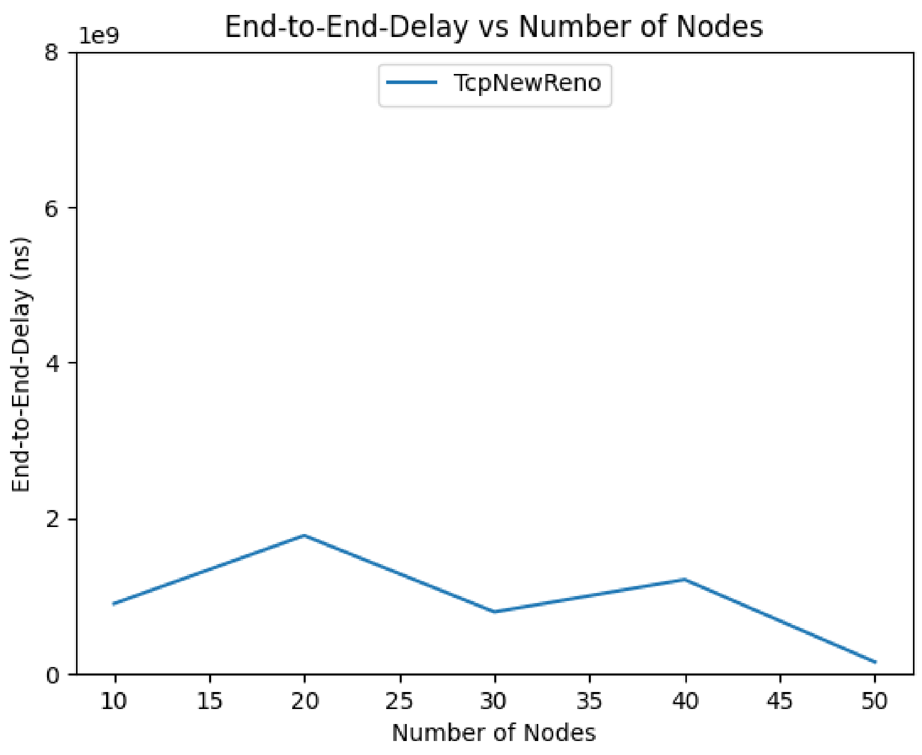
Task A (Wireless low rate (802.15.4) (static)):

- **Varying Number of Nodes (10, 20, 30, 40, 50):**

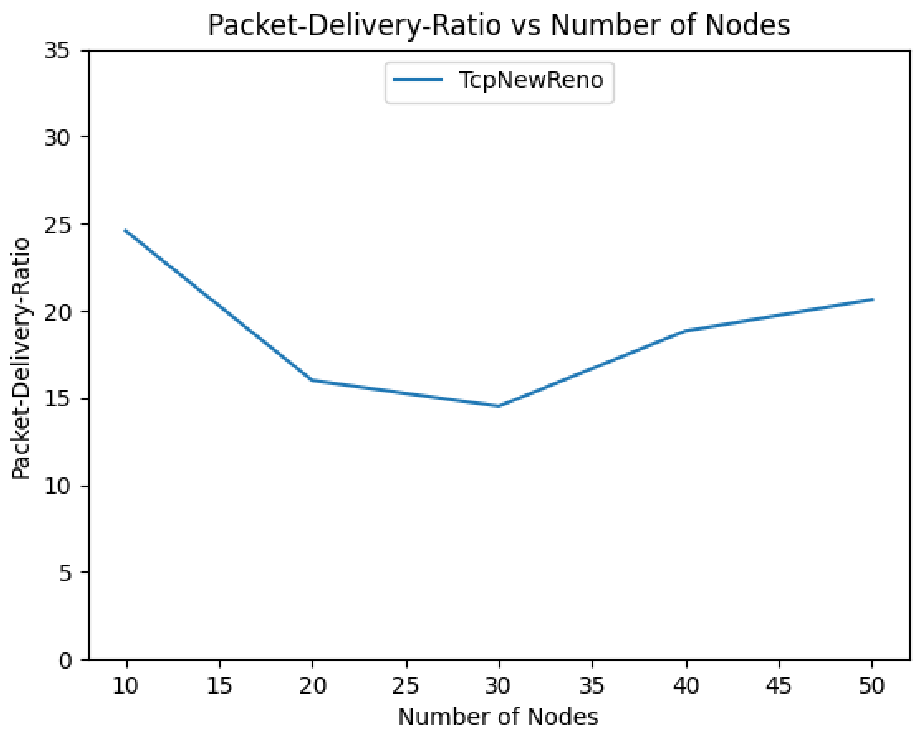
Throughput:



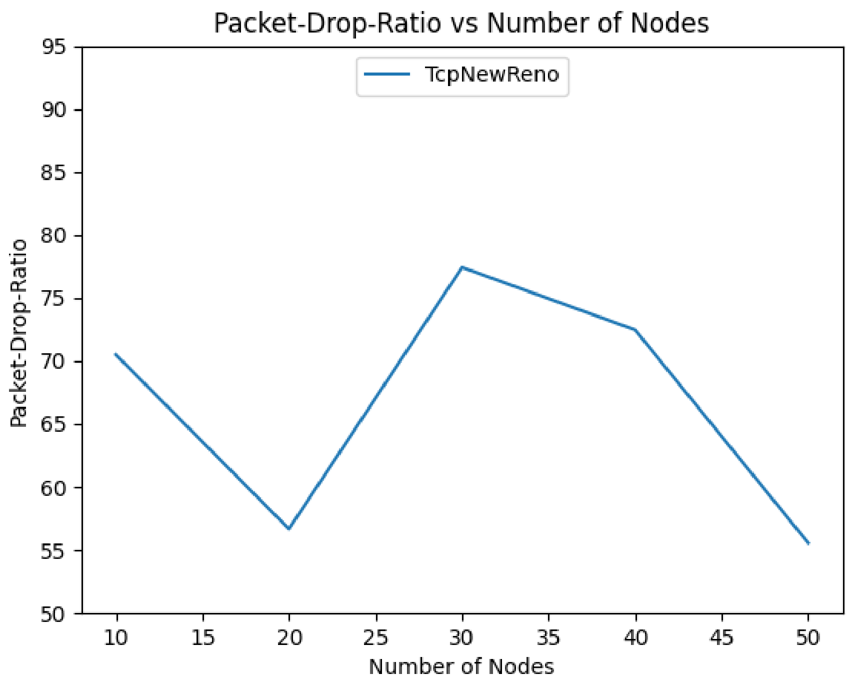
End to end delay:



Packet delivery ratio:

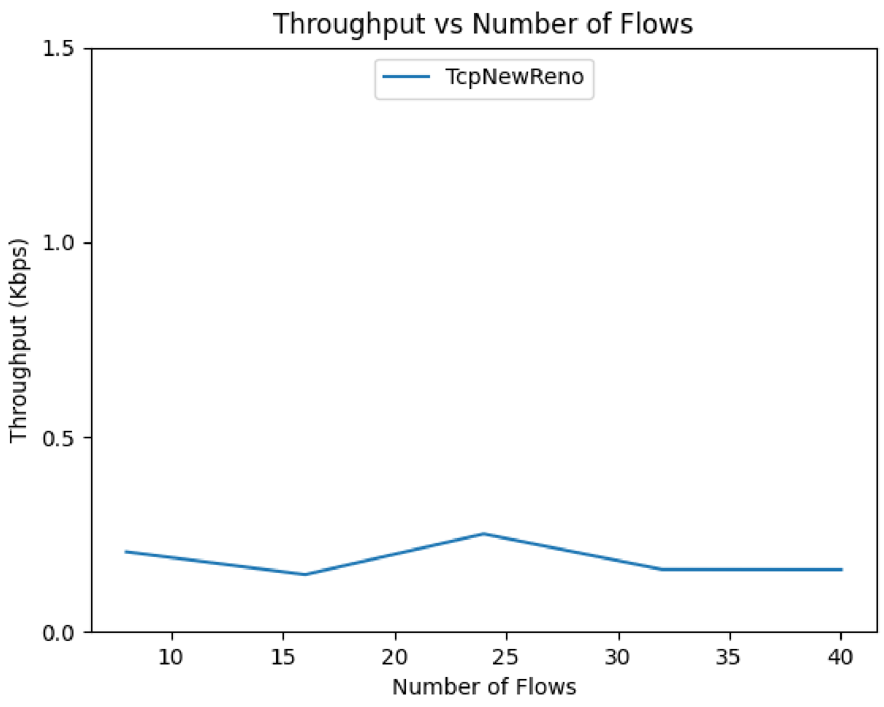


Packet drop ratio:

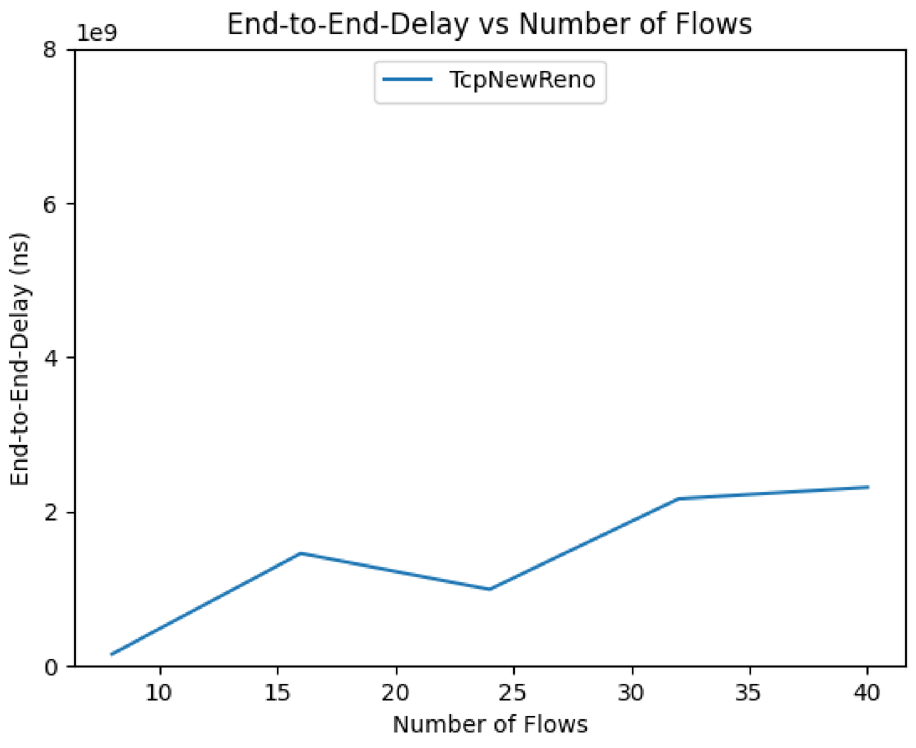


- **Varying Number of Flows (8, 16, 24, 23, 40):** The end to end delay increased with the increase in flows. The packet drop ratio also increased. As there is more traffic in the path to the sink which is in a LAN connection and the router has to pass all the packets of all flows, there is more drop in packets.

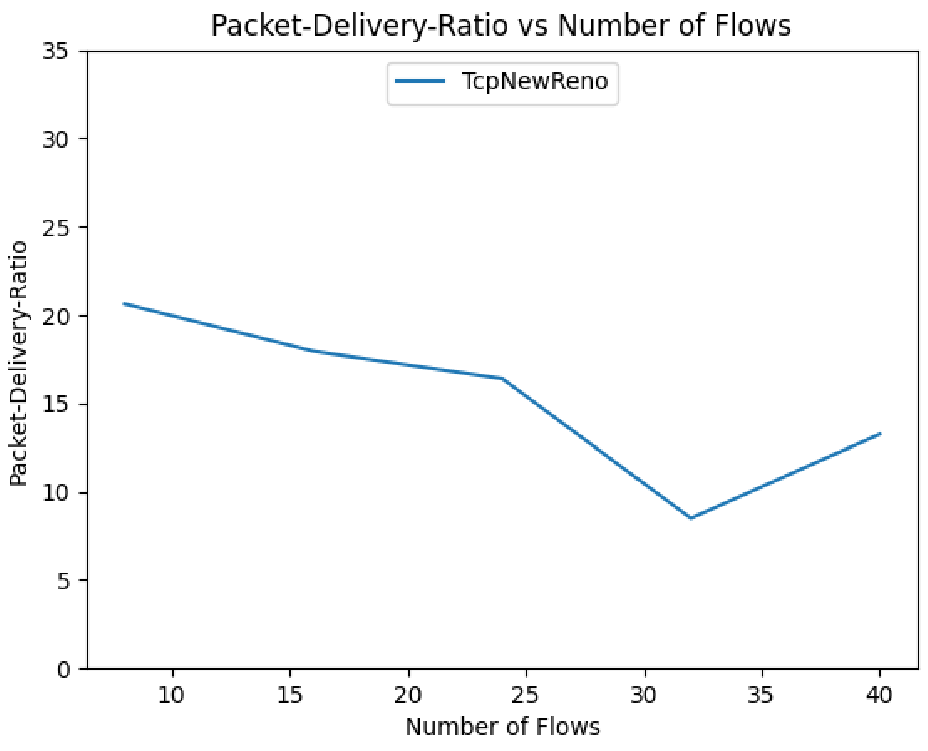
Throughput:



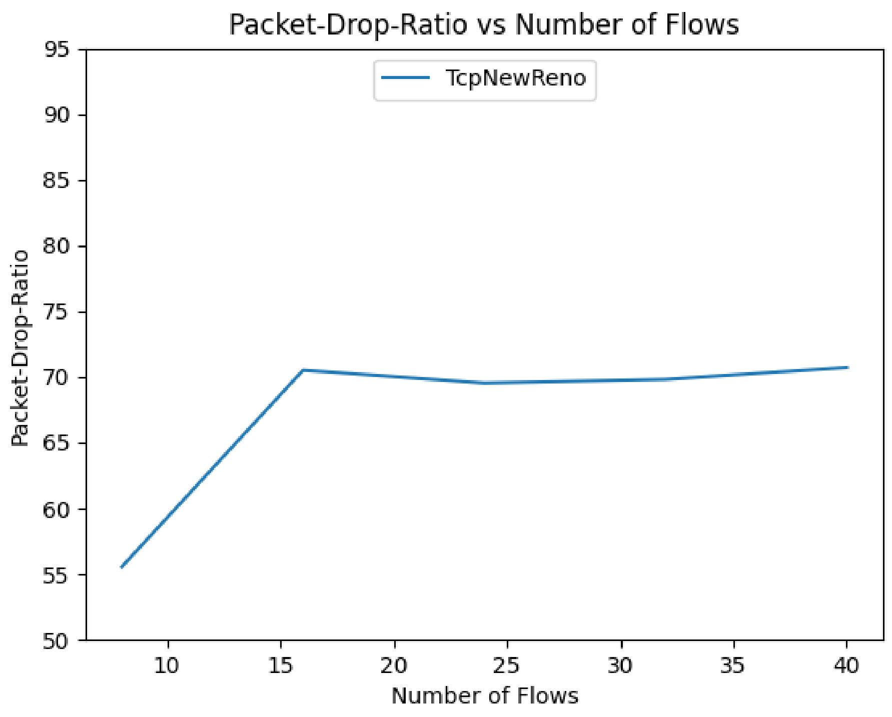
End to end delay:



Packet delivery ratio:

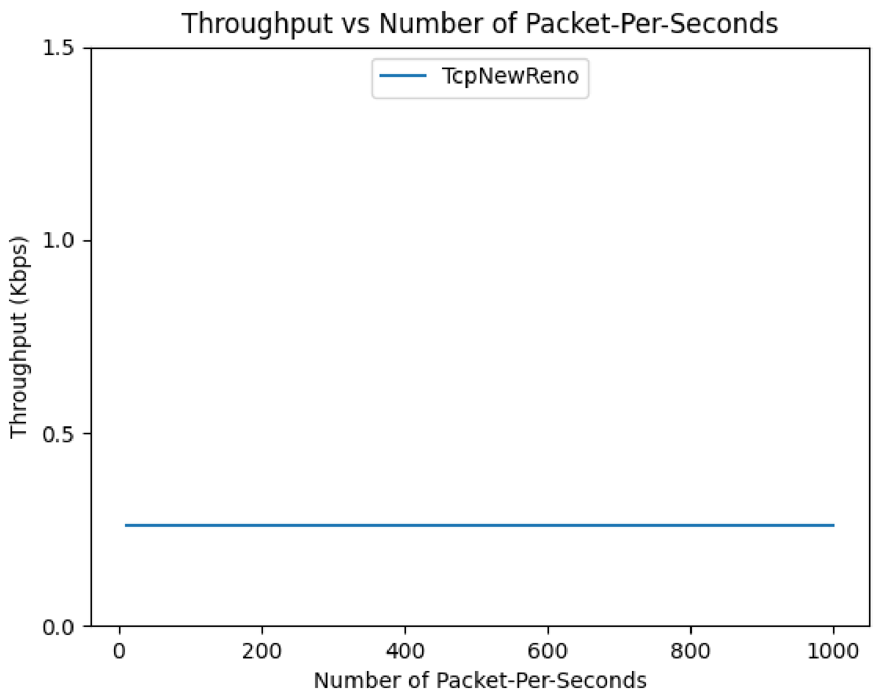


Packet drop ratio:

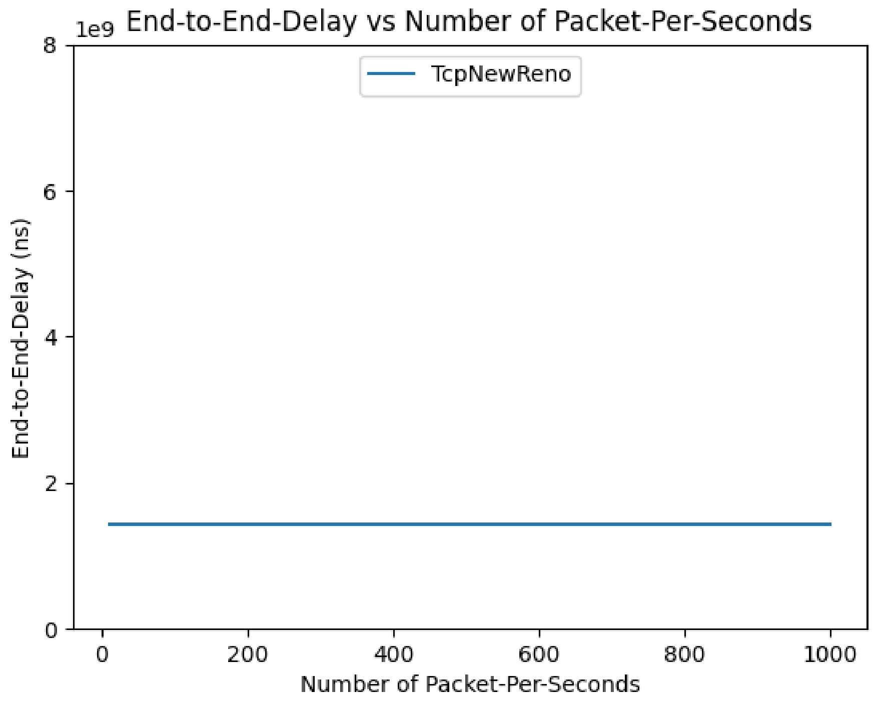


- **Varying Number of Packets per Second (10, 100, 200, 500, 1000):**All the metrics barely changed by varying the number of packets per second.

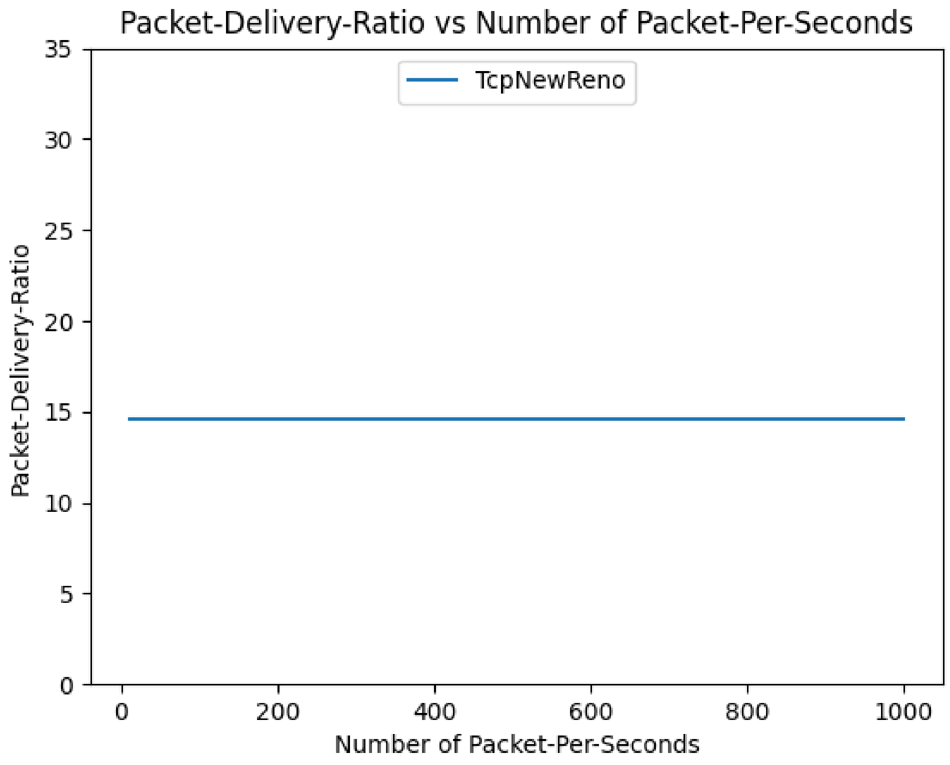
Throughput:



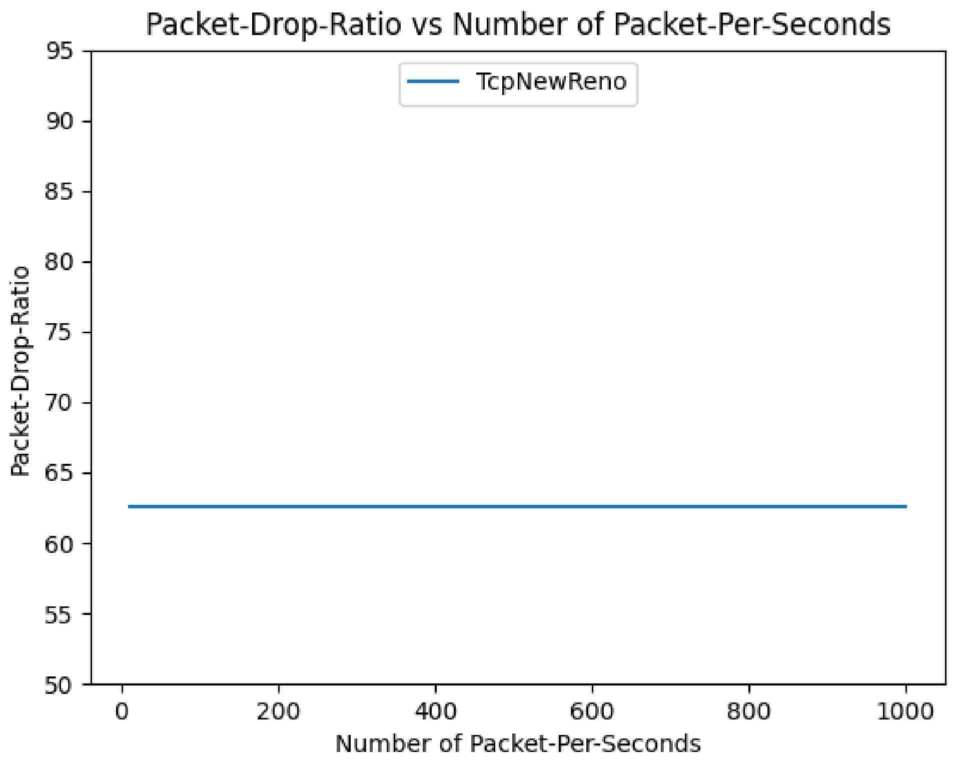
End to end delay:



Packet delivery ratio:

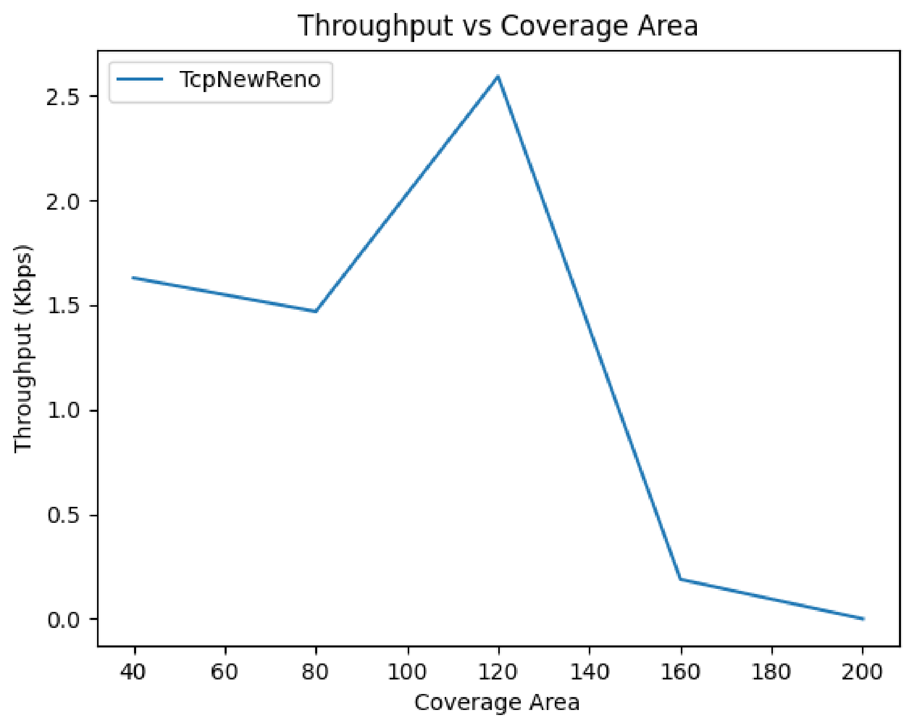


Packet drop ratio:

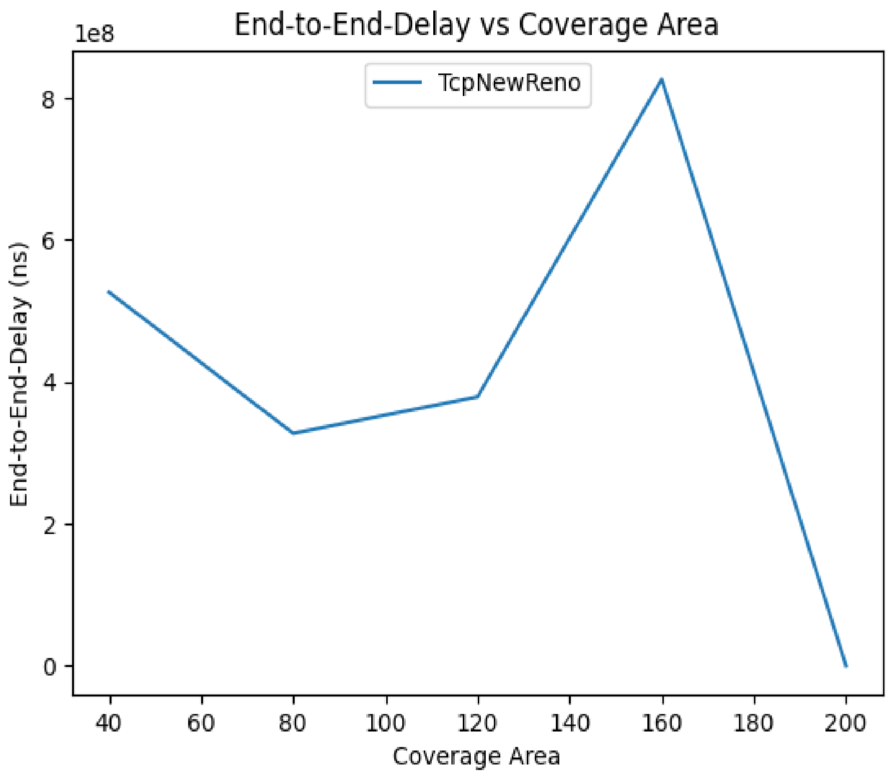


- **Varying Coverage area (40, 80, 120, 160, 200):** Here the coverage area can be denoted as coverageX = DeltaX*gridwidth. The txrange was set to 40 and gridwidth was set to 4. By varying the DeltaX from 10 to 50, coverage area varies. When the DeltaX is greater than 40, with txrange as 40, any node cannot send any packet to any node. That is why, the delivery ratio decreases after 160 and throughput also decreases. The end to end delay was increasing because of the increase in DeltaX. After 160, it was 0 as no packet was transmitted.

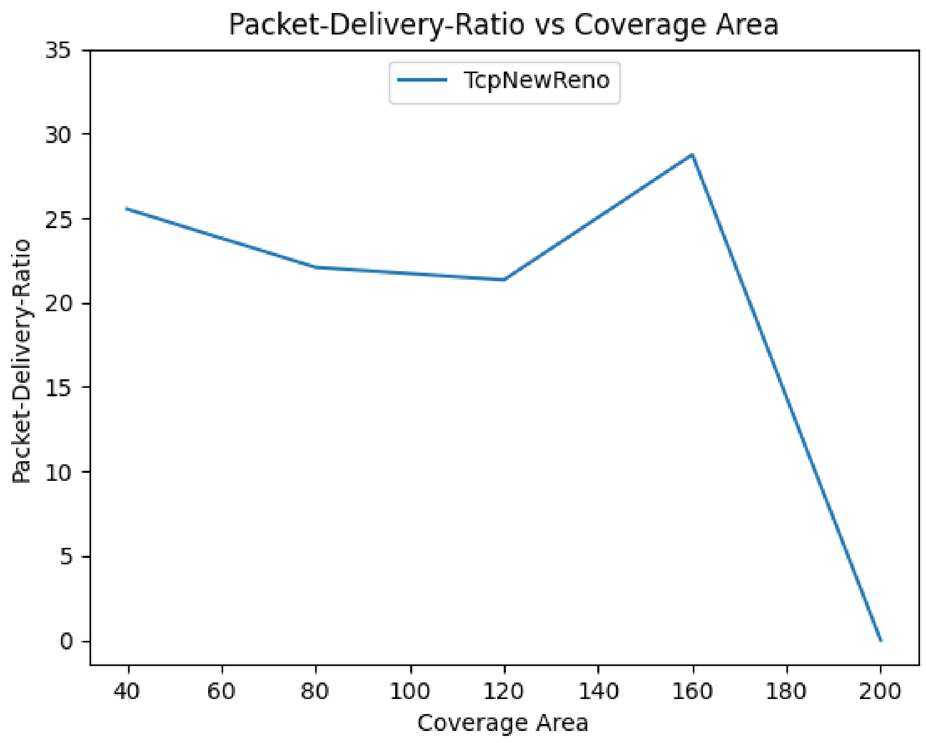
Throughput:



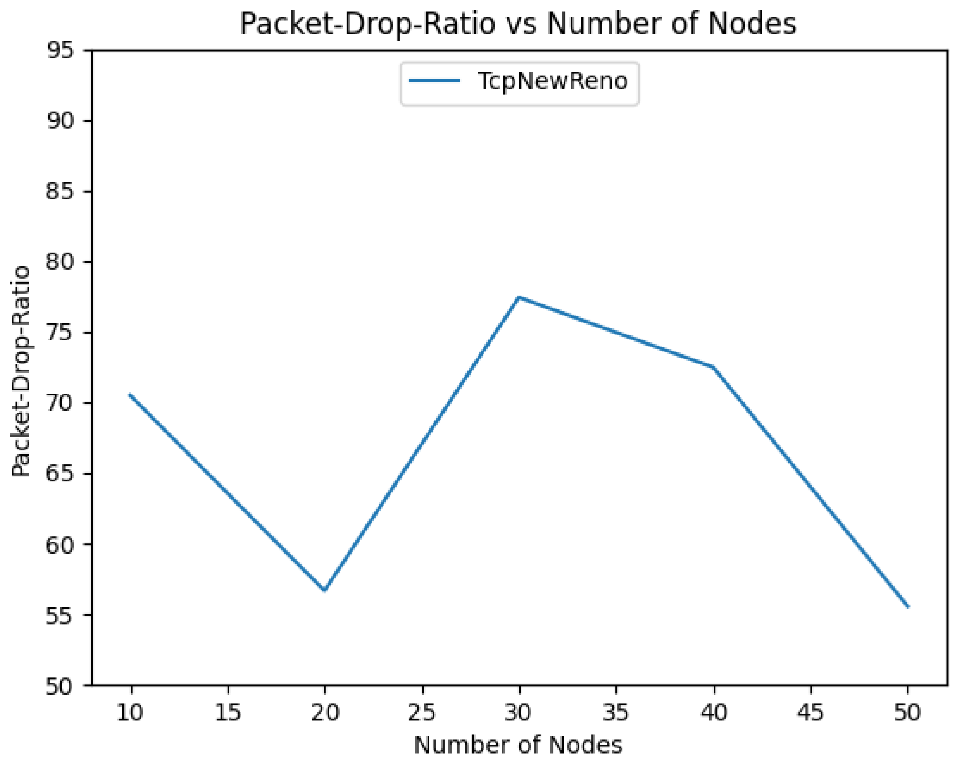
End to end delay:



Packet delivery ratio:



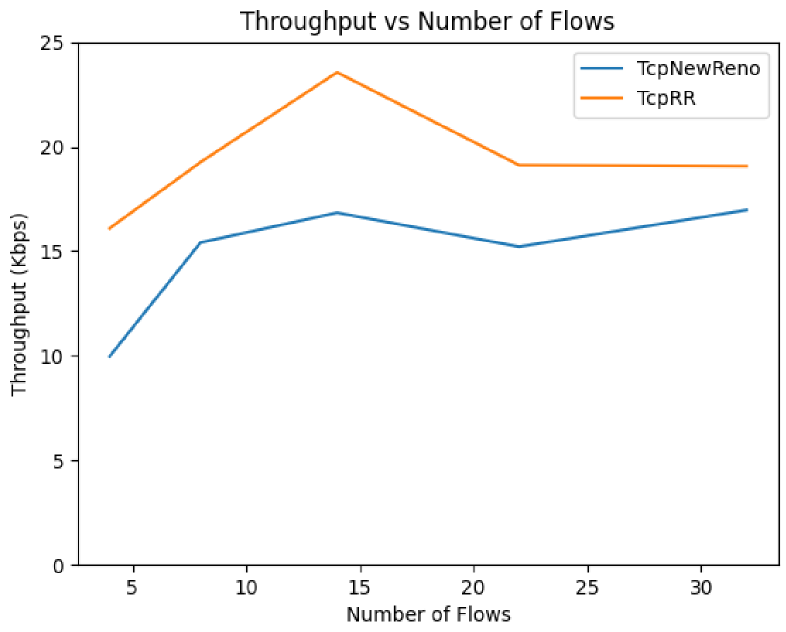
Packet drop ratio:



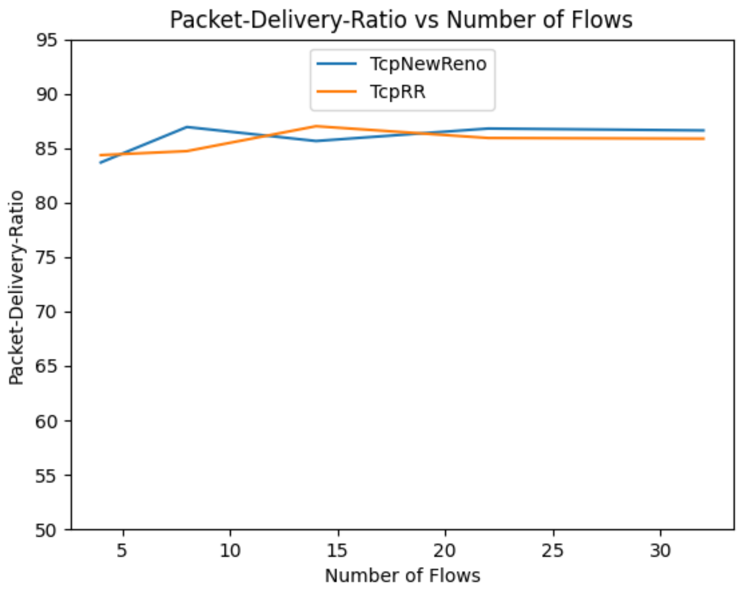
Task B (Wired):

- **Varying Number of Flows (4, 8, 14, 22, 32):** As it is seen that we achieved better throughput in TcpRR than Tcp New Reno as the dupthresh is taking various values in TcpRR rather than having a fixed value 3 like Tcp New Reno. So every time there is a reordering in received packets, the sender is not retransmitting after 3 duplicate Acknowledgements. So the congestion window is not being reduced more often. The packet delivery ratio is similar in both cases.

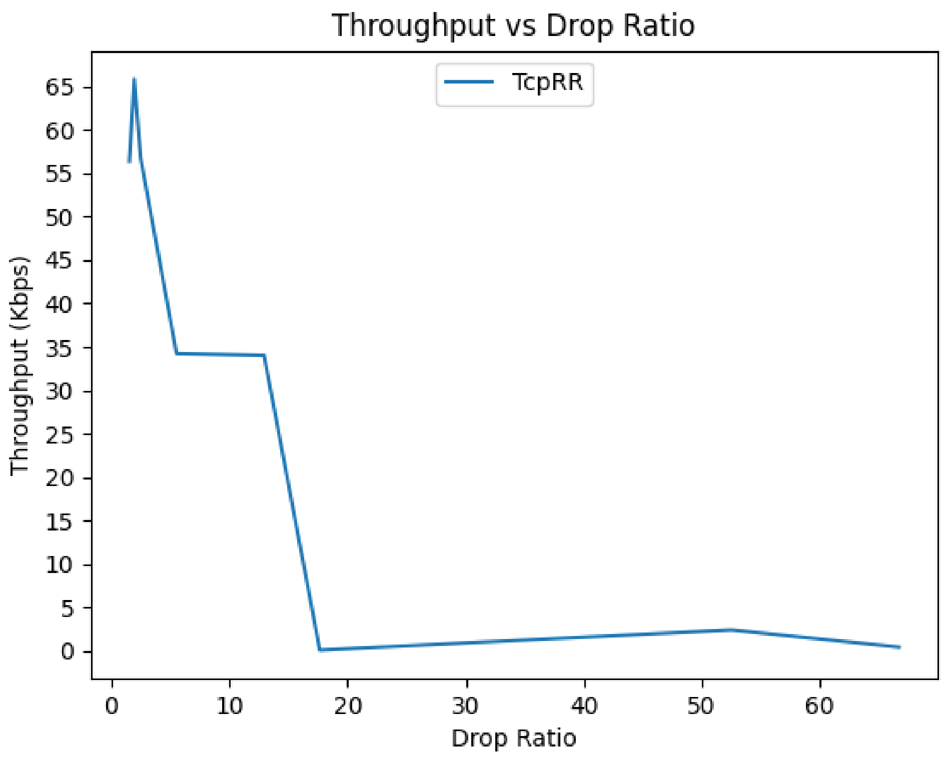
Throughput:

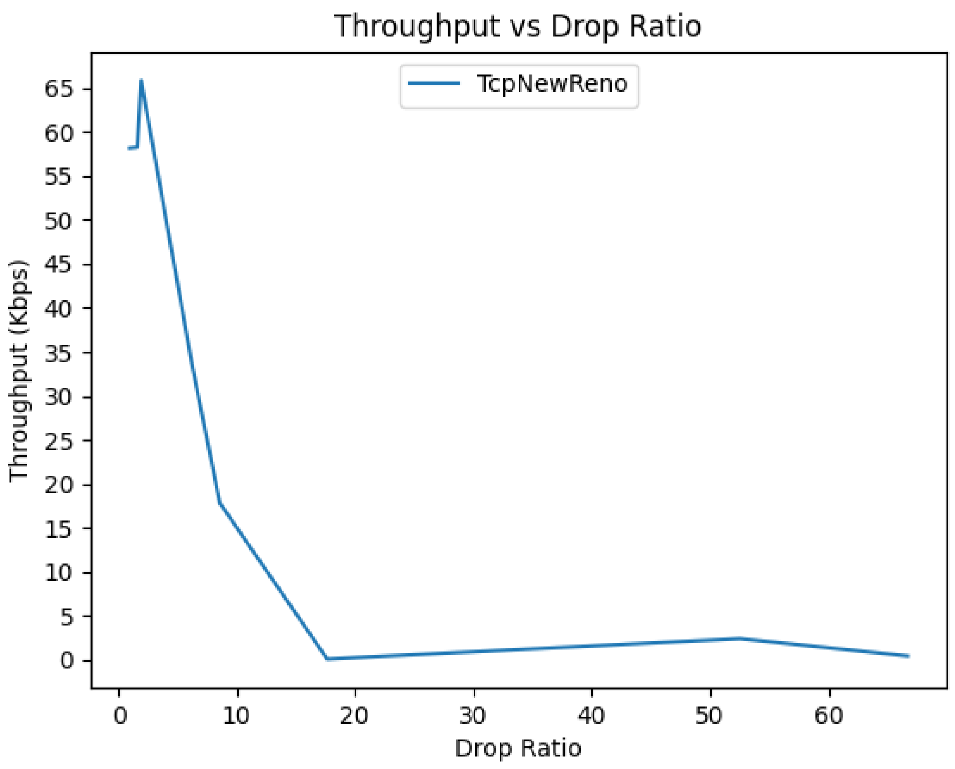


Packet delivery ratio:



Throughput VS Packet drop ratio:


Throughput vs Drop Ratio
TcpRR
Throughput (Kbps)
Drop Ratio


Throughput vs Drop Ratio
TcpNewReno
Throughput (Kbps)
Drop Ratio