CS 143: Final Report

Rafael Fueyo-Gomez   Cortland Perry   Kelsi Riley   Sakthi Vetrivel

December 12, 2018

1 Introduction

For our term project, our team decided to pursue the network simulator. At a high level, the simulator takes in a input of a the description of network, such as its hosts, routers, links, and flows, runs a simulation for a user-specified duration, and outputs graph demonstrating the progression of certain metrics, such as flow rate, window size, or buffer occupancy, after the end of the run.

This simulator is written in Python3 and makes use of the matplotlib package. Code can be found at `https://github.com/rafugo/cs143_simulator`. Instructions to run the test cases can be found in the repository's README file.

2 Simulator Architecture

2.1 Simulator Class

Our simulator class contains all of the network objects needed to simulate our network.

- **__init__(self, string filename)** This function uses the input file specified in filename to create network objects based on a network description.
- **plot_metrics(self)** This function uses all of the metrics that were recorded during simulation and plots them. The graphs are then outputted as png files in the directory of the code.
- **run(self)** This function runs the actual simulator. Upon starting the simulator, we first have all of router make their handshakes to learn the routing table for the network. Then, we run all network events on a timestep basis. So, for a user-specified amount of time, we run the simulator for so many timesteps. For every timestep, we make calls to link class to try to send all packets in the buffer, and calls to every flow to send packets in the flow. We then increment our global clock. Moreover, every five seconds, we send all of our link states throughout the router to update the link states and the routing table if needed.

2.2 Host Class

Our host class simulates the behavior of a host in our network.

- **__init__(self, int hostid, int linkid)** Initializes a host object. The hostid allows us to identify against different host objects and the linkid allows to identify which link is associated with this host.
- **send_packet(self, Packet p)** Send the packet p by adding (or attempting to add) the packet to the link buffer.

- **receive_packet(self, Packet p, int linkid)** Sends acknowledgements of packets received and notifies the correct flow of these acknowledgements.

2.3 Router Class

- **__init__(self, int id, List¡Link¿ links)** Initializes a router object. The id allows us to identify different routers, and the list of links refers to the links that are connected to this router object.
- **receive_packet(self, Packet packet, int linkid)** Processes packets and has case handling for the different packets it might receive such as handshake packets, handshake acknowledgements, standard packets or routing packets.
- **forward_packet(self, Packet packet)** Forwards packets based on the routing table and their destination.
- **send_handshake(self)** Send the initial handshake packet to the adjacent routers to determine which nodes are connected.
- **recalc_link_state(self)** Recalculates the link states.
- **send_link_state(self)** Send out our link state array to neighbors.
- **receive_handshake_ack(self, Packet packet, int linkid)** Processes handshake acknowledgement.
- **receive_link_state(self, string array state_array_actual)** Upon receiving a link state, the router updates its own link state array.
- **run_dijkstra(self)** Run's Dijkstra's algorithm to determine the routing table.

2.4 Packet Class

- **__init__(self, int sourceid, int flowid, int destinationid, int packetid, string packet_type, string data = ”)** Initialize a packet object. sourceid referes to the id of the source (whether it's a router or a host). flowid refers to the id of the flow the packet belongs to. destinationid referes to the id of the destination (whether it's a router or a host). packetid is the identification of the packet within the flow and is its number in the sequence. packet_type is the type of the packet and can take on the values of STANDARDPACKET: a normal packet, ACKPACKET: an acknowledgement packet, ROUTINGPACKET: a routing table packet, SYNPACKET: a synchronization packet, SYNACK: a synchronization packet acknowledgement.

2.5 Link Class

- **__init__(self, int linkid, int connection1, int connection2, float rate, float delay, int buffersize, bool track1=** Initializes a link object for our network. linkid is the ID for this link object. connection1 and connection2 are the id's of the hosts/routers at the ends of the link. rate and delay are floats that refer to the rate and propagation delay of the link. buffersize is the size of the buffer for this link. track1 and track2 are boolean values to determine if we should track metrics for this link.
- **add_to_buffer(self, Packet packet, int sender)** Add packet to buffer based on the sender of the packet.
- **send_packet(self)** Tries to send a packet on both of the half links corresponding to the link.
- **HalfLink** Represents one direction of the Link. (i.e. all packets travelling across a given HalfLink will be going to the same destination) Class has its own definitions for add_to_buffer and send_packet.

3 Congestion Control

Our congestion control takes place in our flow class. For our project, we've implemented two congestion control algorithms, TCP Reno and TCP Fast. To do this, we have two different Flow classes (flow_reno.py and flow_fast.py), with different implementations of the same public methods. Here's an overview of all of the public methods.

- **run(self)** This function is called at every timestep by the simulator for the flow to send any packets as necessary for that timestep.
- **process_ack(self, Packet p)** Process an acknowledgement once it's received by a host. Checks for duplicate acknowledgements and changes congestion control states, window size, and threshold accordingly.
- **send_packets(self)** Send the packets for the flow depending on the current system time and the acknowledgements that have been received thus far.

3.1 TCP Reno

In our TCP Reno implementation, we have three possible states for the algorithm to be in: slow start, congestion avoidance, or fast recovery. Dependent on these states, we have different behavior. If we are in the slow start state of our algorithm, for every acknowledgement that is not a duplicate acknowledgement received, we increase our window size by one. If we are in the congestion avoidance state, we increase our window size by $\frac{1}{cwnd}$ where $cwnd$ is the current window size. If we are in fast recovery, we do not send any packets until the retransmission of the dropped packet is successful. We then half our window size and enter congestion avoidance.

To start, our window size is initialized at 1 packet. Our window start, which refers to the first packet in the next window of consecutive packets to be sent, starts at 0. The congestion control algorithm starts in the slow start state with a slow start threshold of 1000. We chose this initial value to be arbitrarily large, so the network will either timeout or receive three duplicate acknowledgements before it enters congestion avoidance for the first time. Before we start sending any packets from our flow, we send a synchronization packet. We then wait for this packet to return and be acknowledged to calculate our initial RTT value.

To send packets for our flow, at every timestep, we see if there are any packets in our current window that have not yet been sent, and if there are, we send them. To keep track of duplicate acknowledgements, we have a counter for consecutive duplicate acknowledgements. Once we reach three, we enter fast retransmit and fast recovery.

Morever, to calculate the round trip time for our packets, we use Karn's algorithm, which states that the round trip time calculation is only based on packets that are sent and acknowledged only once. All other packets are ignored in the metric. To keep track of round trip times, we have a dictionary send_times as an attribute of the flow_reno class. This maps packet IDs to the time at which they were last attempted to be sent. This dictionary only includes packets that have been sent but not yet acknowledged. Likewise, we have a dictionary dup_count that maps packet ids to the number of times they have been sent. So if a packet has a dup_count larger than 1, then we do not consider it in the calculation of the round trip time. Thus, every time we process an acknowledgment, we check the dup_count of the packet that is acknowledged, and then access its send_time to calculate the latest RTT.

For packet timeouts, we use a timeout marker. This timeout marker is based on the last non-duplicate acknowledgement received and an RTO value. This RTO value is calculated as $2 \times RTT$. Thus, every time an acknowledgement is processed, the timeout marker is reset to be the current system time + RTO. Then, at every time step, we check if we have passed this timeout marker, meaning a packet has timed out – specifically the first packet that we have sent but have not received an acknowledgement for. Upon recognizing a packet timeout, we set our slow start threshold to half of the current window size and set our window size to 1 and enter the slow start state. We also retransmit the lost packet and then double the RTO for future timeouts. Additionally, to avoid entering a loop where packets timeout repeatedly, resulting in overpenalizing the network for dropping packets, we keep track of a next_cut_time. This variable refers to the next time that cut our threshold in half and ensures that we wait 1 RTO before cutting the threshold again.

3.2 TCP FAST

In our TCP FAST implementation, we also have three possible states for the algorithm to be in: slow start, congestion avoidance, or fast recovery. The way TCP FAST works, it is not so much dependent on ACKs to modify its state, but moreso on the RTT.

In the slow start state, we increment the window size by one for every ACK, similar to in Reno. However, we limit this behavior to every other RTT. That is, we have an active RTT and a frozen RTT. During the "active" RTT, we increment the window size and during the "frozen" one, we do not do anything. Our implementation then exits slow start and enters congestion avoidance when it thinks that it is approaching the estimated throughput. It also may exit slow start when there are three duplicate ACKs, and then enters fast recovery.

To calculate when we may be nearing our estimated throughput, we do a simple estimate. We keep track of how many packets we receive during each frozen RTT (in reality, just the current and the previous frozen RTTs). Thus, we have the given throughput of an RTT as $\frac{m_{current}}{RTT}$ with $m_{current}$ being the amount of packets received in the current frozen RTT. Note that, since we are doubling the window size every other RTT, we should expect

$$\frac{2 * m_{previous}}{RTT_{previous}} = \frac{m_{current}}{RTT_{current}} \Rightarrow \frac{2 * m_{previous} * RTT_{current}}{RTT_{previous}} = m_{current}$$

If we receive 90% less than the predicted $m_{current}$, then we believe to be increasing the buffer size in the bottleneck link along the path, so we go into congestion avoidance in order to avoid dropping packets. We chose 90% in order to be conservative and not drop any packets by being too aggressive.

We enter fast recovery in the same way we would enter it in Reno, and it is handled exactly the same way. Duplicate ACKs are also handled the same as in Reno.

Once we enter congestion avoidance, we continue with the alternating active and frozen RTTs. Every time we enter a new active RTT, we set a goal window size that we want our window size to be by the end of the active RTT. This goal window size is whatever the "next window size" is at that given moment. The next window size is updated every 20 ms, according to the following equation:

$$w \leftarrow \min(2w, (1-\gamma)w + \gamma(\frac{\text{baseRTT}}{\text{RTT}}w + \alpha))$$

where w on the right hand side is the current window and the w on the left hand side is the next window size. We chose $\alpha = 15$ and $\gamma = 0.5$.

For our RTTs, we used the same algorithm as in Reno. We also followed the same timeout procedure.

4 Metrics Tracking

5 Results

5.1 Test Case 0

5.1.1 TCP Reno

5.1.2 TCP Fast

5.2 Test Case 1

5.2.1 TCP Reno

5.2.2 TCP Fast

5.3 Test Case 2

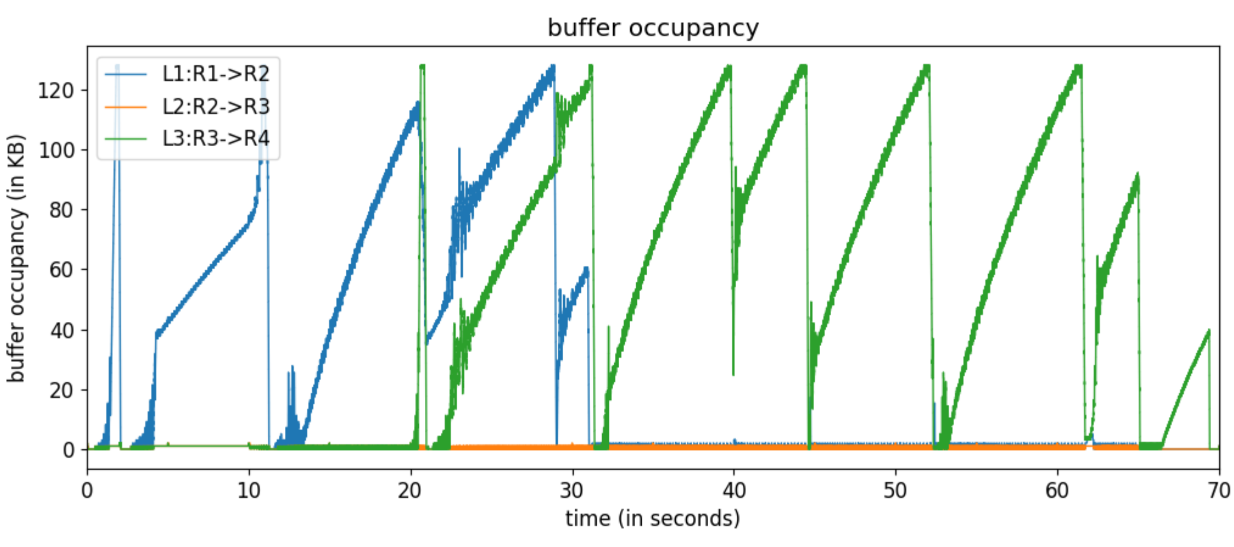
5.3.1 TCP Reno

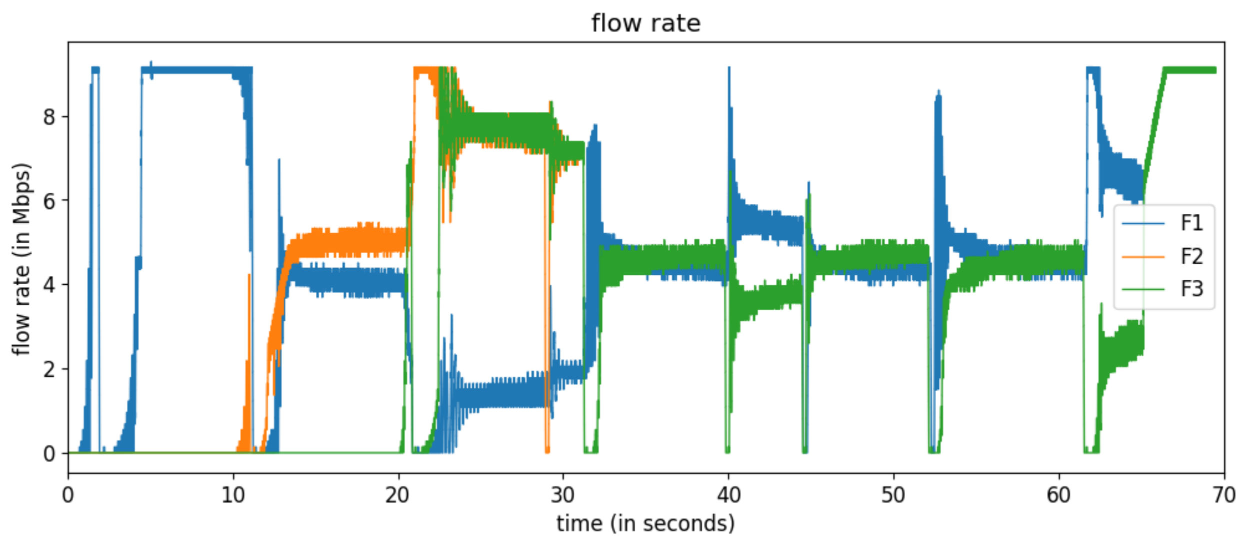
5.3.2 TCP Fast

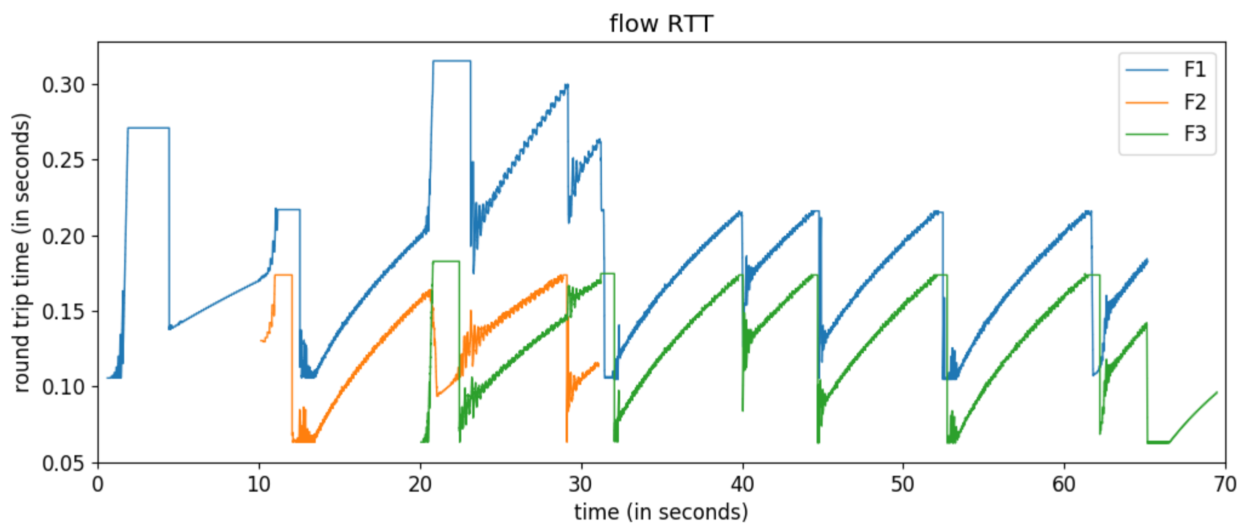
6 Analysis

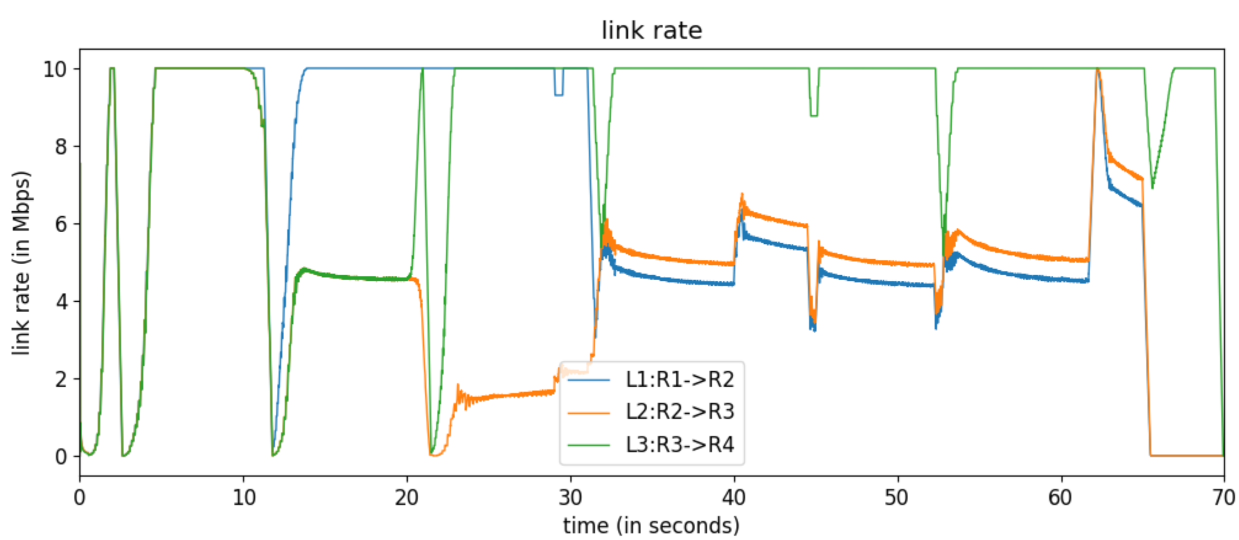
6.1 Test Case 2

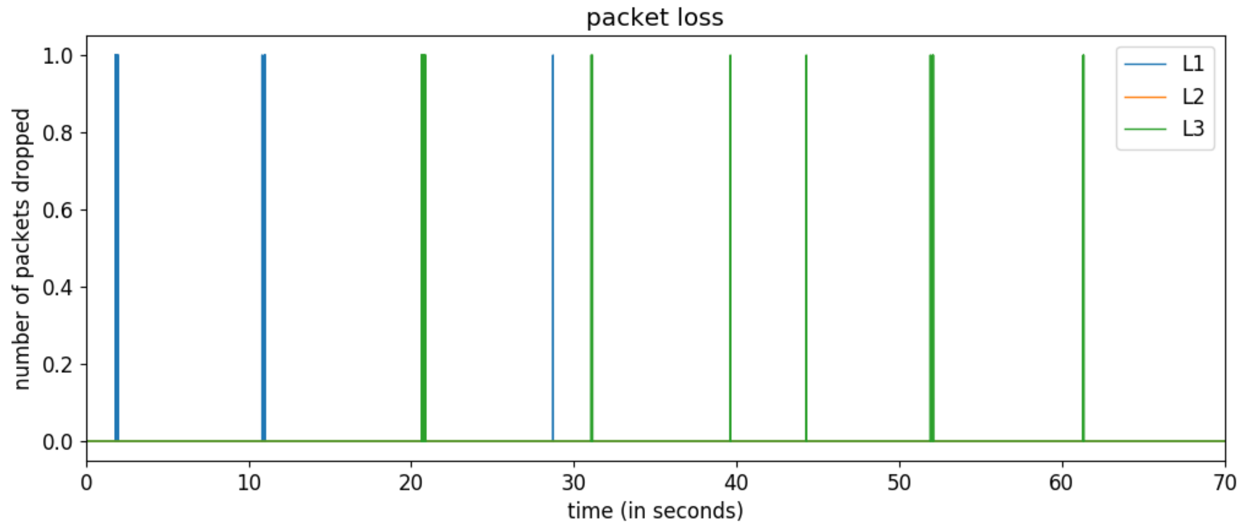
6.1.1 TCP Reno

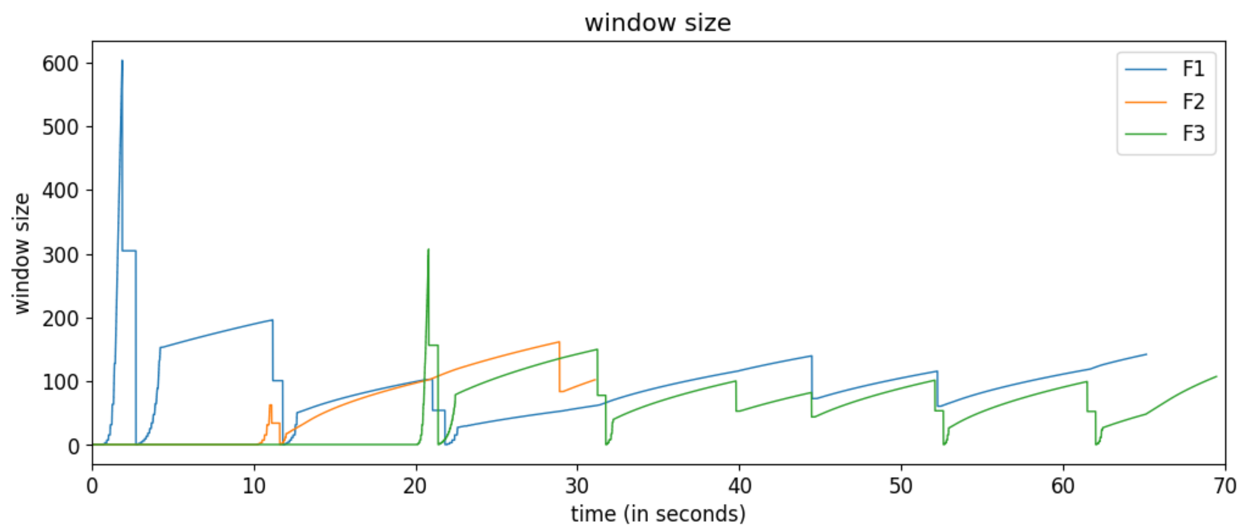



flow rate
flow rate (in Mbps)
time (in seconds)
F1
F2
F3
0
2
4
6
8
0
10
20
30
40
50
60
70

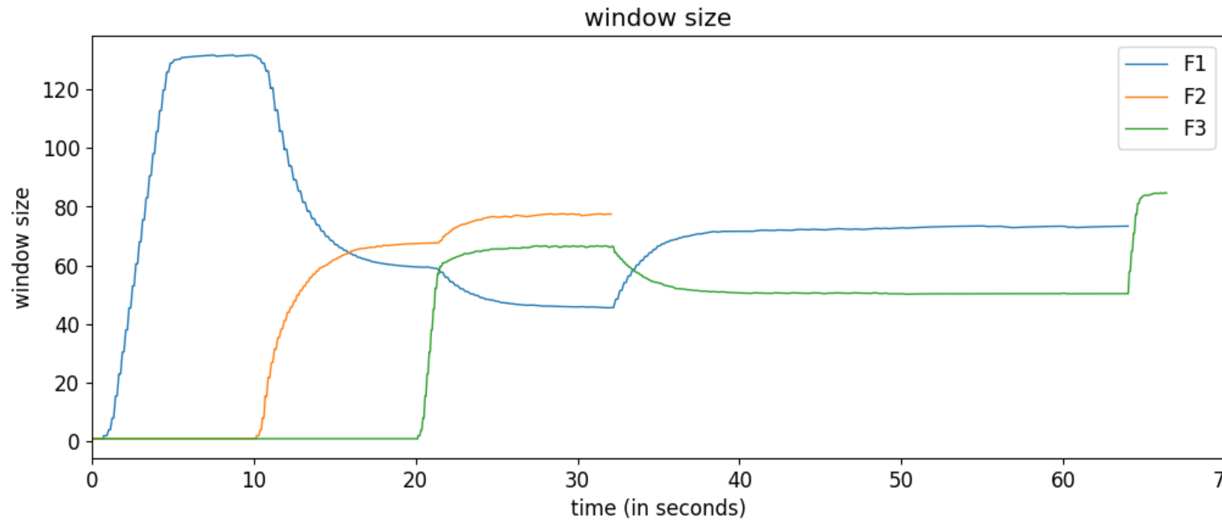

flow RTT
round trip time (in seconds)
time (in seconds)
F1
F2
F3
0.05
0.10
0.15
0.20
0.25
0.30
0
10
20
30
40
50
60
70


link rate
link rate (in Mbps)
time (in seconds)
L1:R1->R2
L2:R2->R3
L3:R3->R4
0
2
4
6
8
10
0
10
20
30
40
50
60
70

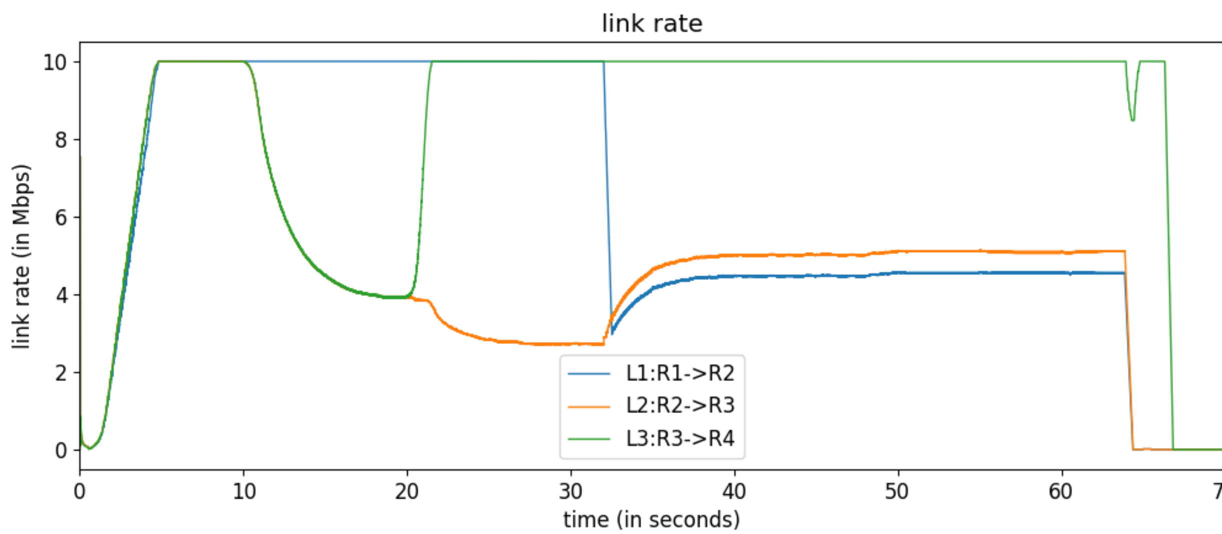

packet loss
number of packets dropped
time (in seconds)
L1
L2
L3
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
70


window size
window size
time (in seconds)
F1
F2
F3
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70

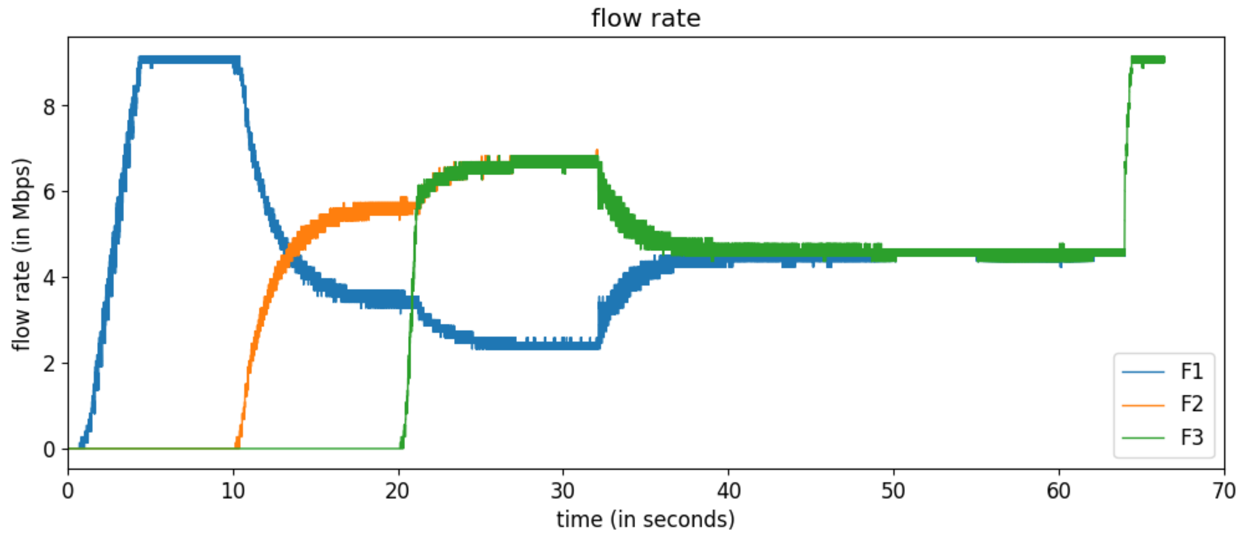
6.1.2 TCP Fast

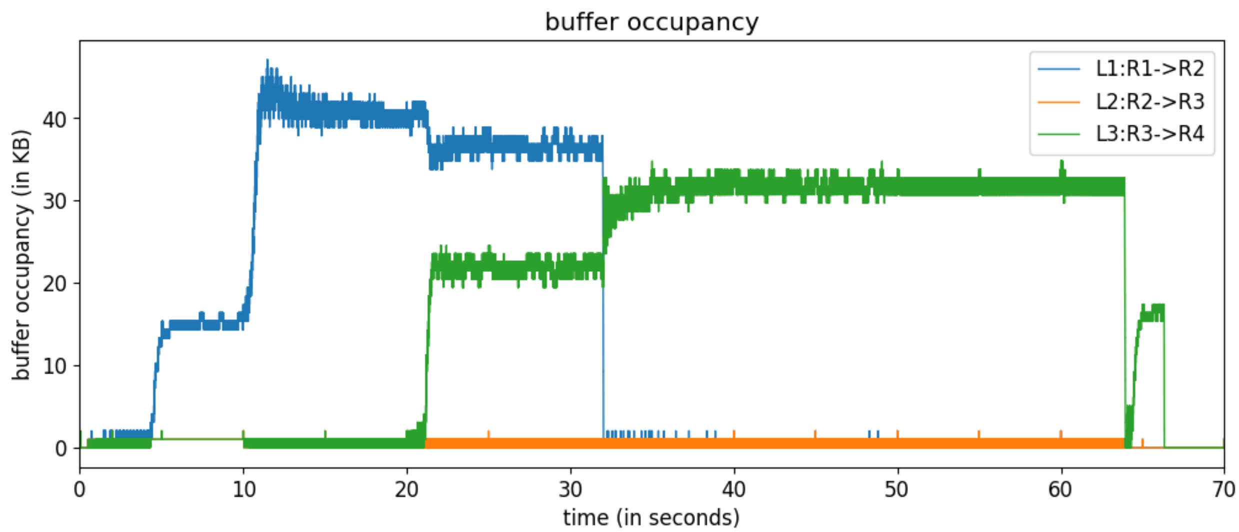


Here we have the window size graph of our flow for TCP fast. Looking at three critical points, 0, 10, and 20 seconds, where each flow starts. Each flow initially starts in the fast slow start state, which causes each flow to increase its window size until the throughput that the flow is using in the last round trip time is less than 90 percent of the expected throughput. At that point, each of our flows enters congestion control and reaches equilibrium. Then, when the next flow starts, our equilibrium window size will lower in order to adjust to the window size that the new flow is contributing.



Here we have the link rate measurements of link 1, 2, and 3 in test case 2. Initially, we only have 1 flow using all three links, which means that all three links can be used to full capacity with the one flow. When our second flow enters, it is only using link 1, which then means that link 1 is going to be used to full capacity, whereas the flow rate of our first flow will be lowered, and thus will make the link rate of links 2 and 3 lower because they will not be used to their full potential. Then, when our third flow starts and all three flows are running simultaneously, we can see that link 1 and 3 are being used by two flows and thus will be used to their full capacity, whereas the second link will be used only by flow 1. Finally, when just flow 1 and flow 3 are active, we can see that link 3 is being used to full capacity. Link 1 and link 2 are only being used by flow 1, but when flow 2 died flow 1 sees link 2 as having some small amount of packets in the buffer, (because of how our links have priorities) which is why it will then be used slightly faster than link 1.





Note that in the flow rate for fast, we have unexplained behavior in which flow 2 has a larger rate than flow 1, and has the same flow rate as flow 3.

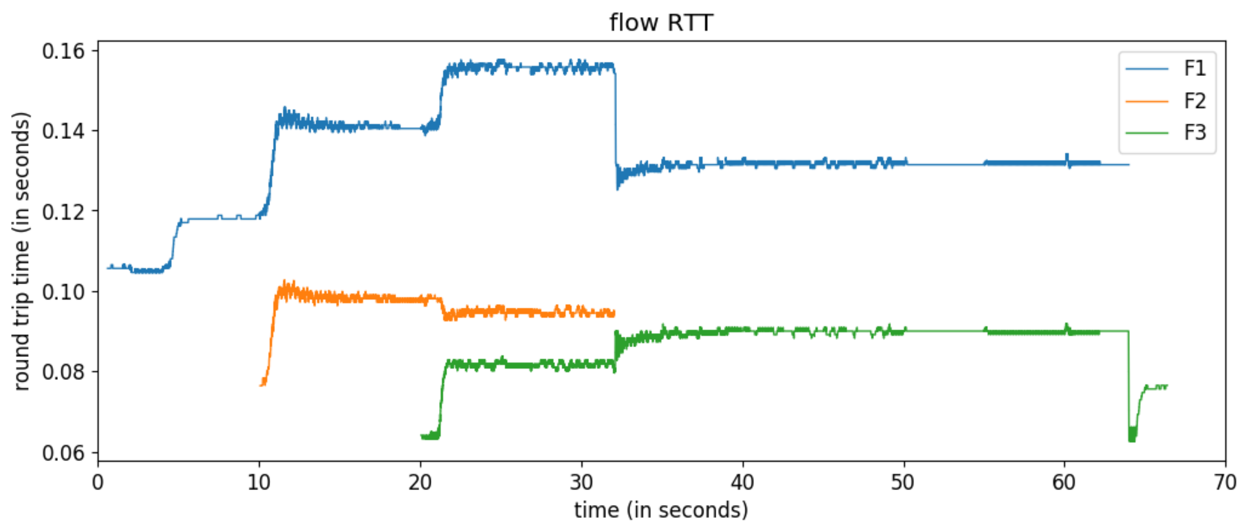
We can explain this with the following derivation. First, we can model the throughput of our function using FAST for some flow i as: $x_i = \frac{\alpha}{q_i}$

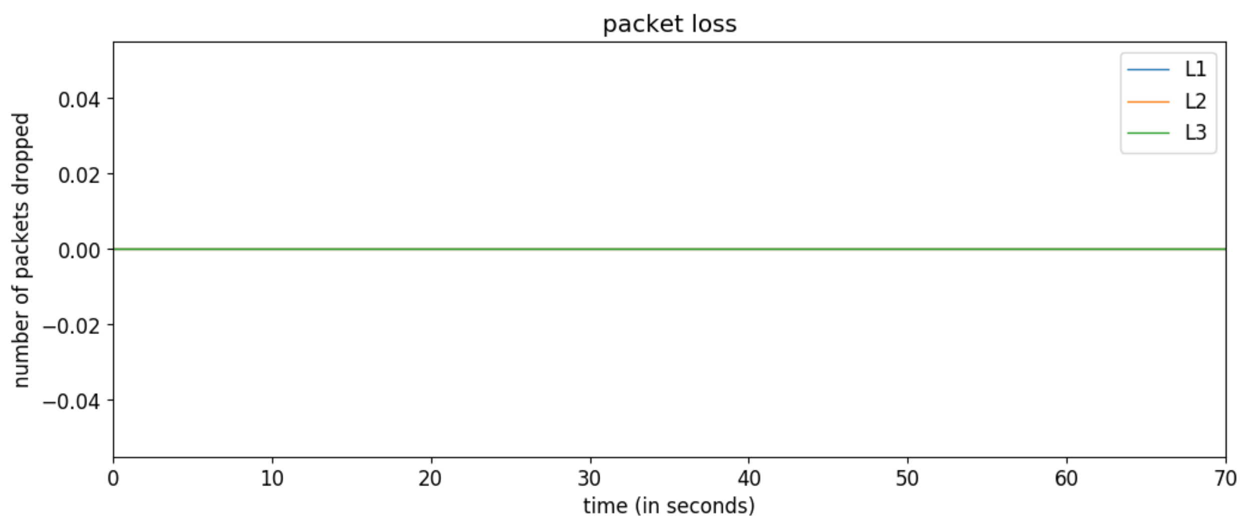
In the time in question, we see that flows 1 and 2 share L1, and flows 1 and 3 share L3, which then we know both become the bottleneck links. We then have $x_1 + x_2 = 10$ and $x_1 + x_3 = 10$. We know that L2 is underutilized, which means there are no queues on L2 and thus $p_2 = 0$ where p is the queuing delay on link 2.

We know, based on our initial claim, $x_2 = \frac{\alpha}{p_1} = \frac{\alpha}{p_3} = x_3$

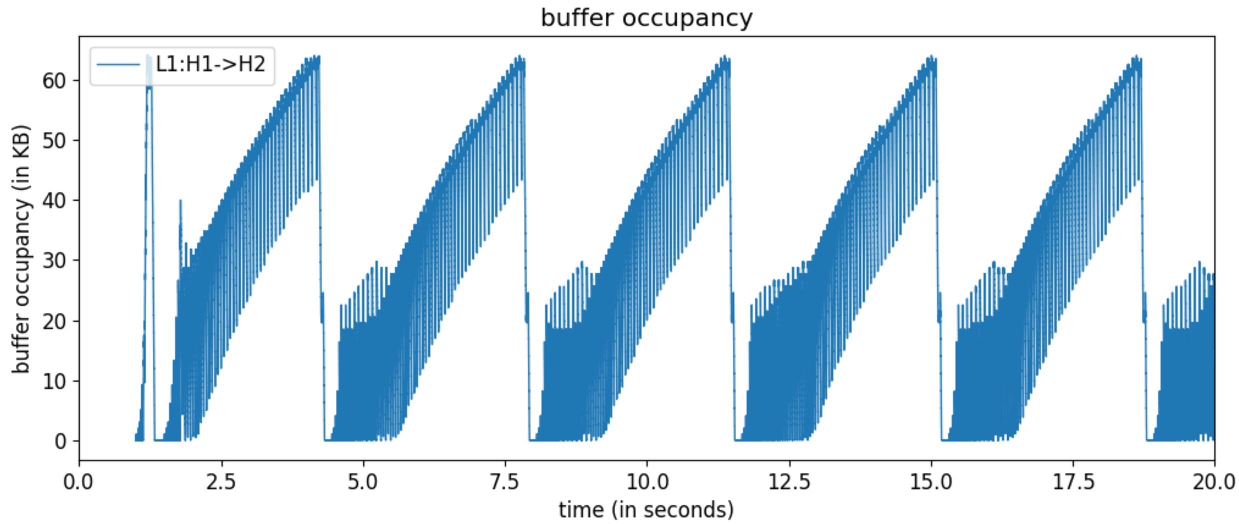
We also know, $x_1 = \frac{\alpha}{p_1+p_2} = \frac{x_2}{2}$ because we know the queuing delay is the same on both $p_1 and p_2$ (because of how we defined the links).

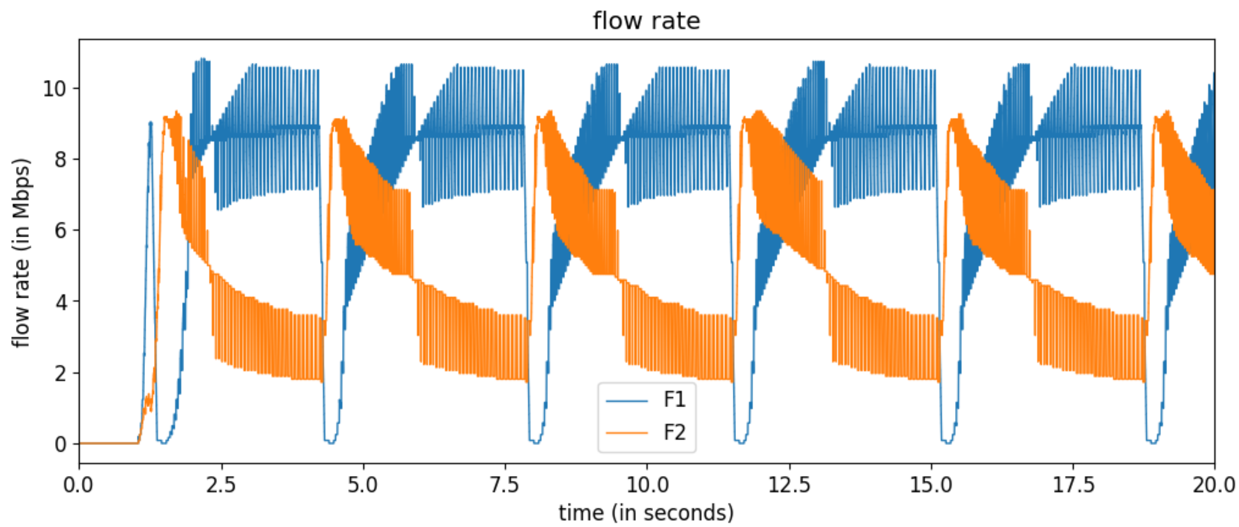
Thus, we get a ratio between x_1 and x_2 of 1:2, which is what we see in the graphs and thus explains our interesting behavior.

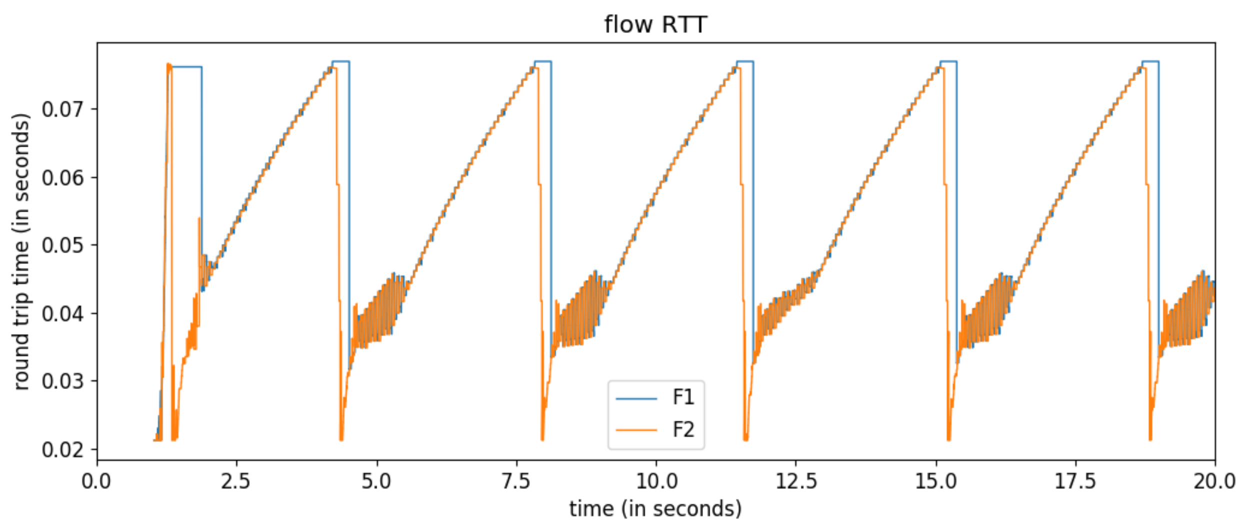

flow RTT
round trip time (in seconds)
time (in seconds)
F1
F2
F3
0.16
0.14
0.12
0.10
0.08
0.06
0
10
20
30
40
50
60
70

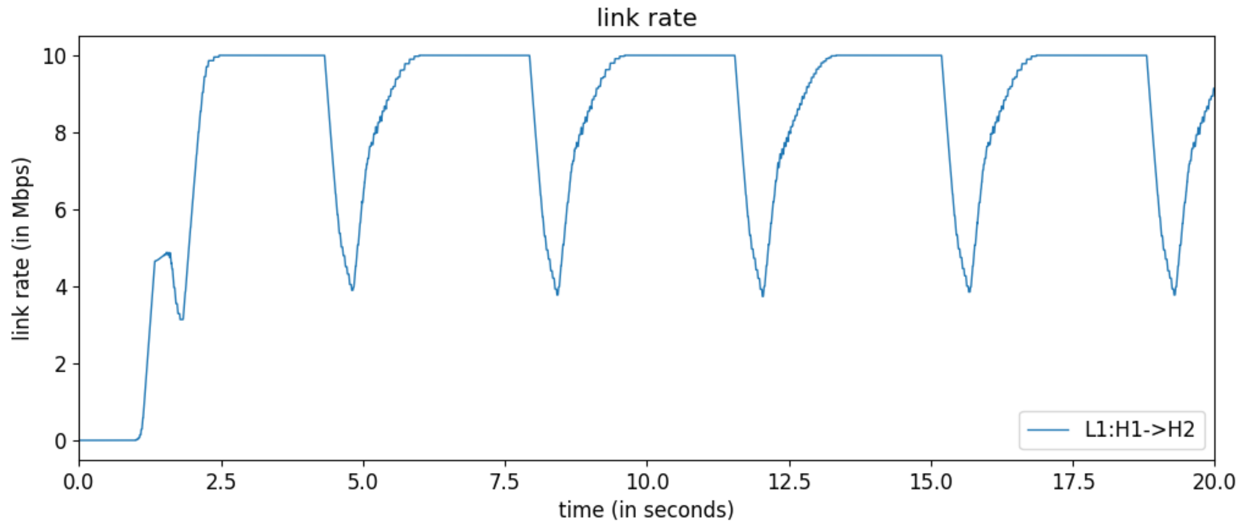

packet loss
number of packets dropped
time (in seconds)
L1
L2
L3
0.04
0.02
0.00
−0.02
−0.04
0
10
20
30
40
50
60
70

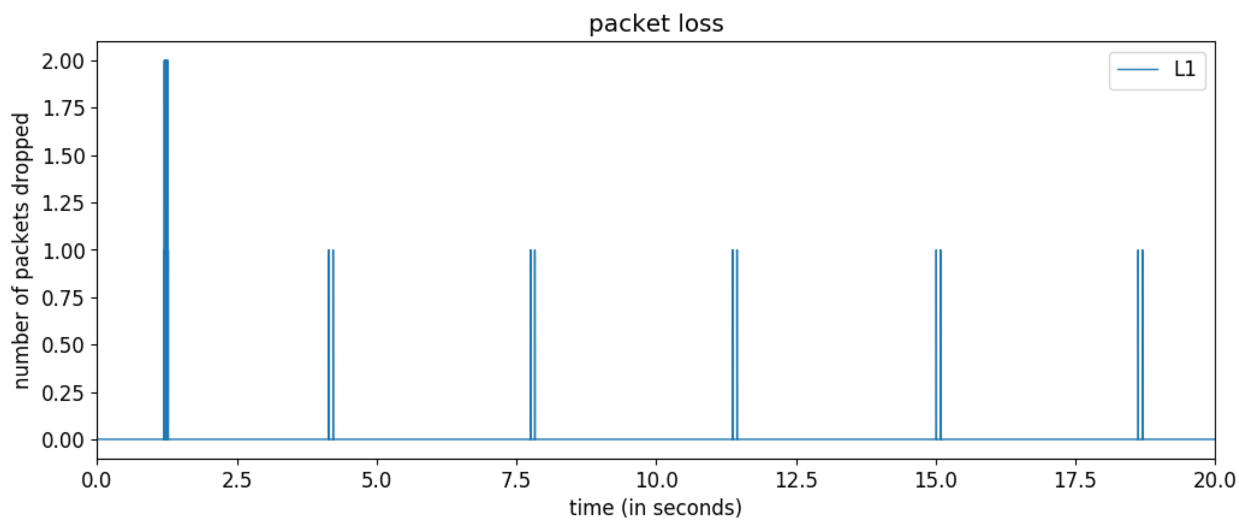
6.2 Test Case 3

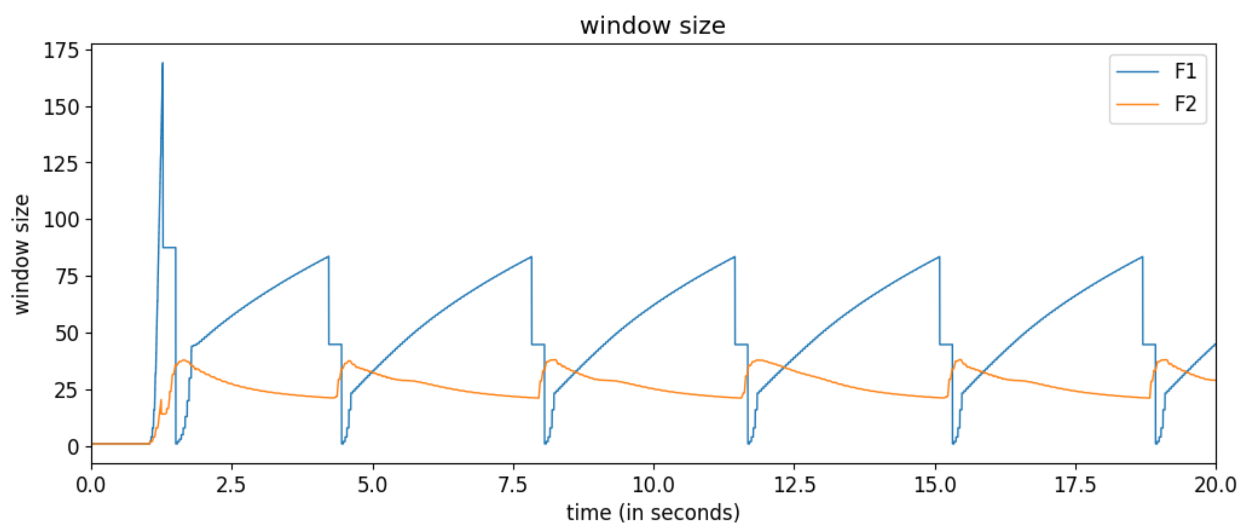




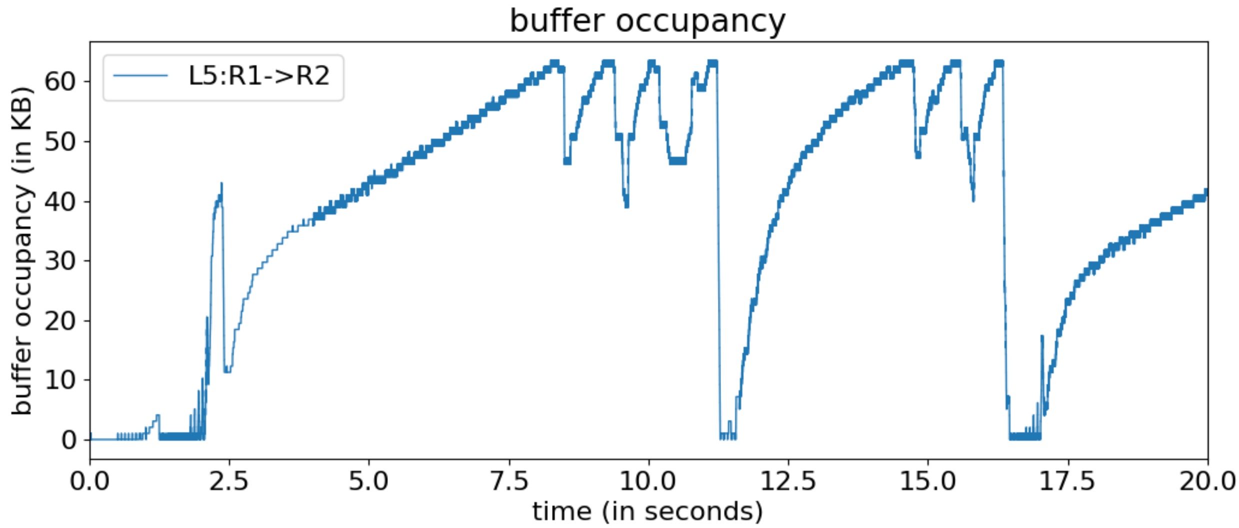


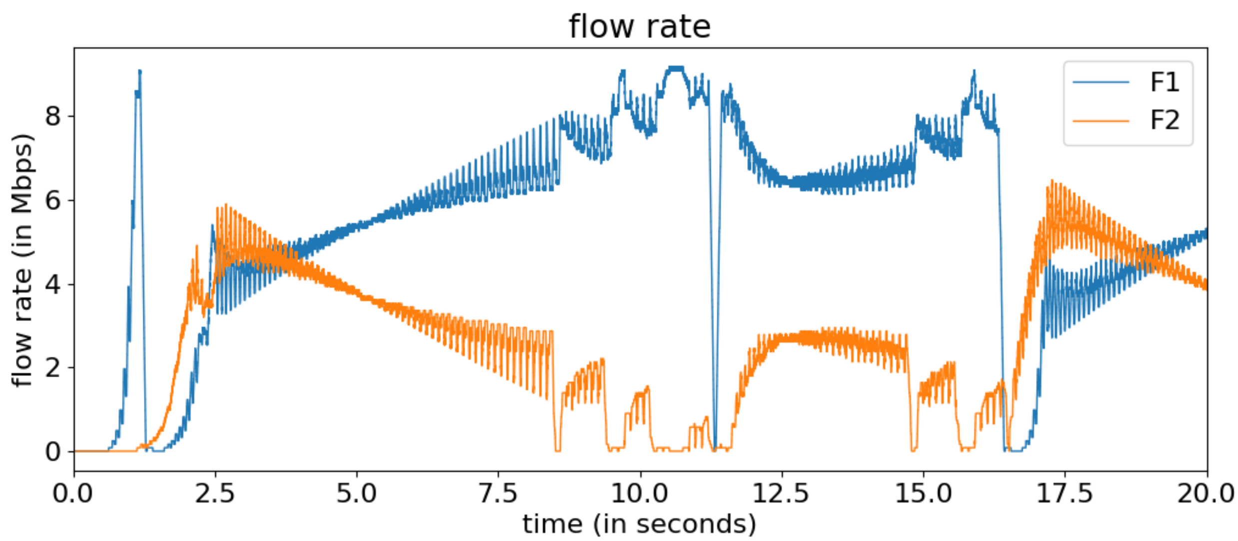

link rate
link rate (in Mbps)
time (in seconds)
L1:H1->H2

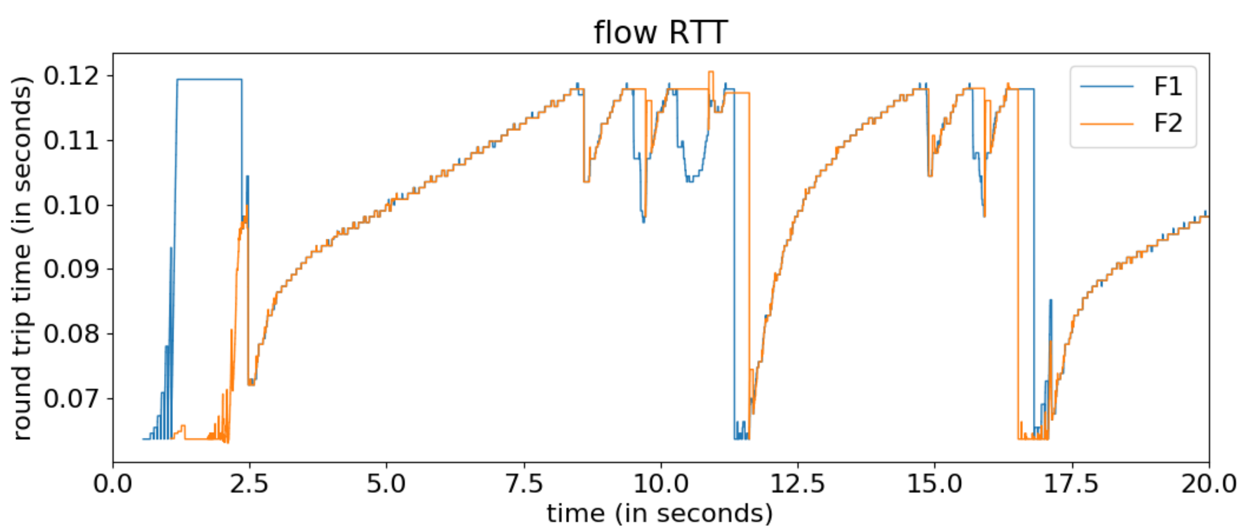

packet loss
number of packets dropped
time (in seconds)
L1

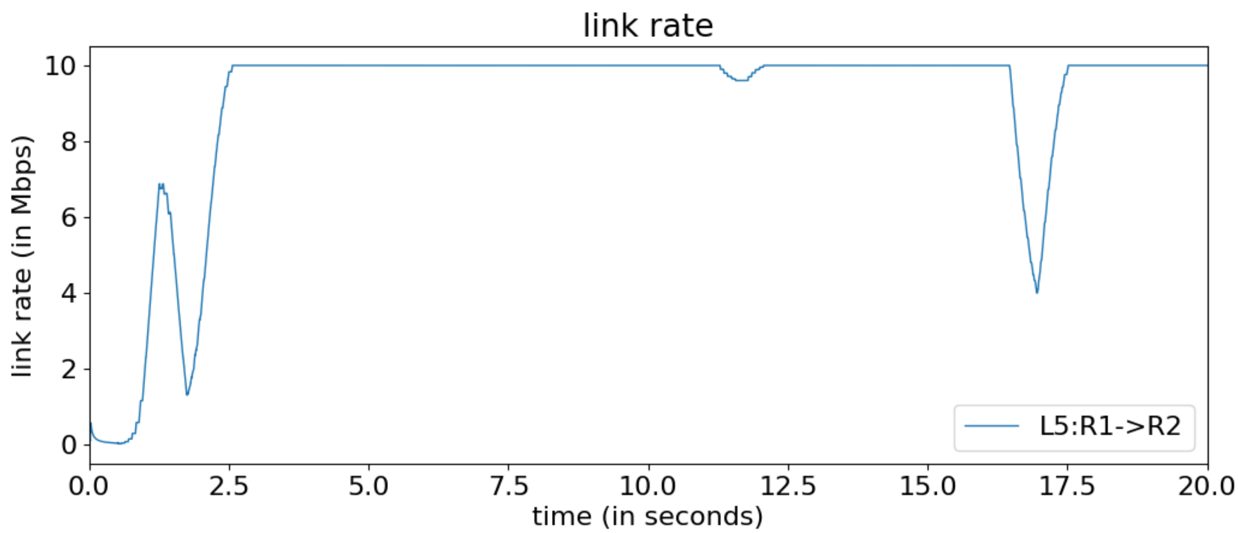

window size
window size
time (in seconds)
F1
F2

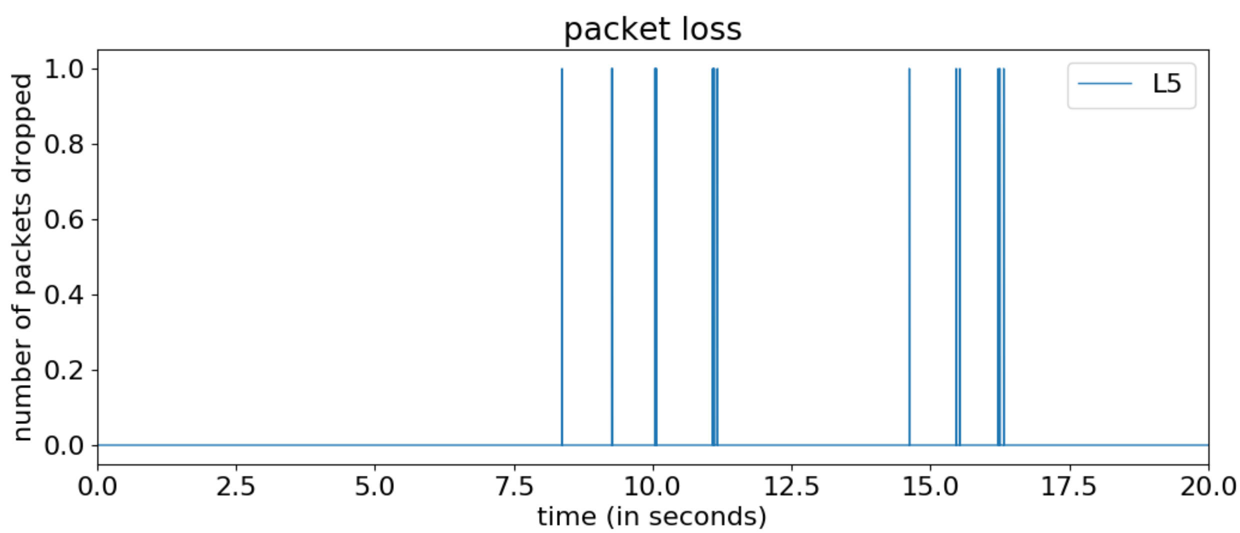
6.3 Test Case 4

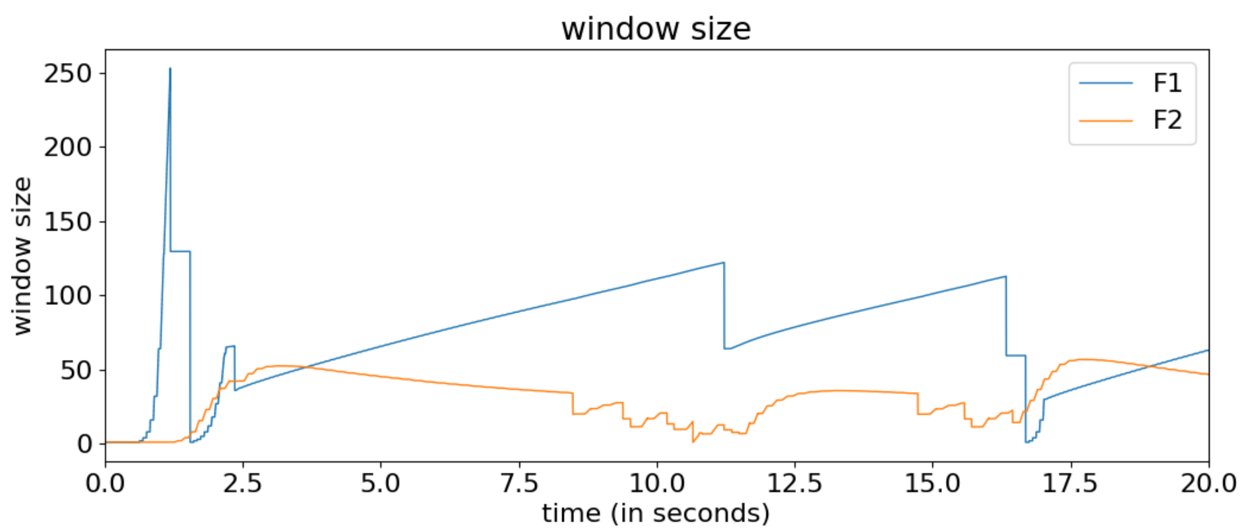






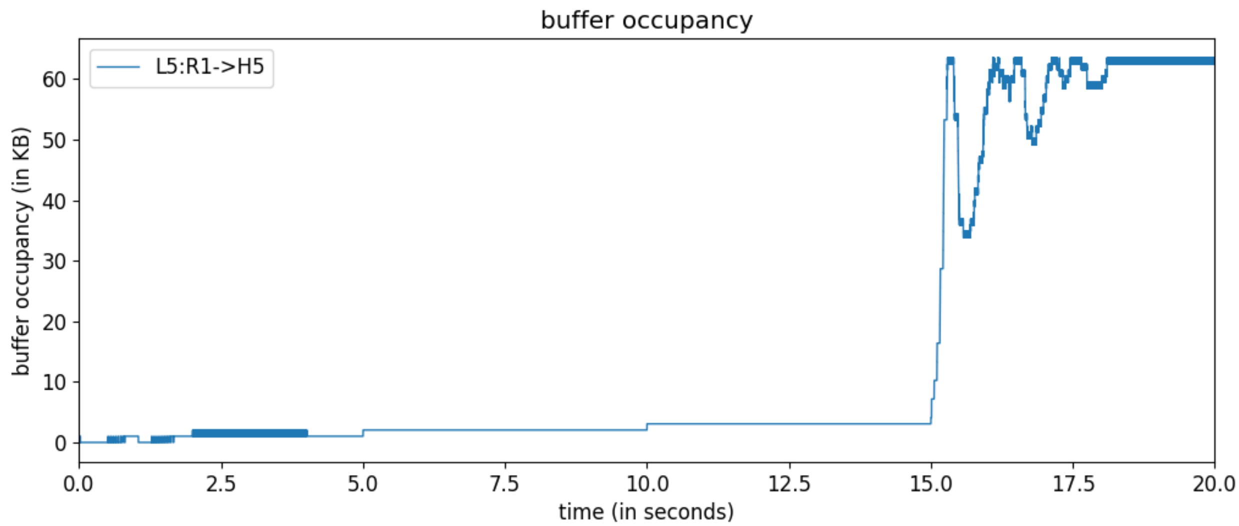

link rate
link rate (in Mbps)
time (in seconds)
L5:R1->R2

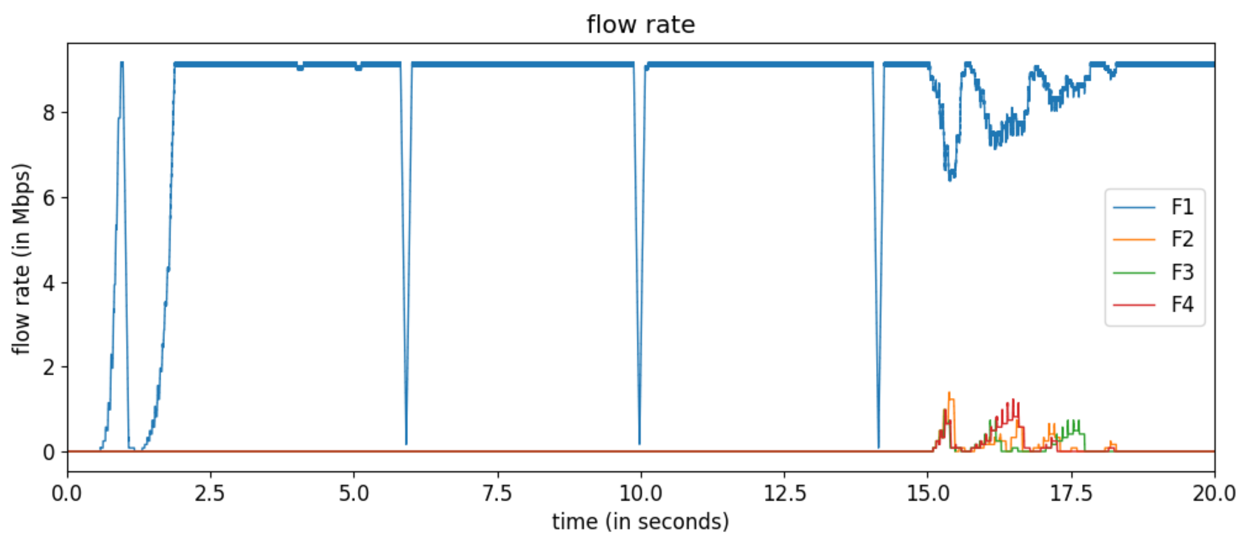

packet loss
number of packets dropped
time (in seconds)
L5

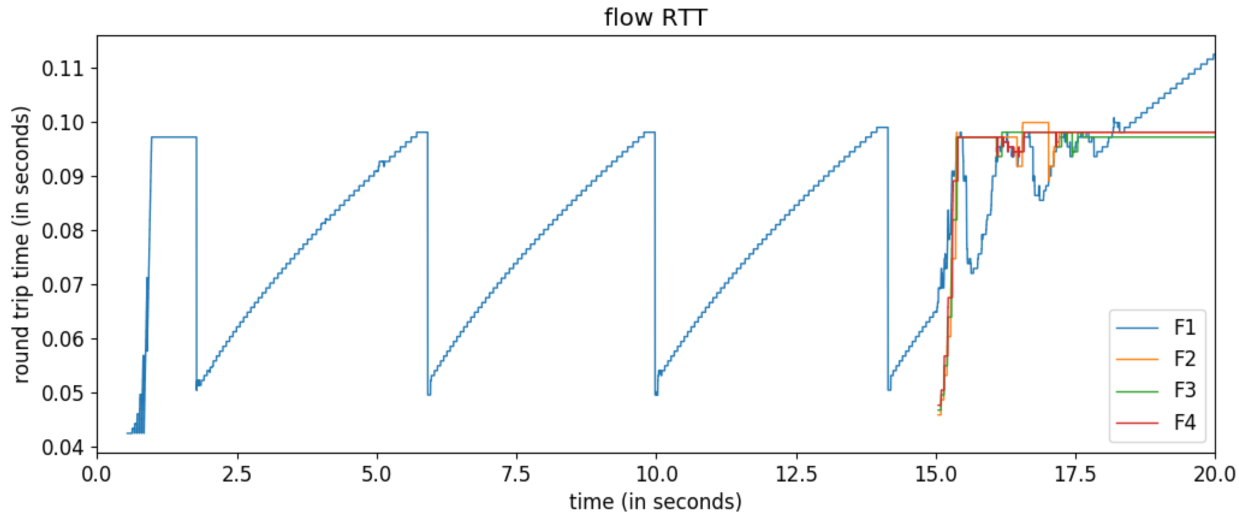

window size
window size
time (in seconds)
F1
F2

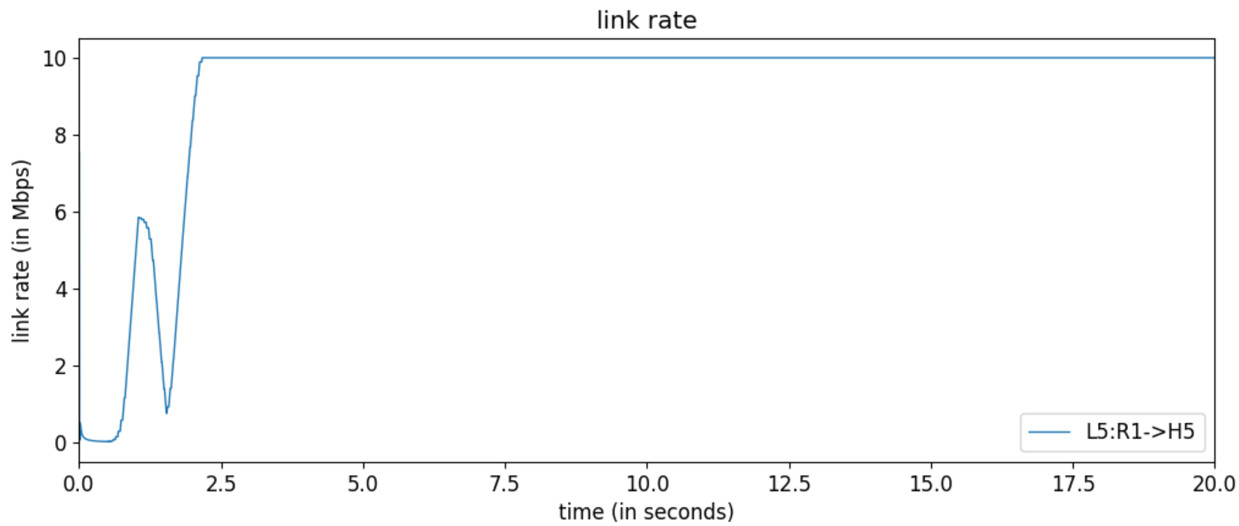
6.4 Test Case 5

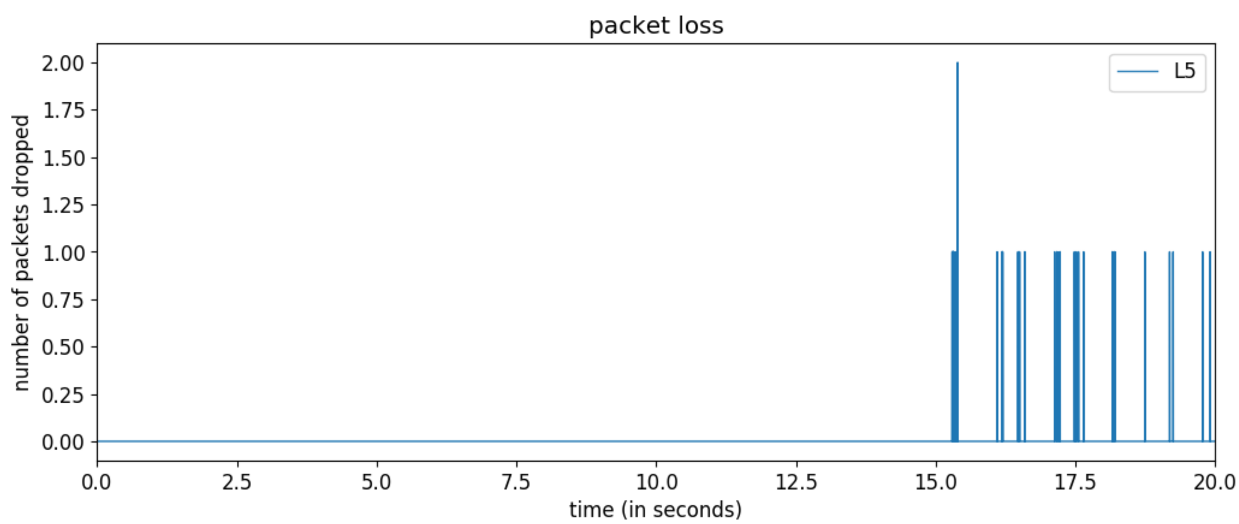
6.4.1 TCP Reno

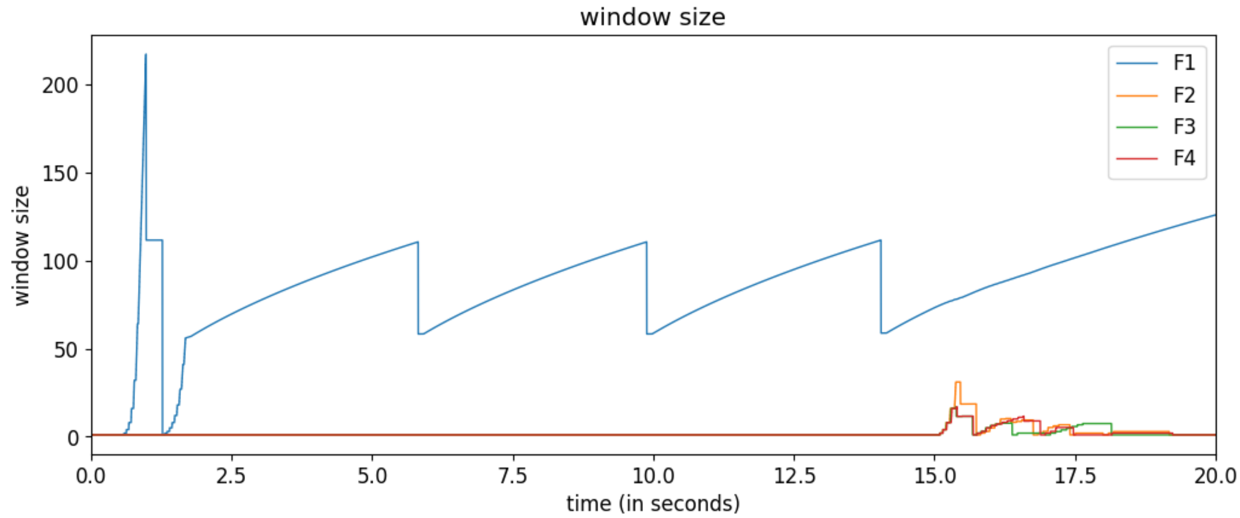




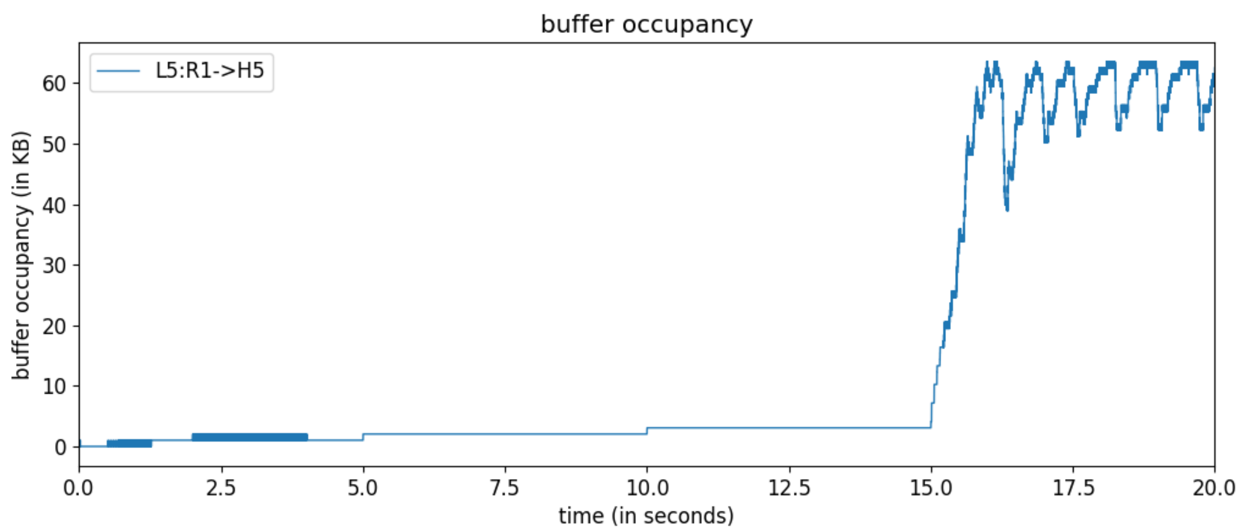

flow RTT
round trip time (in seconds)
time (in seconds)
F1
F2
F3
F4

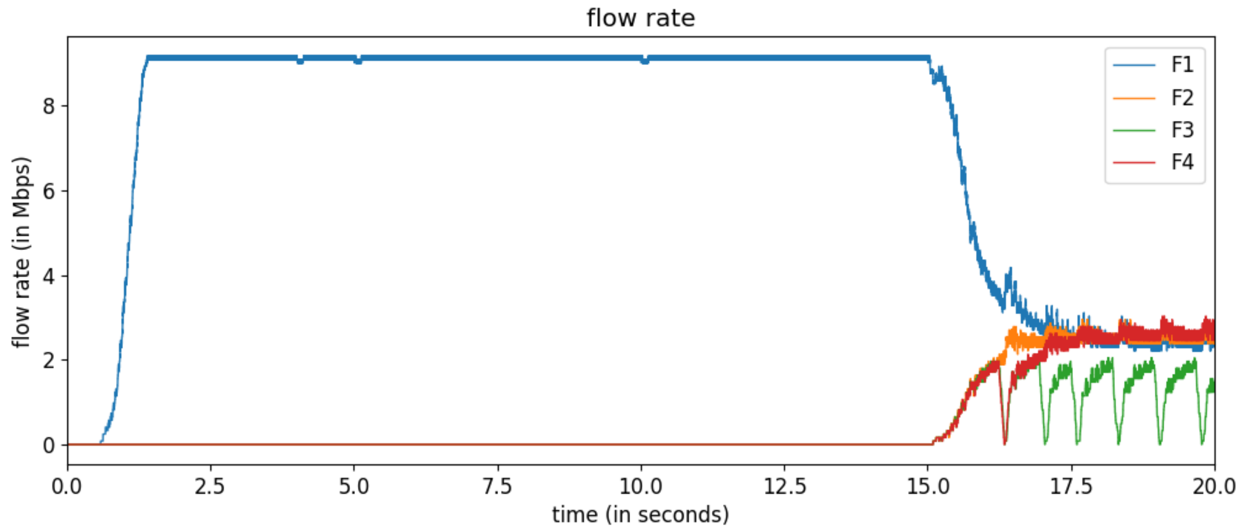

link rate
link rate (in Mbps)
time (in seconds)
L5:R1->H5

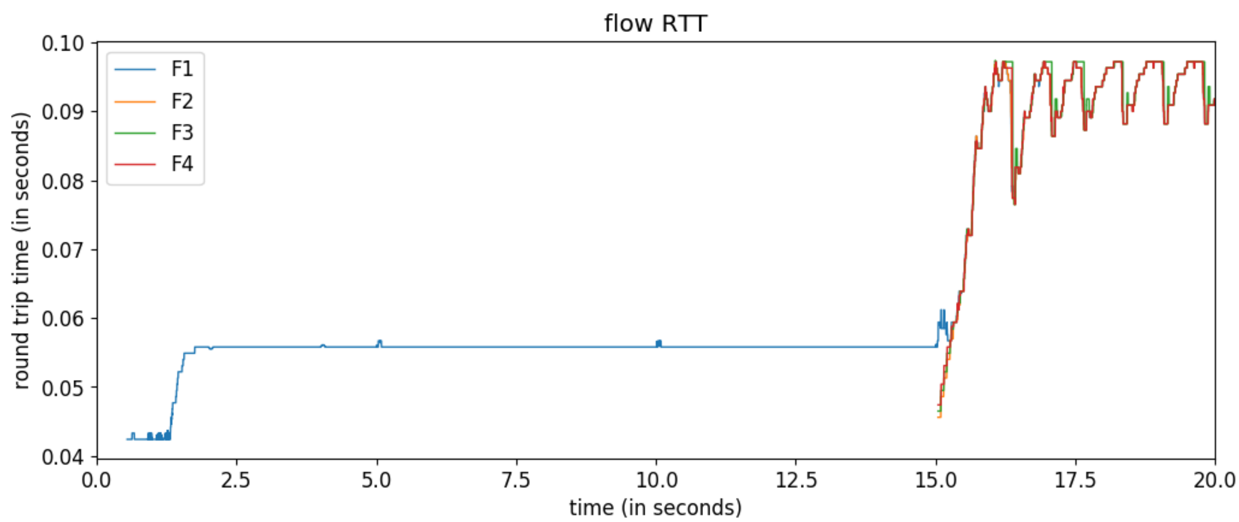

packet loss
number of packets dropped
time (in seconds)
L5

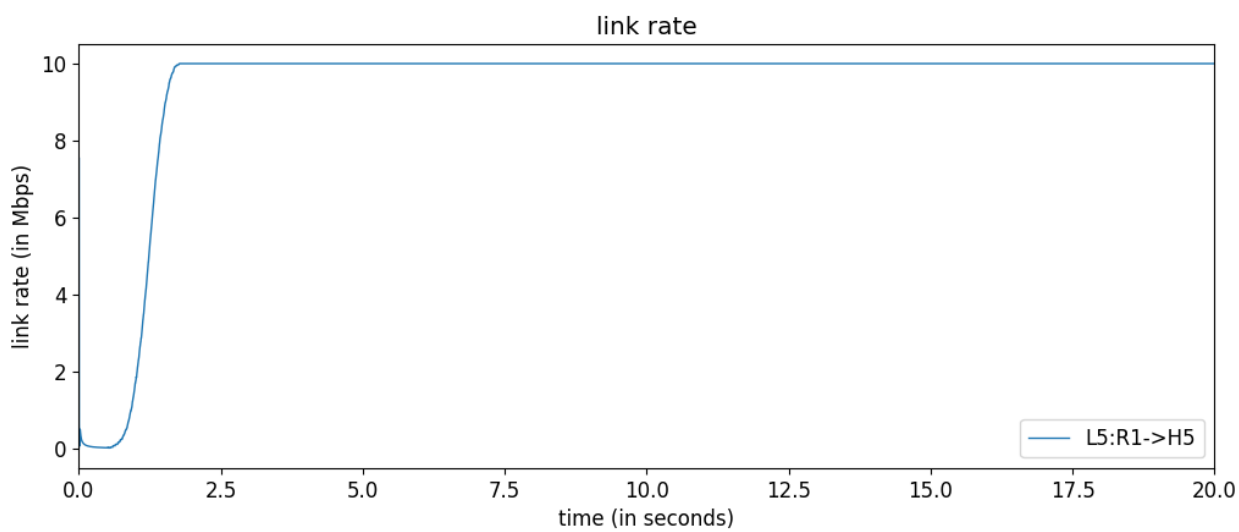

window size
F1
F2
F3
F4
window size
time (in seconds)

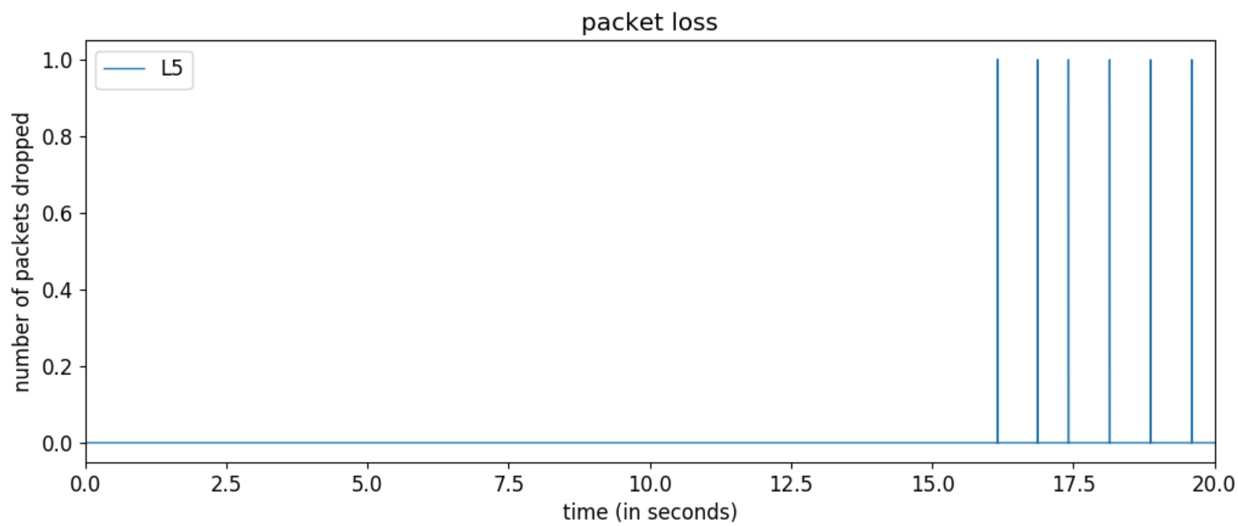
6.4.2 TCP Fast

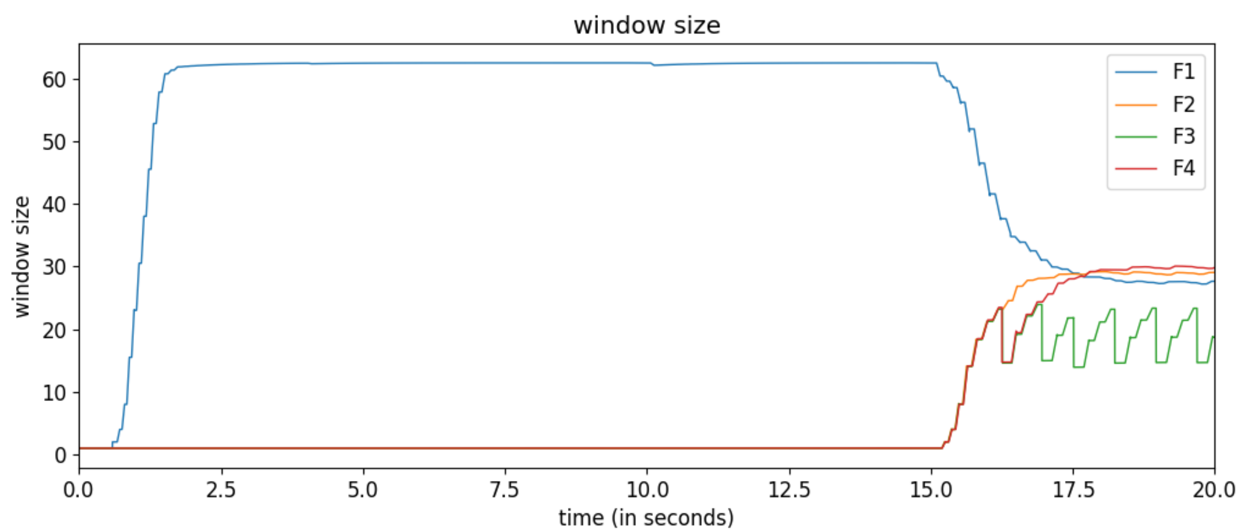

buffer occupancy
L5:R1->H5
buffer occupancy (in KB)
time (in seconds)


flow rate
flow rate (in Mbps)
time (in seconds)
F1
F2
F3
F4

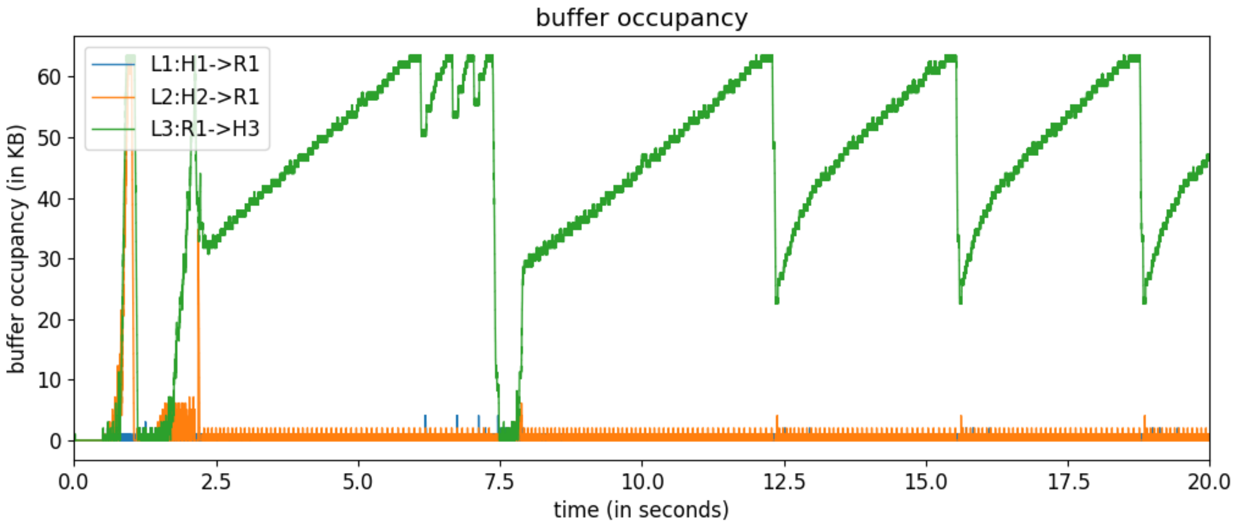

flow RTT
round trip time (in seconds)
time (in seconds)
F1
F2
F3
F4

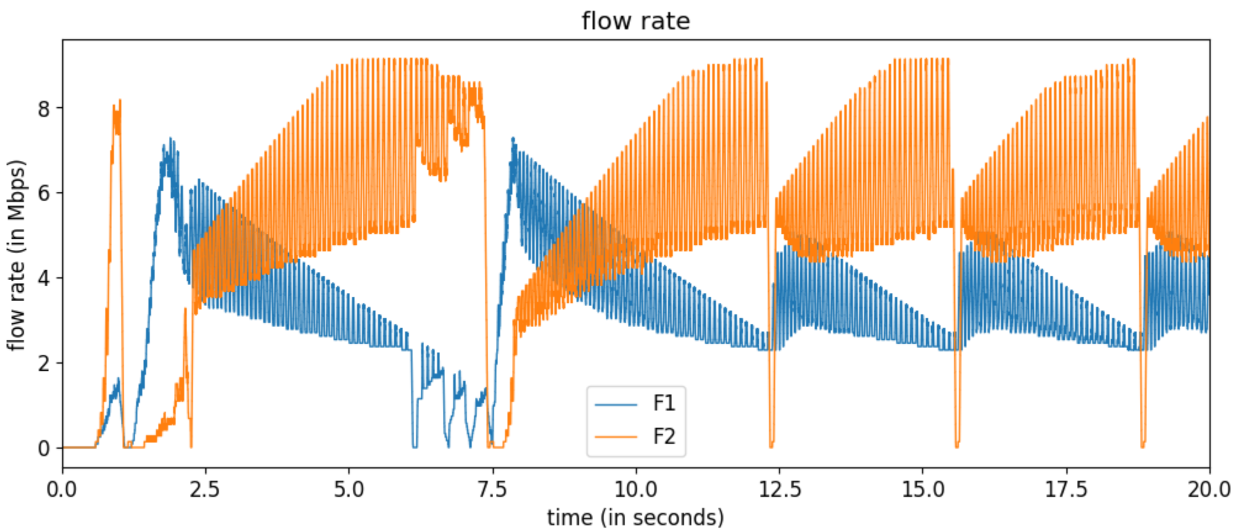

link rate
link rate (in Mbps)
time (in seconds)
L5:R1->H5

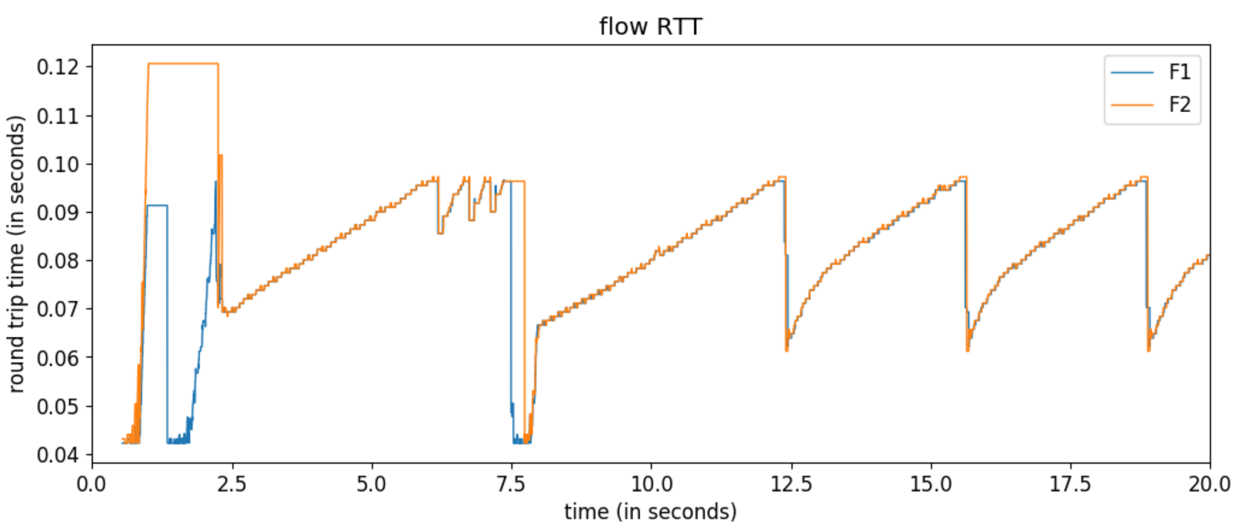

packet loss
L5
number of packets dropped
1.0
0.8
0.6
0.4
0.2
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
time (in seconds)

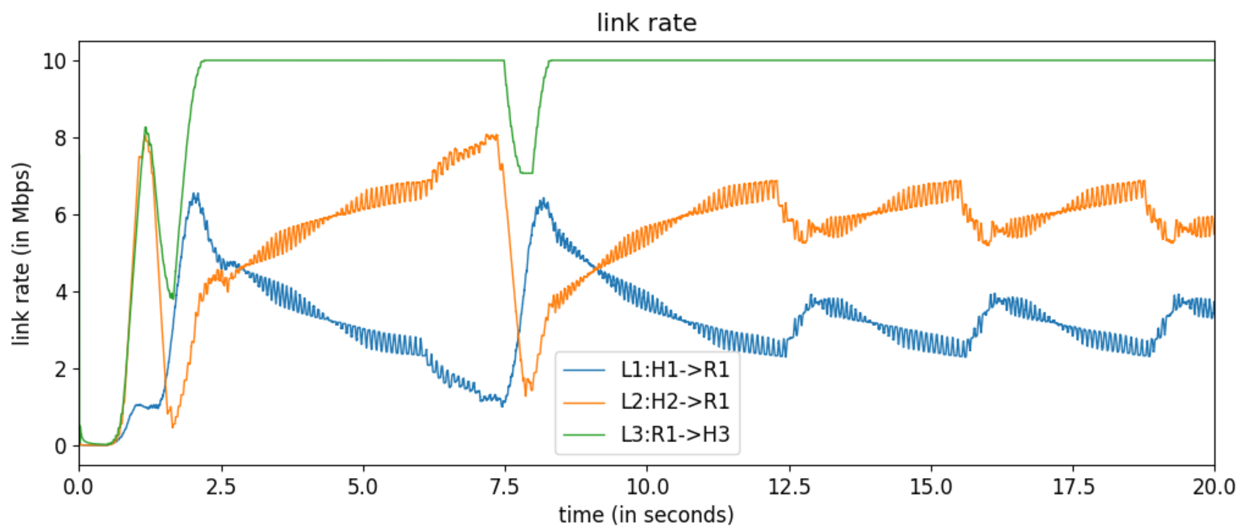

window size
F1
F2
F3
F4
window size
60
50
40
30
20
10
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
time (in seconds)

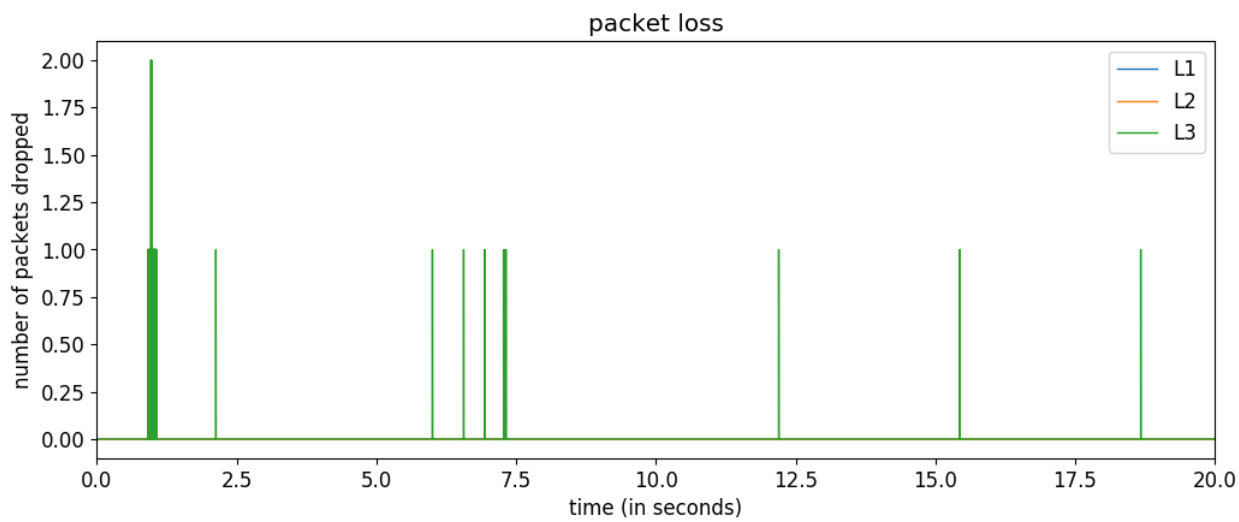
6.5 Our Test Case

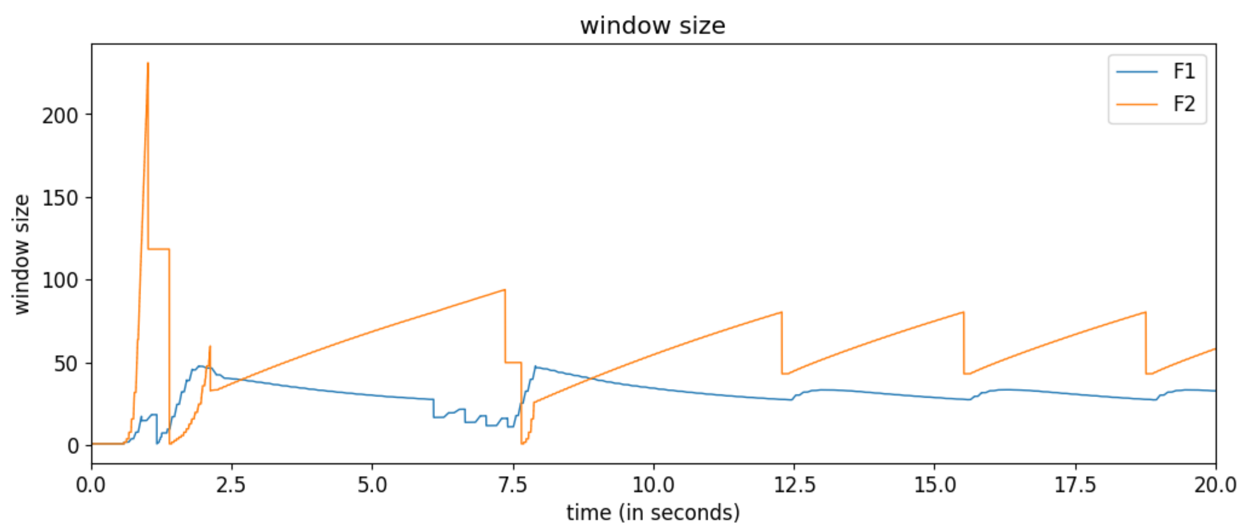







link rate
link rate (in Mbps)
time (in seconds)
L1:H1->R1
L2:H2->R1
L3:R1->H3
0
2
4
6
8
10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


packet loss
number of packets dropped
time (in seconds)
L1
L2
L3
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


window size
window size
time (in seconds)
F1
F2
0
50
100
150
200
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

7 Timeline & Division of Labor

7.1 Timeline

- **Weeks 4-6:** Completed initial architecture design of simulator and initial implementation of a few simulator classes, such as Packet and Link classes
- **Weeks 7-8:** Completed initial implementation of all class in the simulator, started debugging Router and Flow classes
- **Week 9:** Finished debugging fundamental architecture for simulator, started work on implementing and debugging TCP Reno congestion control
- **Week 10:** Finished implementation of TCP Reno Congestion Control, debugging metric tracking
- **Week 11:** Finished implementation of TCP Fast, finished report and presentation

7.2 Division of Labor

- **Everyone:** Initial architecture design, report & presentation slides
- **Rafael Fueyo-Gomez:** Hosts, routers, flow, congestion control (TCP Reno and TCP Fast)
- **Cortland Perry:** Routers, congestion control (TCP Fast)
- **Kelsi Riley:** Links, initial implementation, metric tracking, congestion control (TCP Fast)
- **Sakthi Vetrivel:** Flows, congestion control (TCP Reno)