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   Dynamic / Static Routing

Our routing tables are calculated in the following way. First, all of our routers send "handshakes" along each of the links that they are connected to. These handshakes go down each of these links, and then are received by either the host or routing table that is at the other end of the link. These routers / hosts then send back an acknowledgement packet that tells the router information about who is at the end of the link.

Using this information, each of our routers creates an initial representation of what we call the "link state array". This link state array contains all information about the cost of each link, and which nodes are connecting each of the links. This information is sorted in each routers own array like so: [Source, Dest, LinkID, Cost]. Note that the cost of each link is based on the link delay and the link rate. Our routers then every 5 seconds send their link state array to their neighboring routers, and then, after recalculating the costs of links adjacent to them send it to all the other routers / hosts in the network. Each router receives these link state arrays and updates their own for all links that are not connected to them. (Every link cost that has changed that is not connected to a given router is updated based on the information that was just sent to them).

Every time that a router updates its own link state array, it will then update its routing table and send its new link state array to neighbors. The routing table is updated and calculated using Dijkstra's shortest path algorithm using all of the information about the network that is stored in the link state array.

This process and methodology allows our routers to dynamically compute and change their routing tables based on the current state of our network.

4   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 process any acknowledgements or 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.

4.1   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.

5 Metrics Tracking

6 Results

6.1 Test Case 0

6.1.1 TCP Reno

6.1.2 TCP Fast

6.2 Test Case 1

6.2.1 TCP Reno

6.2.2 TCP Fast

6.3 Test Case 2

6.3.1 TCP Reno

6.3.2 TCP Fast

7 Analysis

7.1 Test Case 0

7.1.1 TCP Reno

flow rate

flow RTT

link rate

packet loss

window size

7.1.2 TCP Fast

buffer occupancy

flow rate

flow RTT

link rate

packet loss

window size
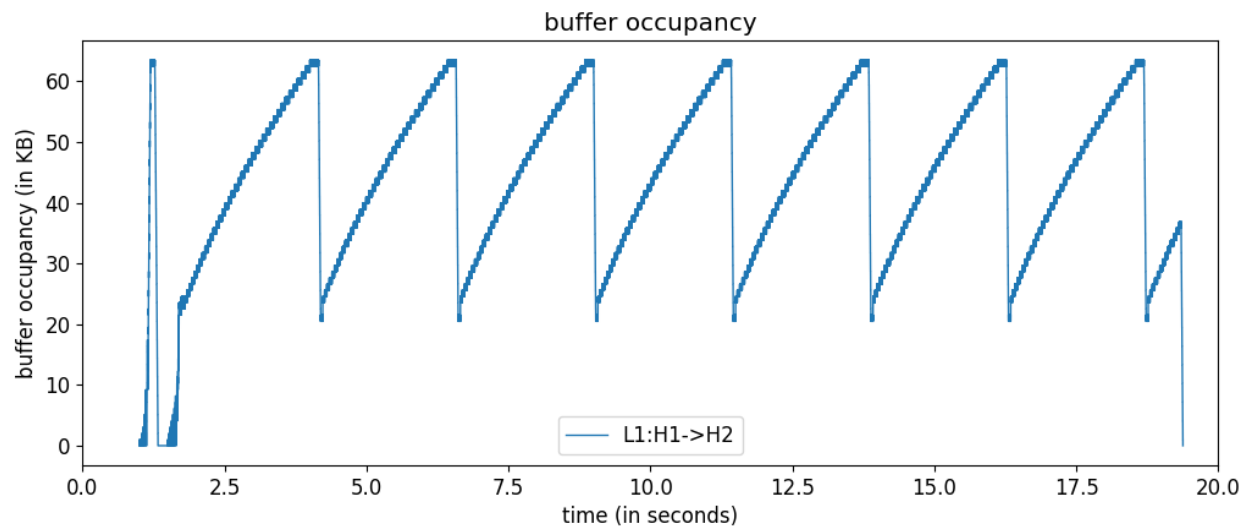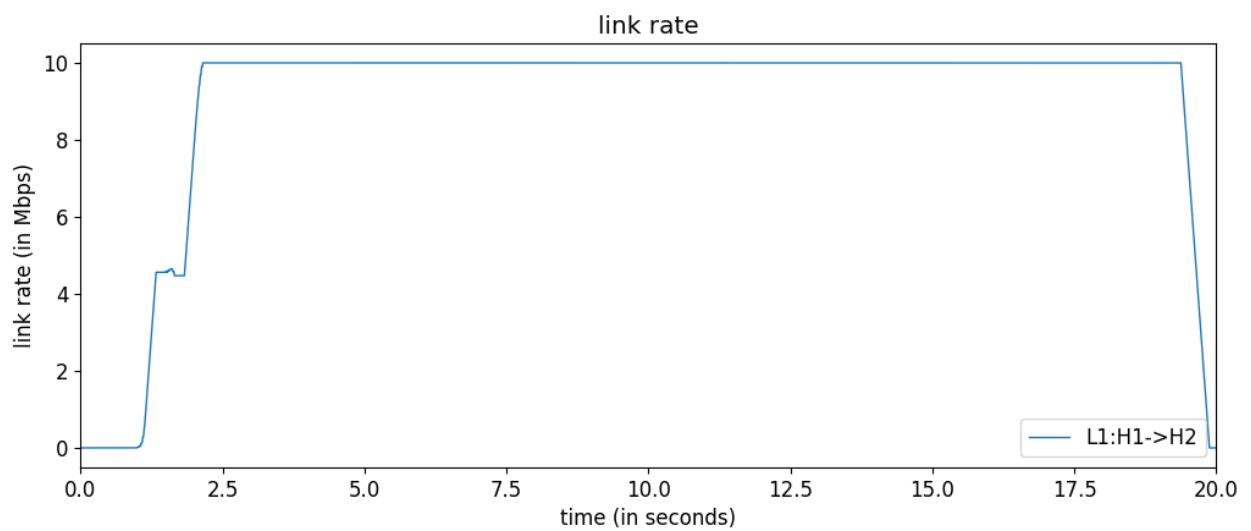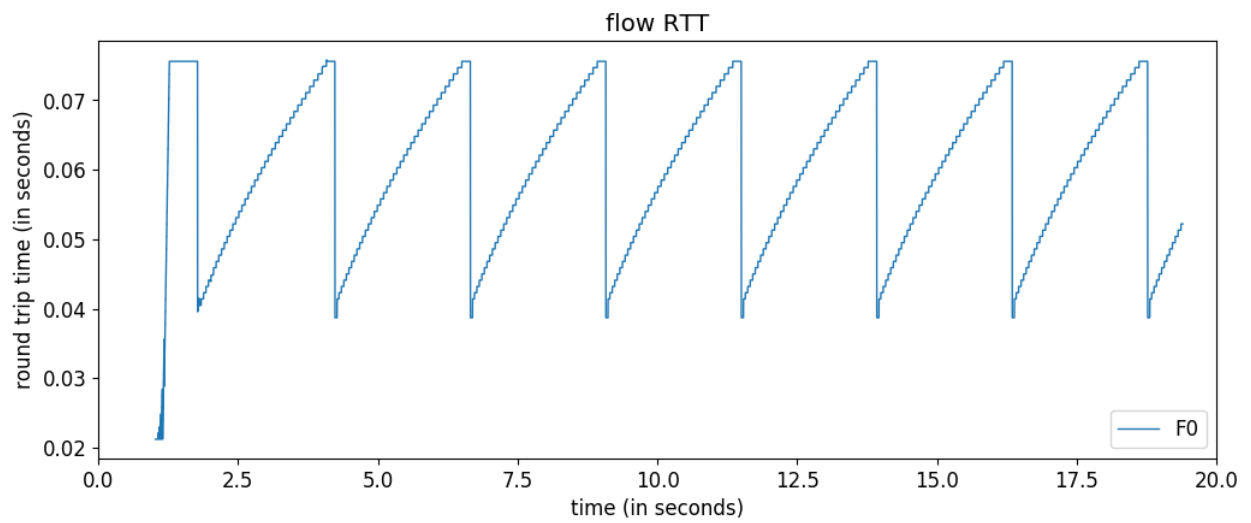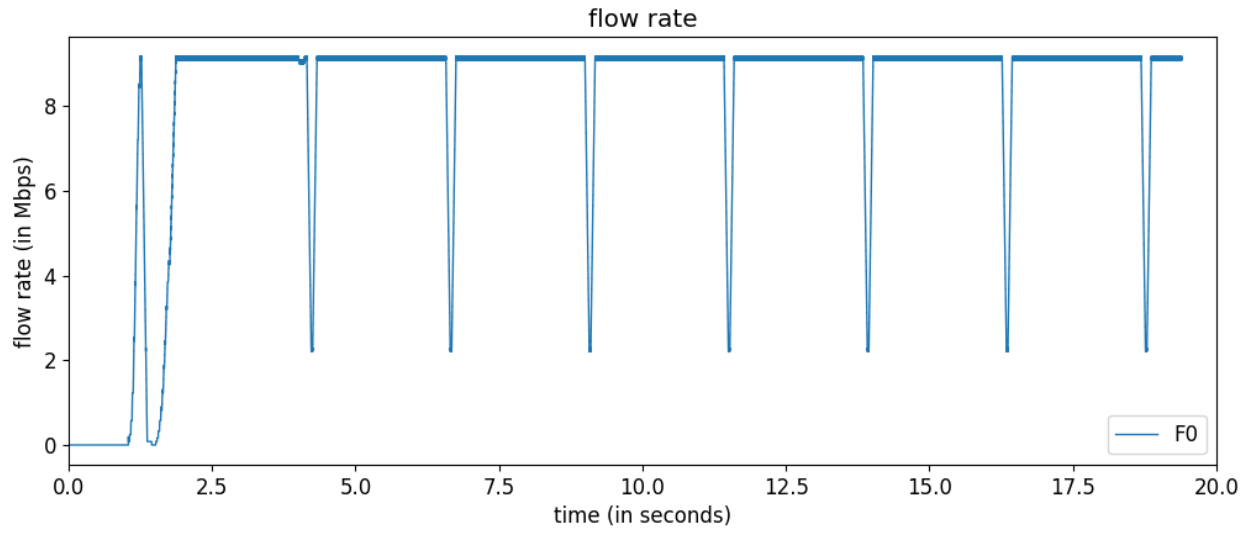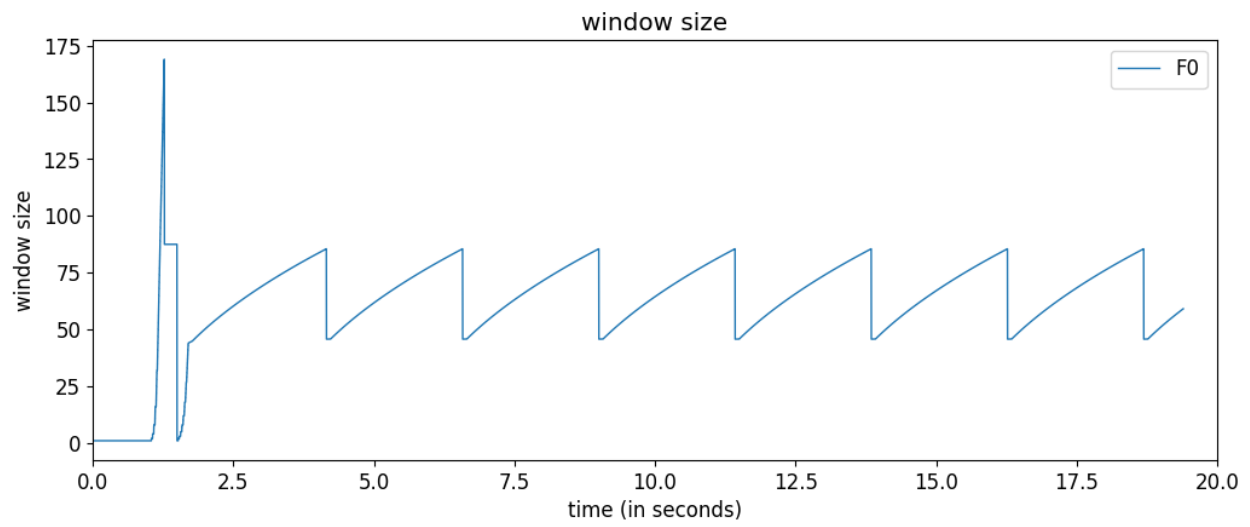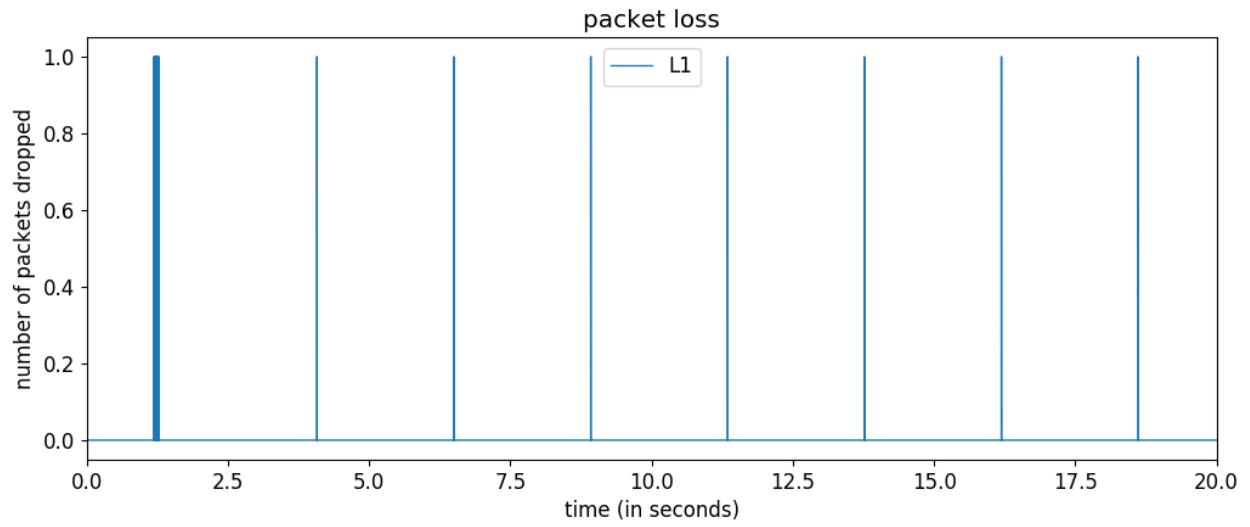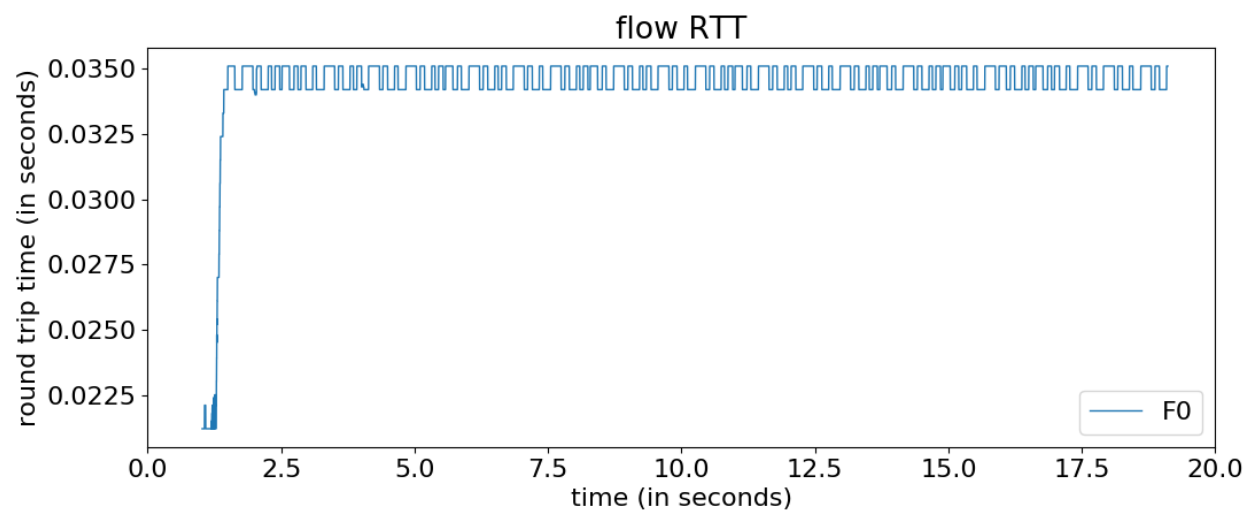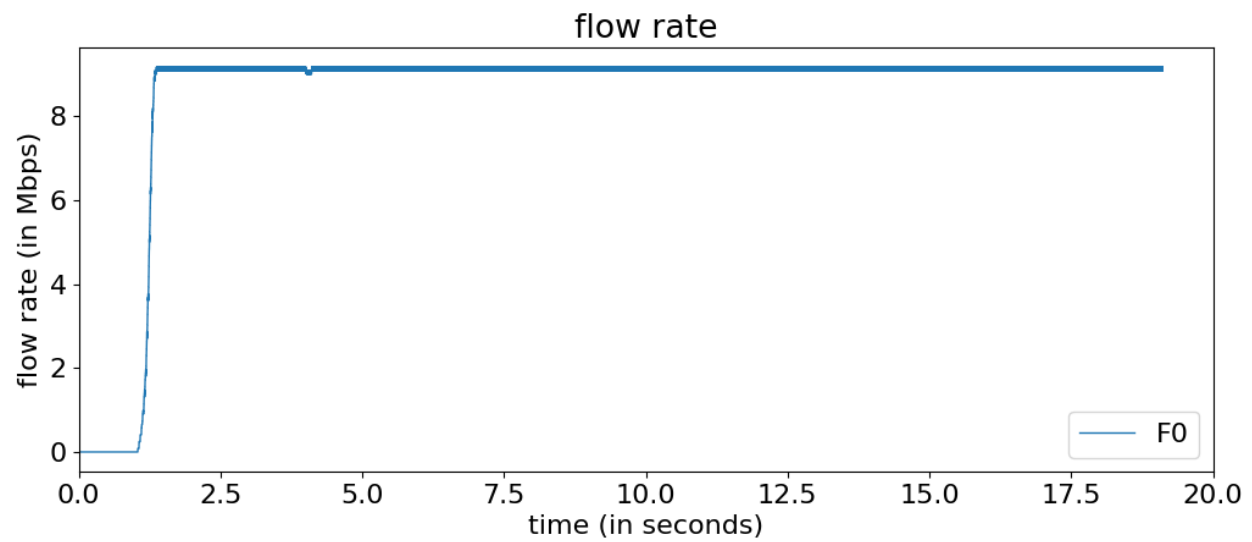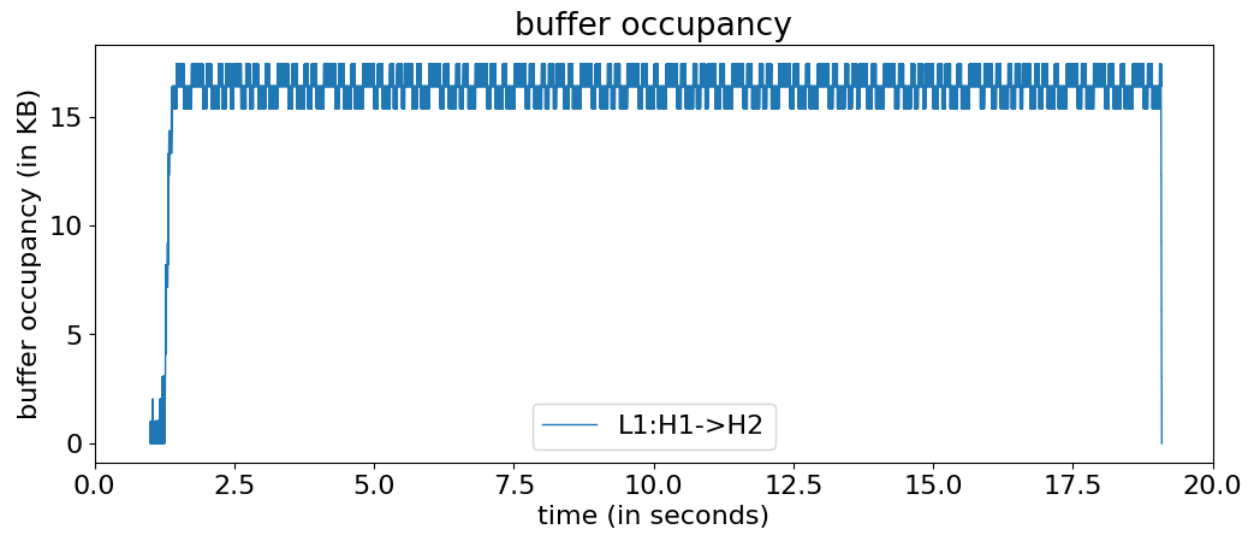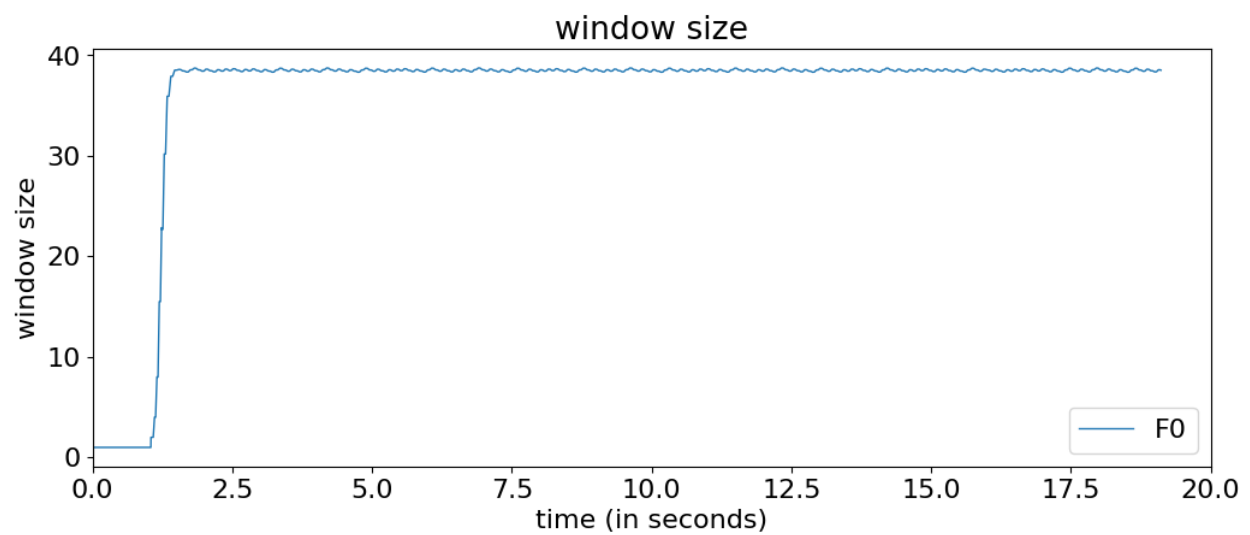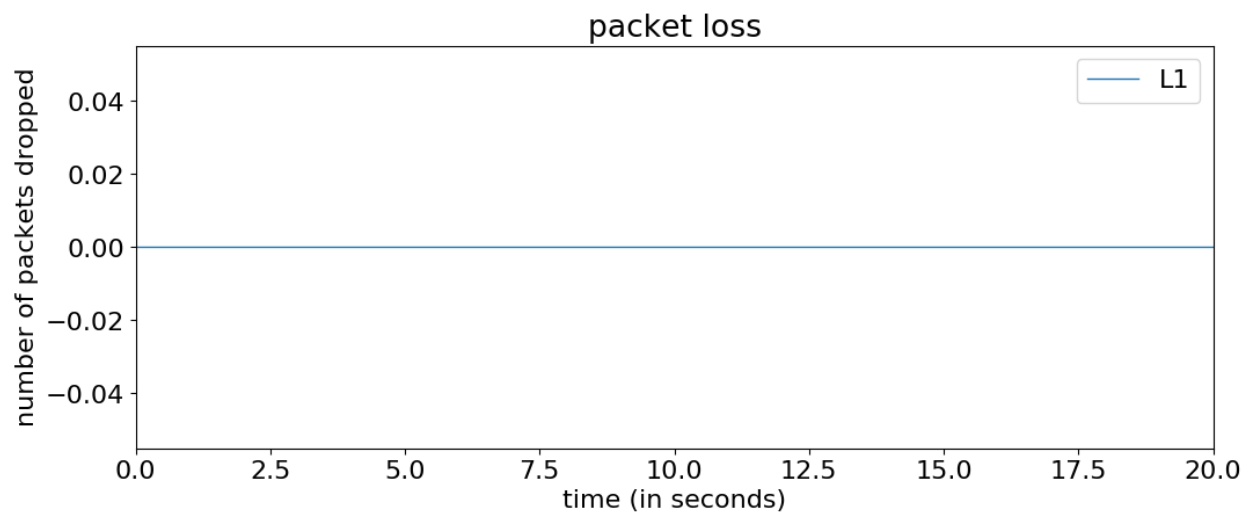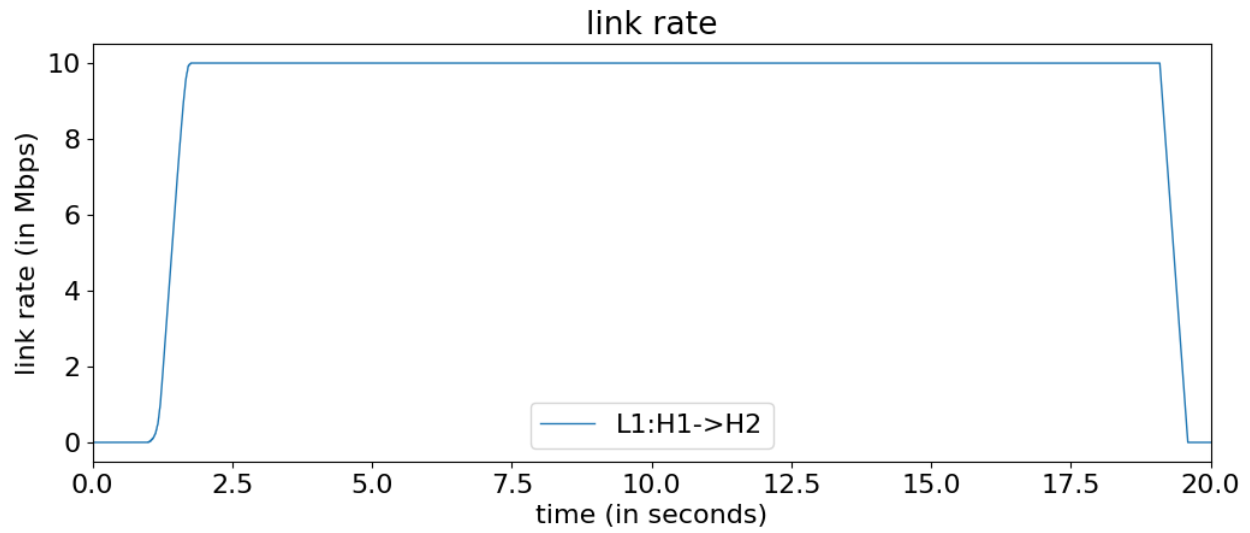
7.2   Test Case 1

7.2.1   TCP Reno

buffer occupancy

flow rate

flow RTT

link rate

packet loss

window size

7.2.2 TCP Fast

buffer occupancy
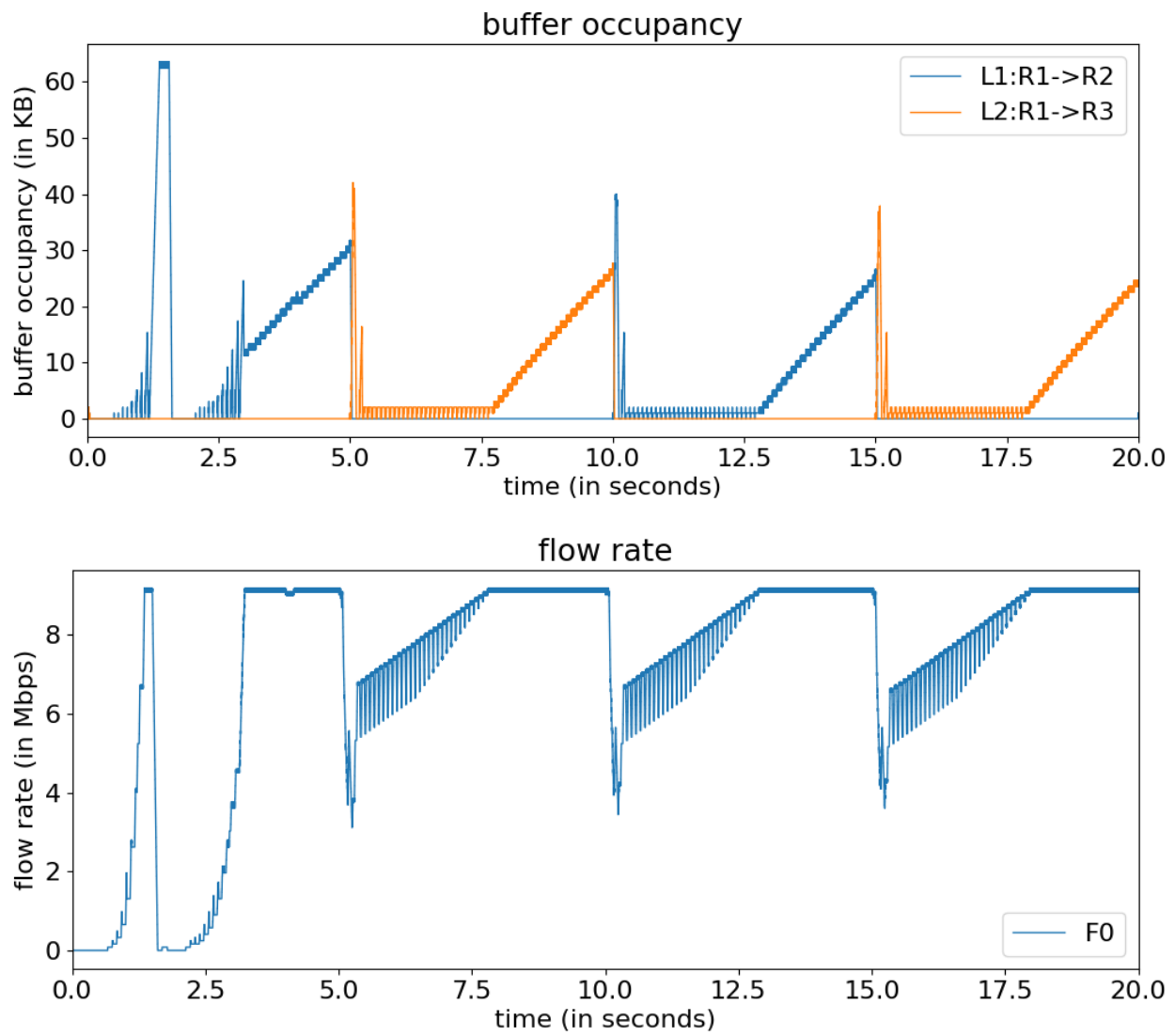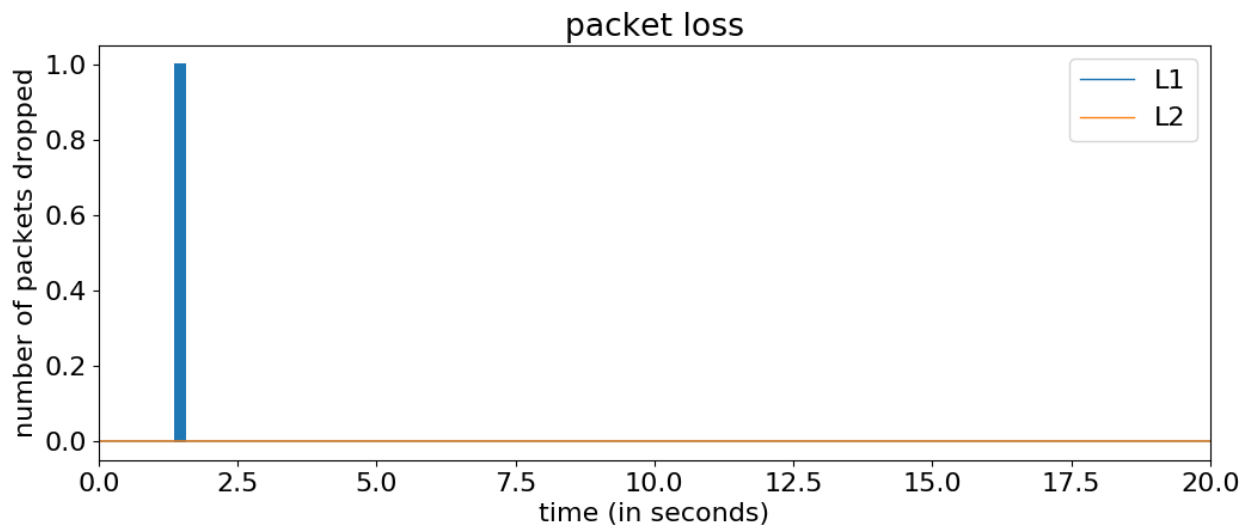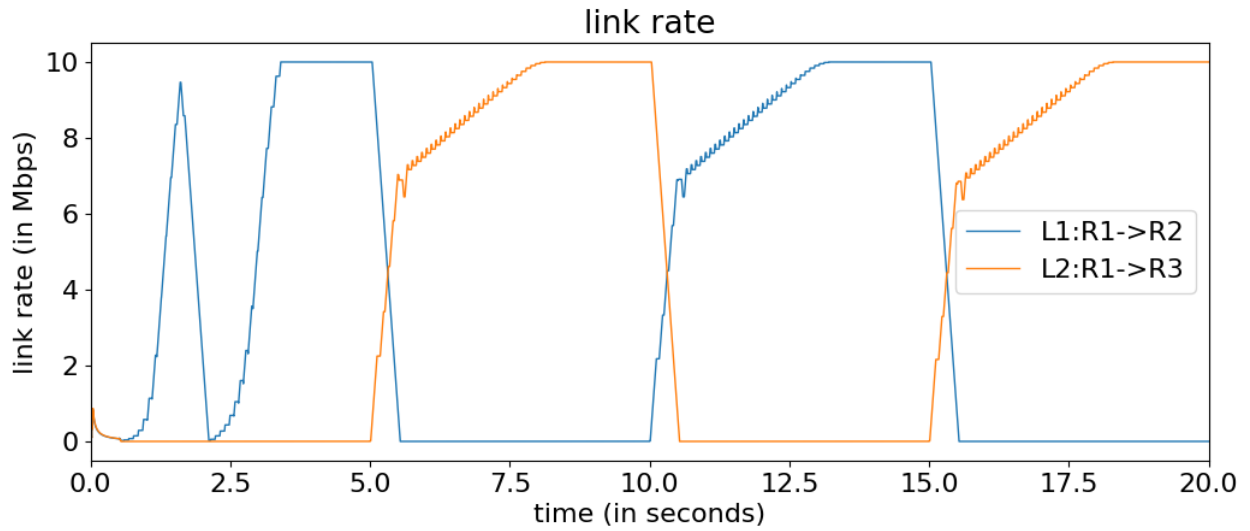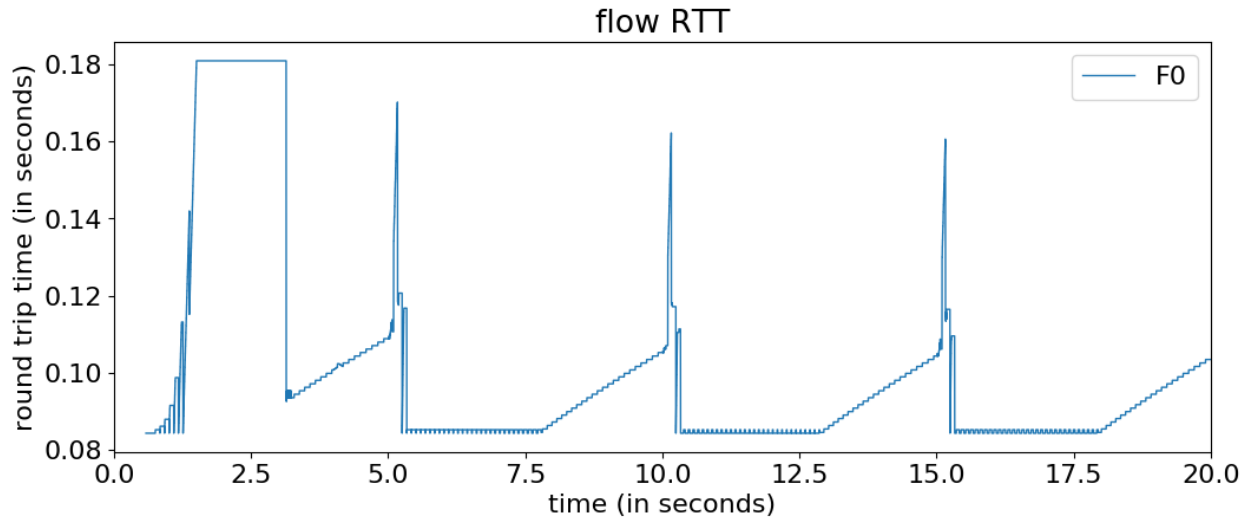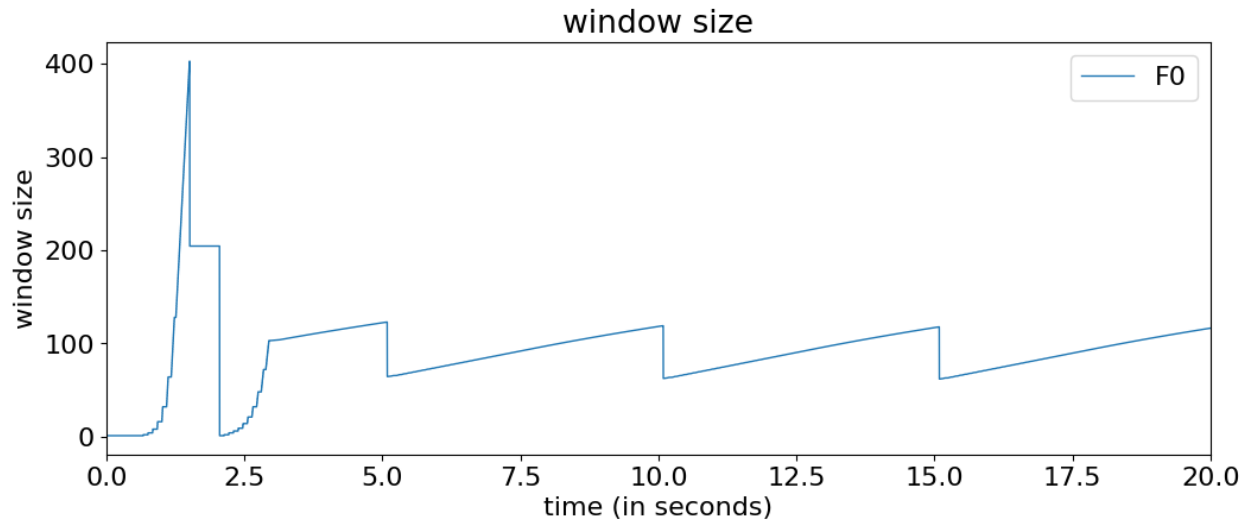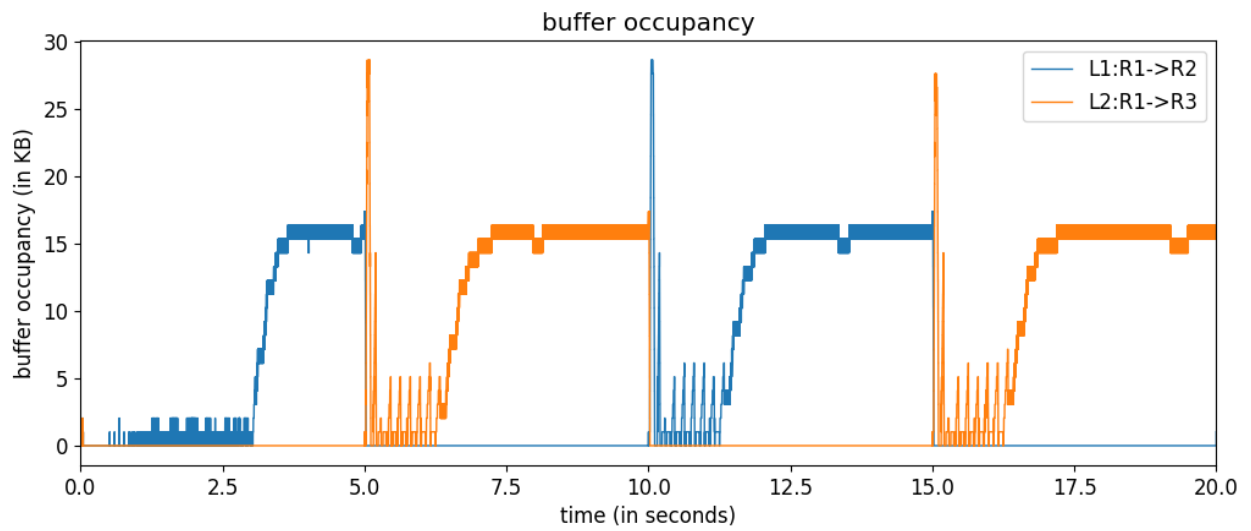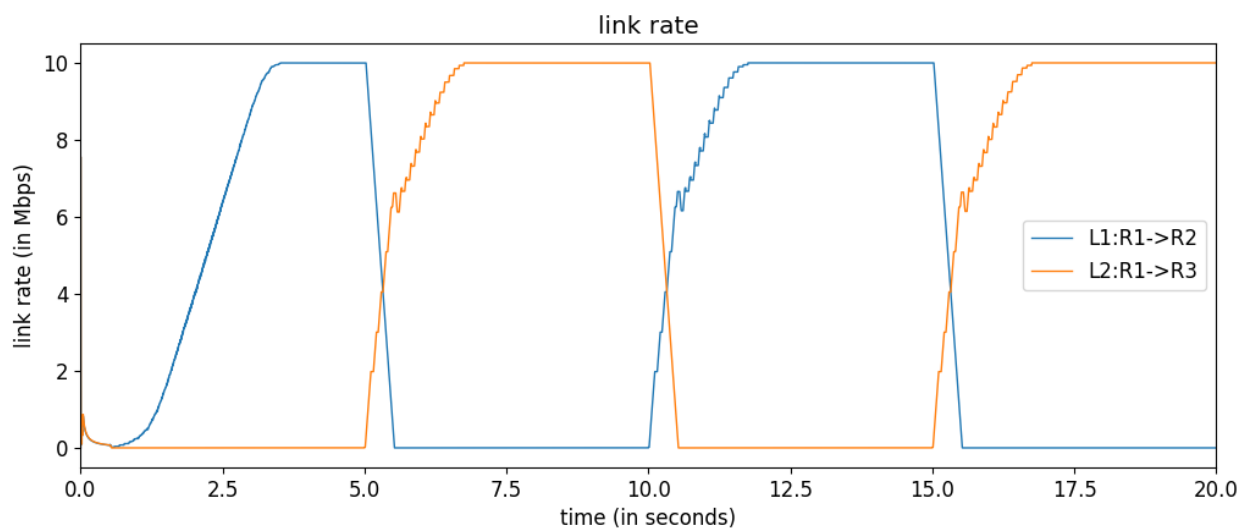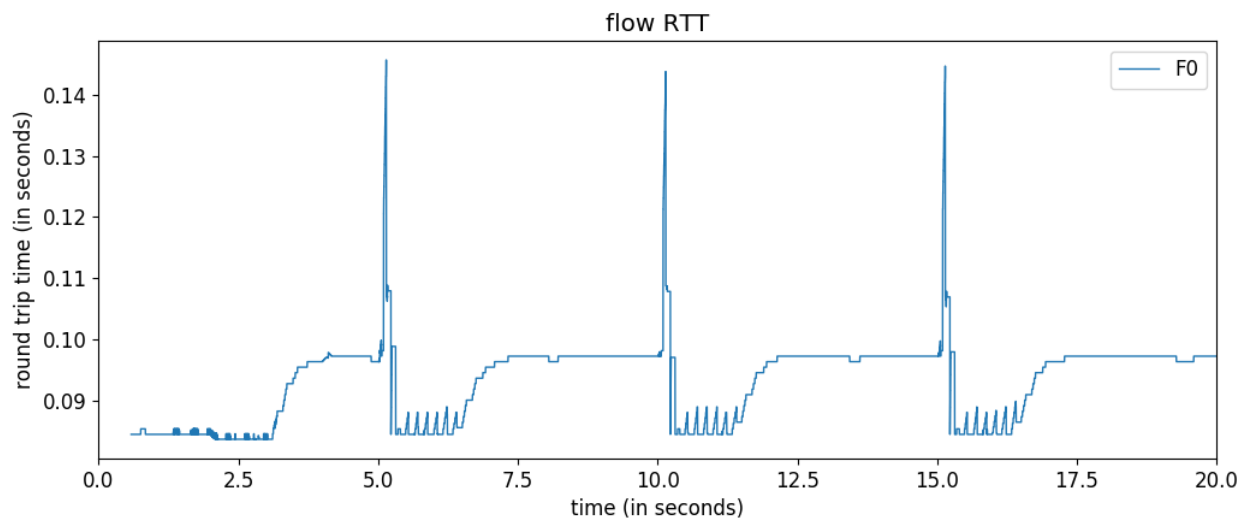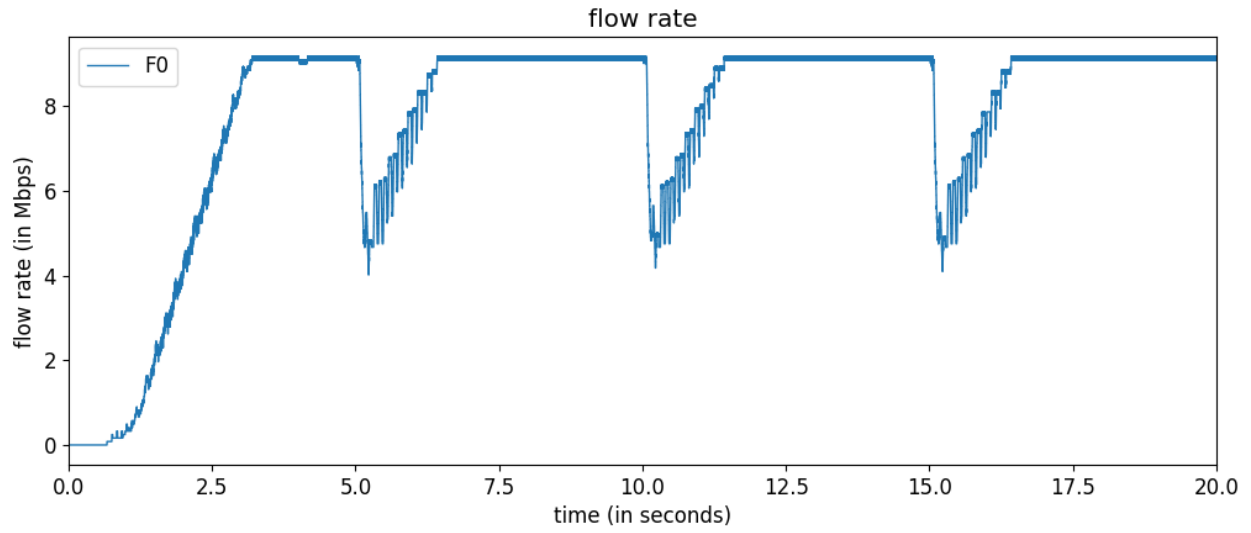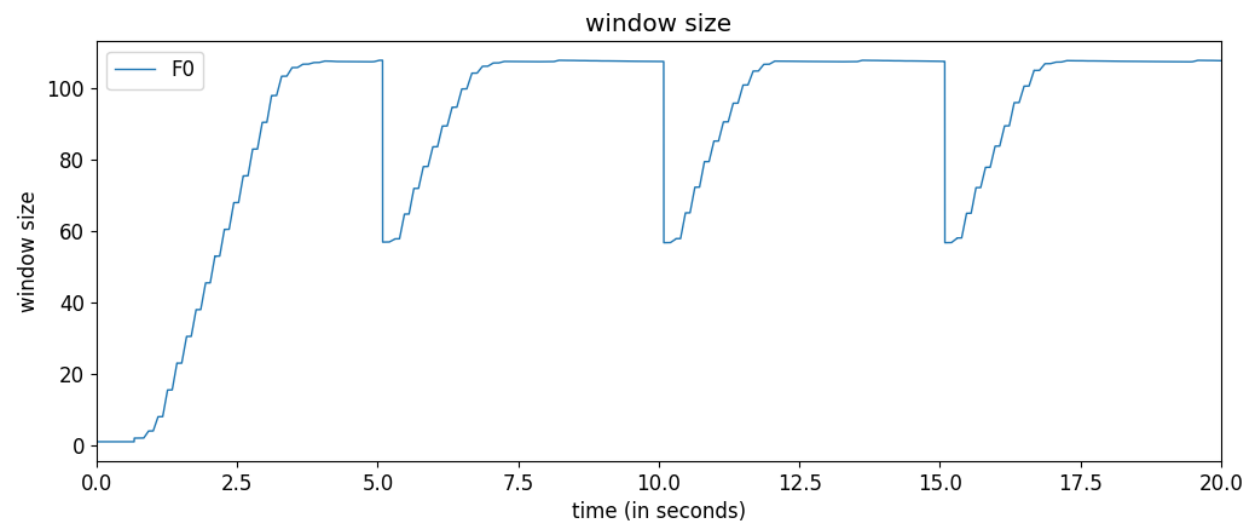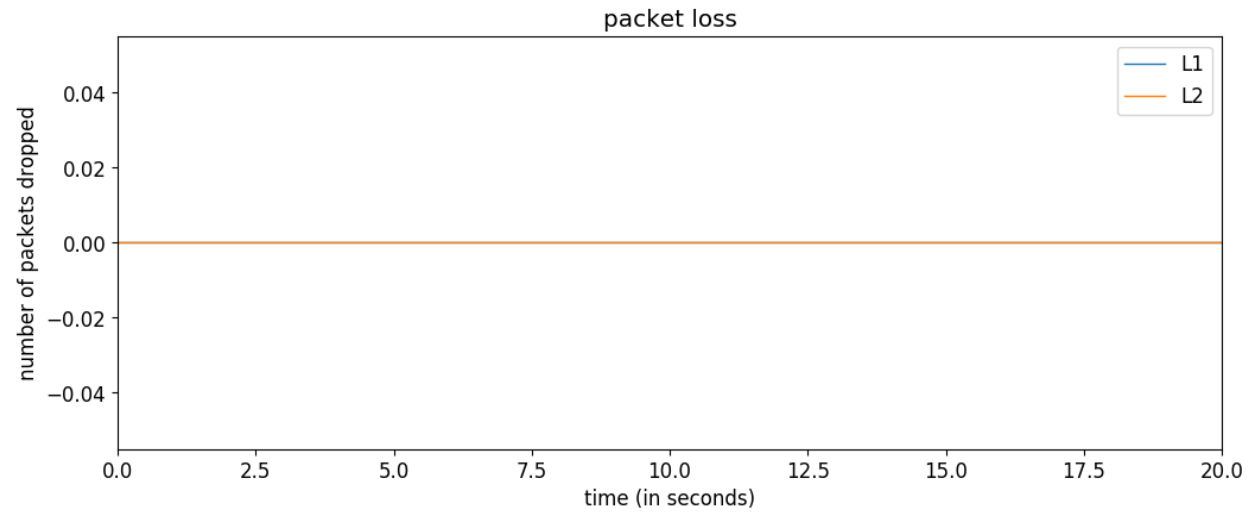
7.3 Test Case 2

7.3.1 TCP Reno

buffer occupancy
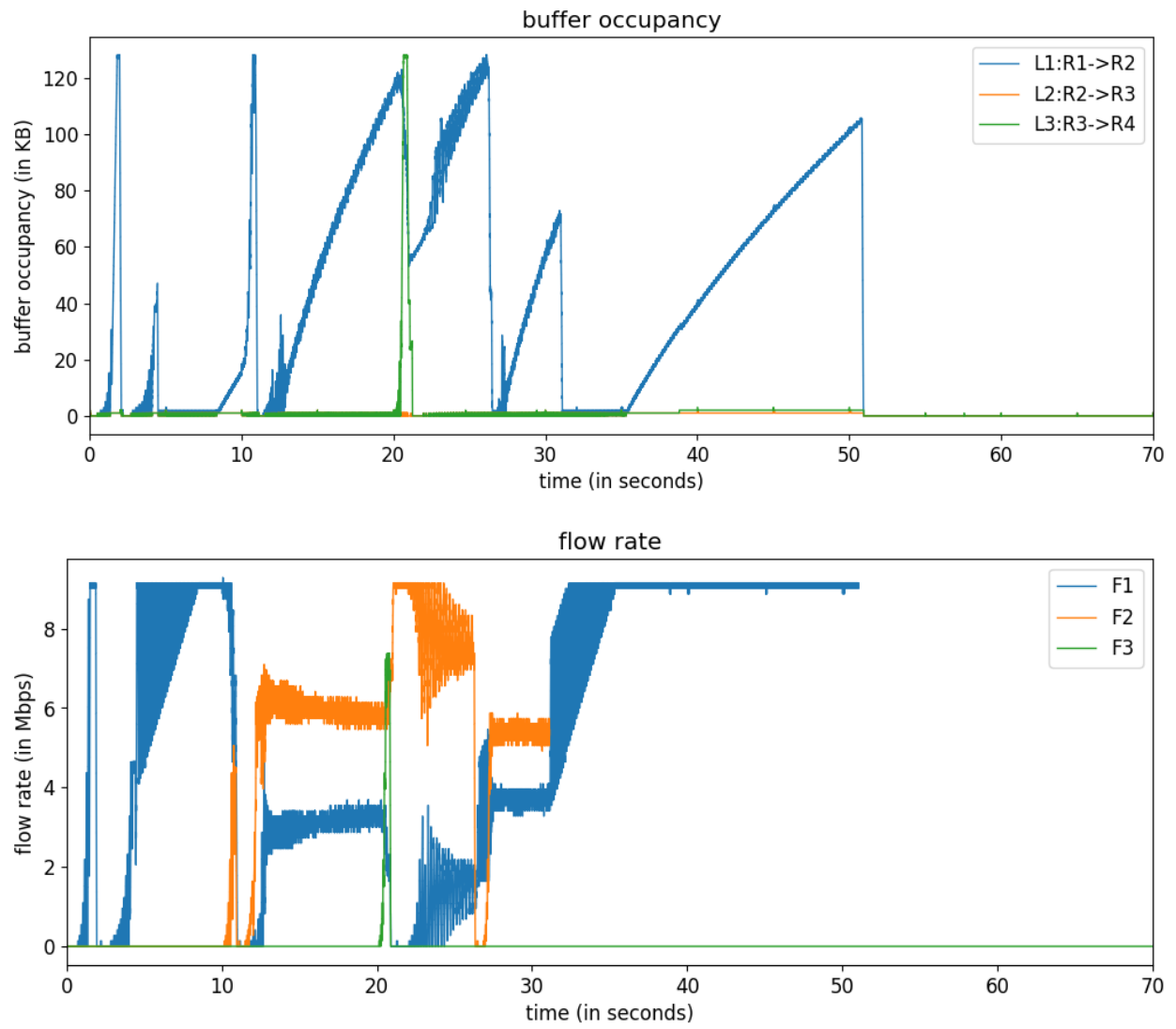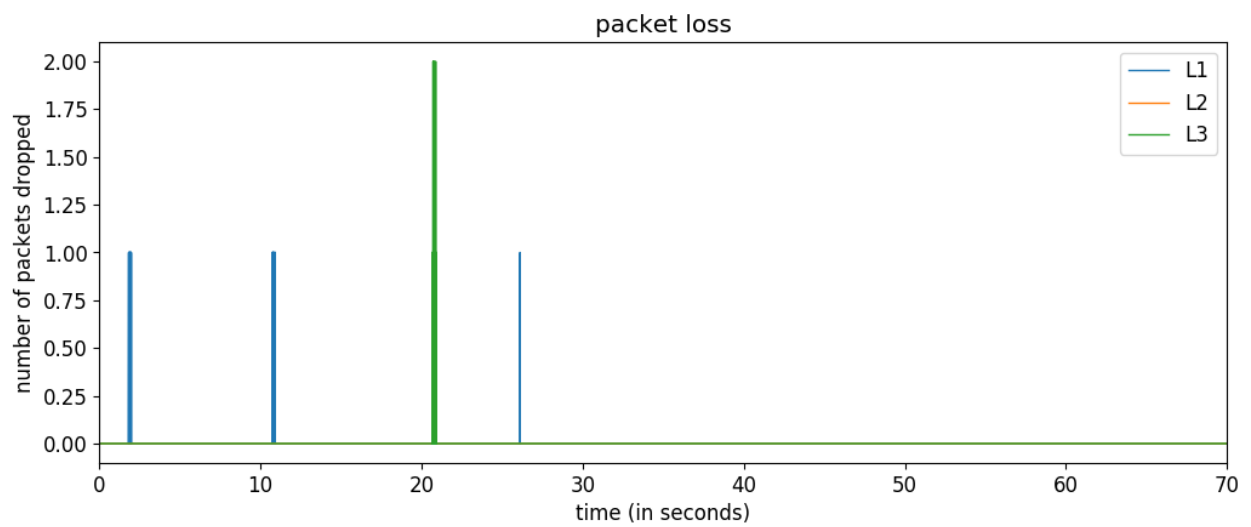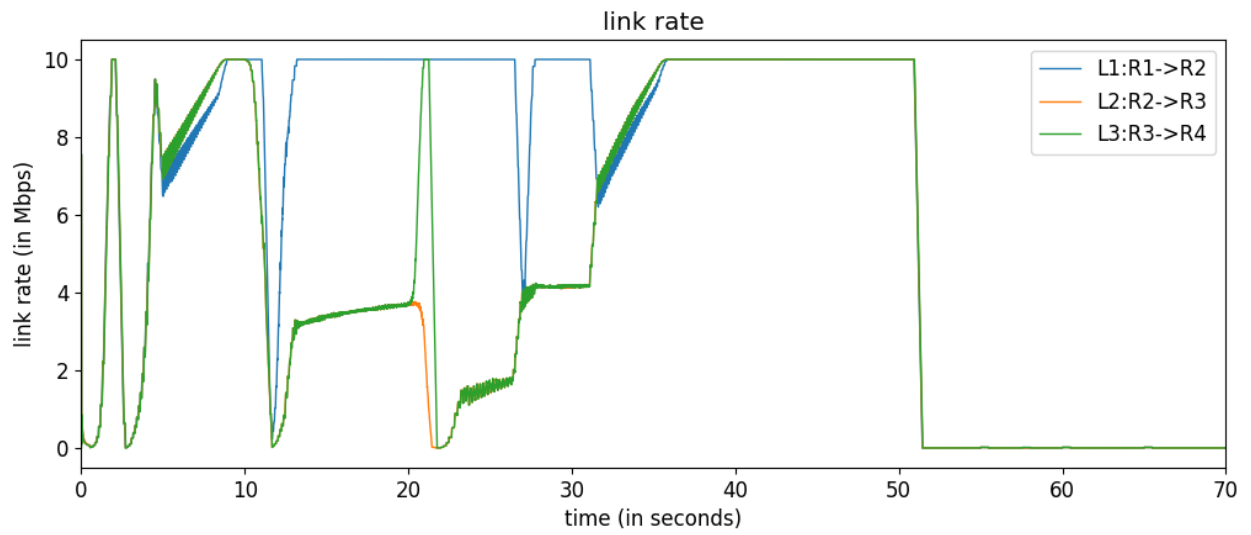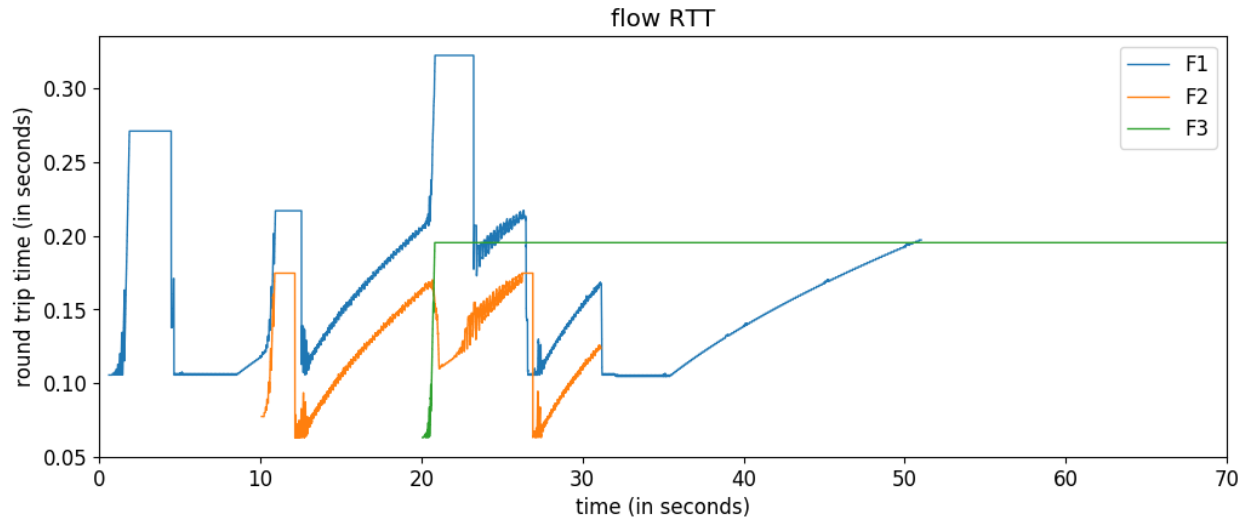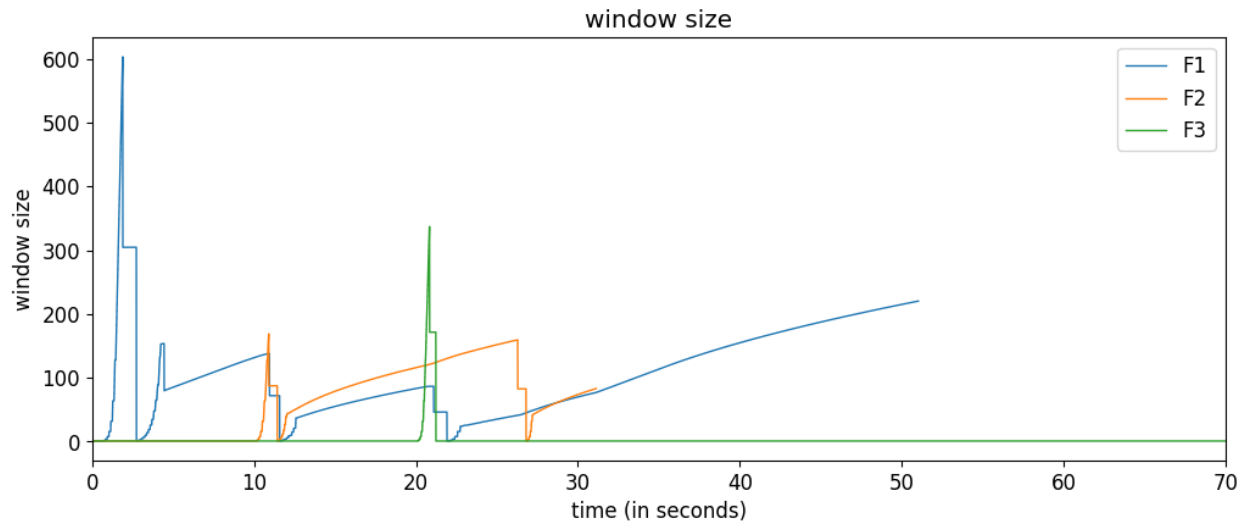
flow rate

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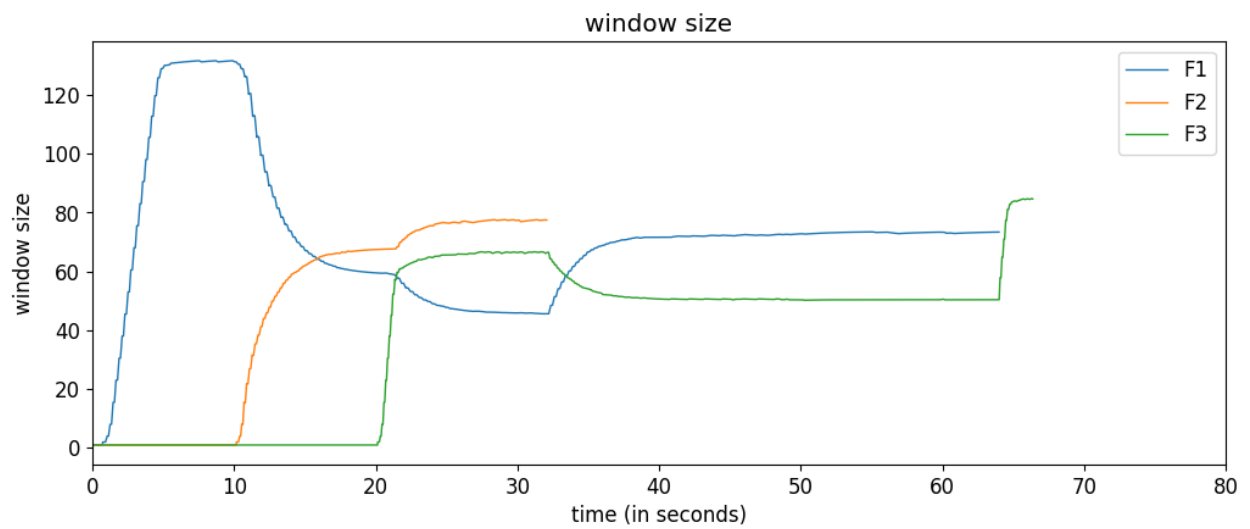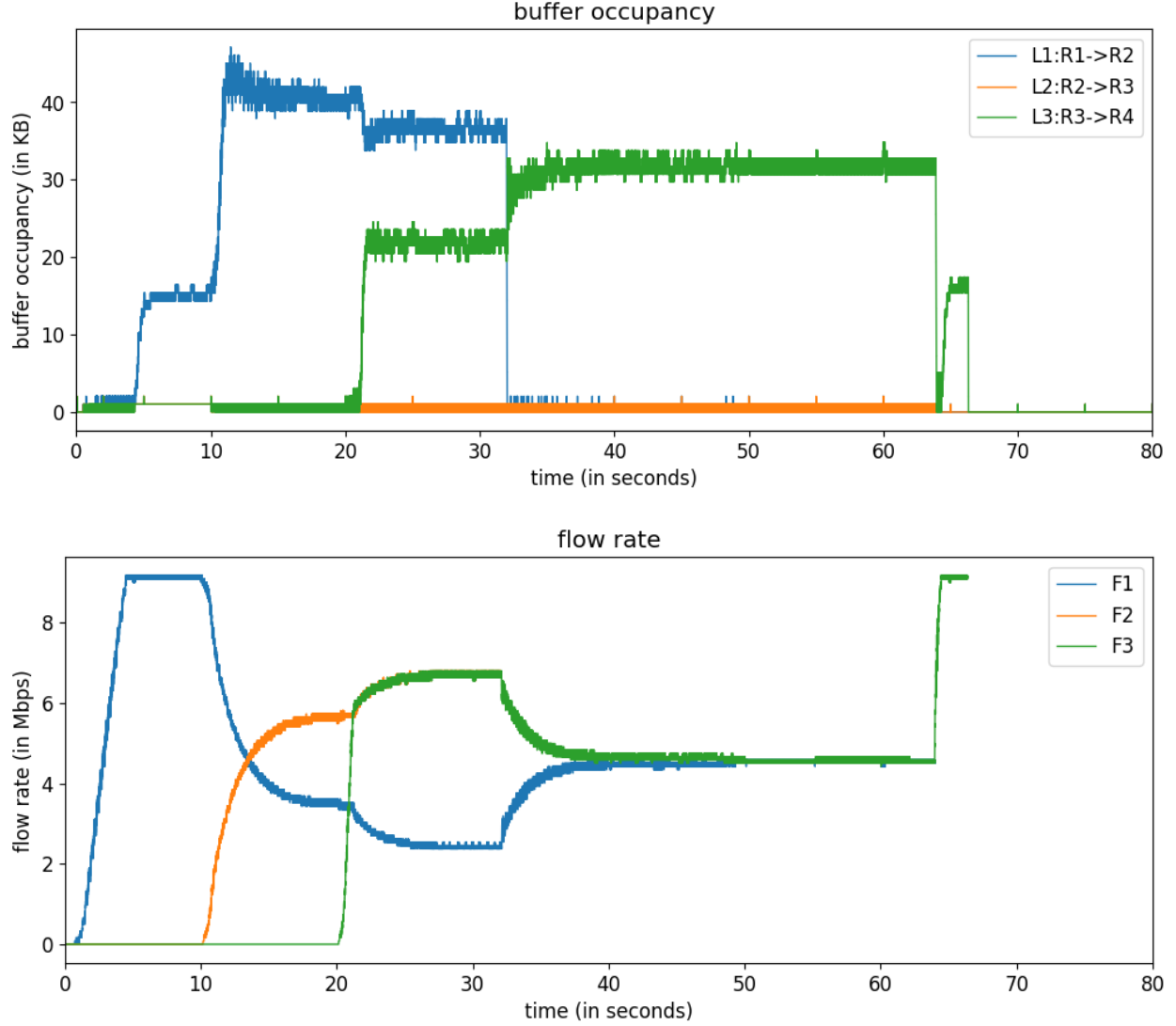
Note that in the flow rate for fast, we have unexplained behavior in which flow 2 has a larger rate than flow 1, and has the same flow rate as flow 3.

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.
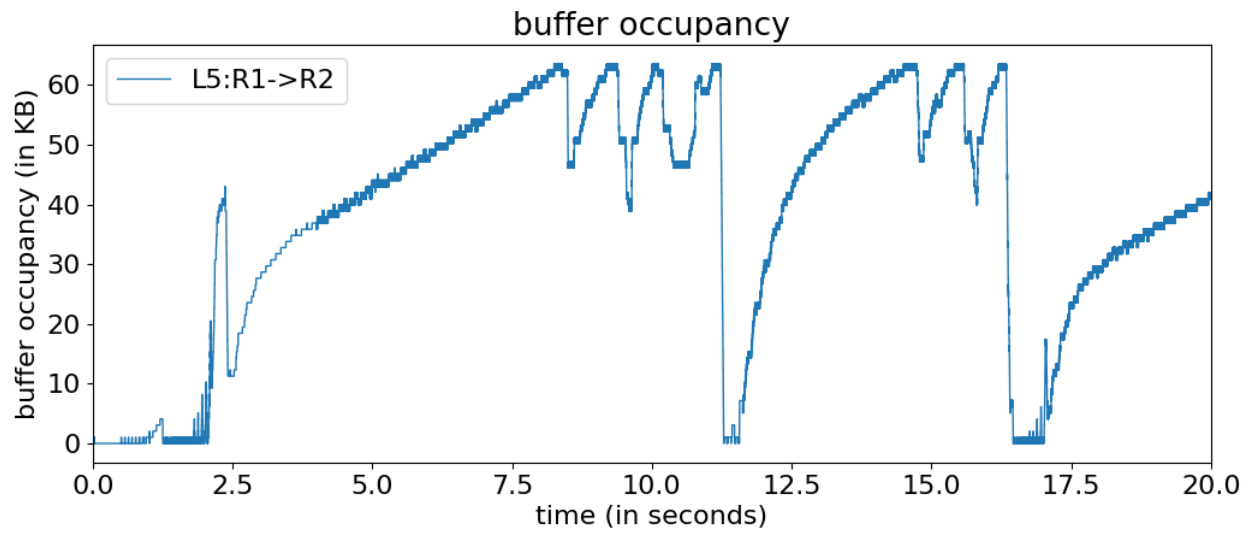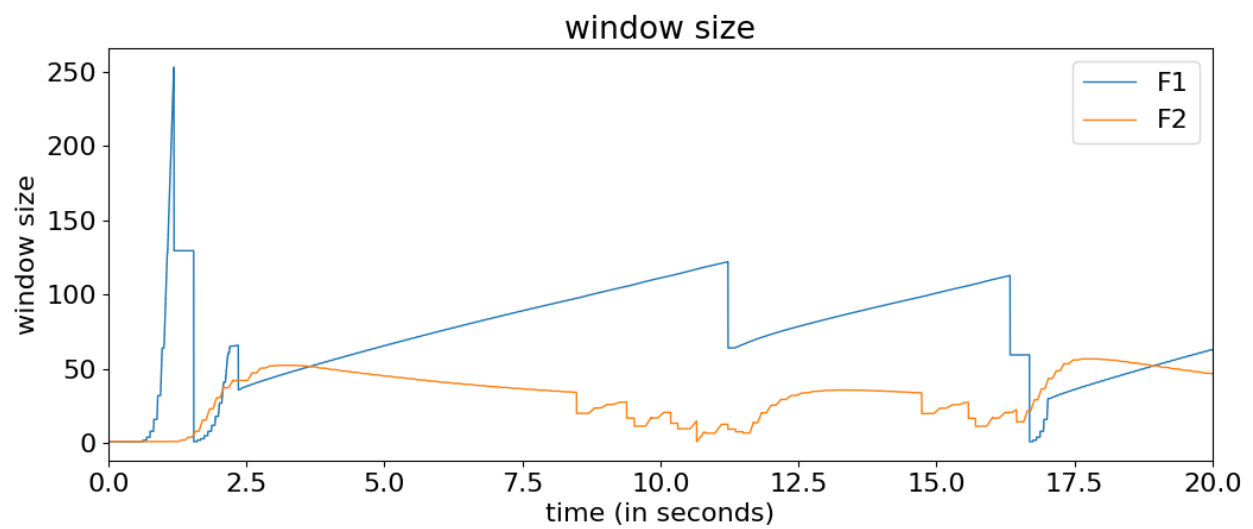
7.4 Test Case 3

buffer occupancy

flow rate

flow RTT

7.5 Test Case 4

buffer occupancy
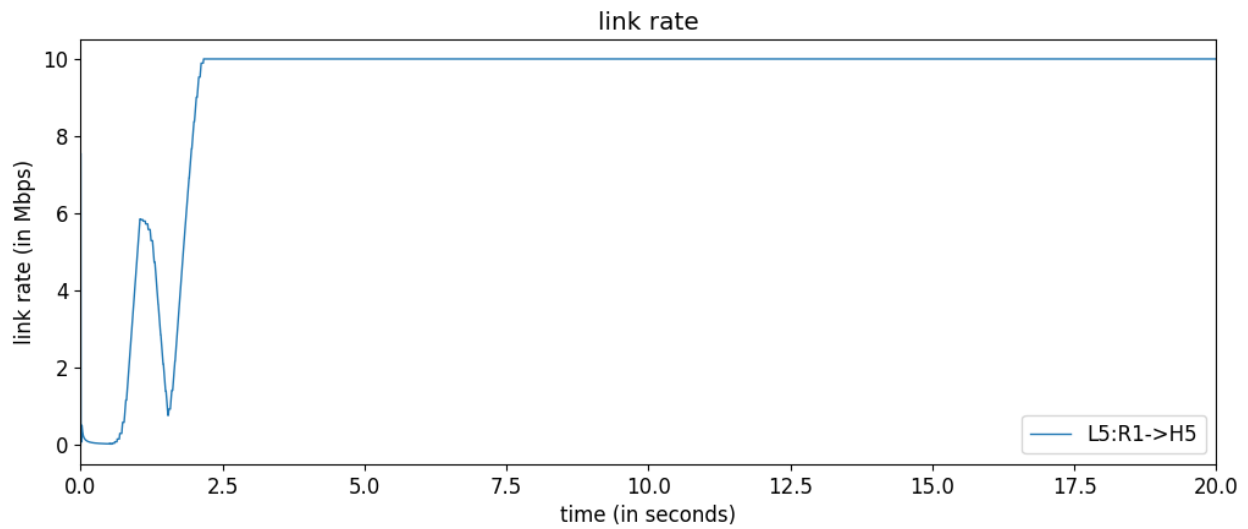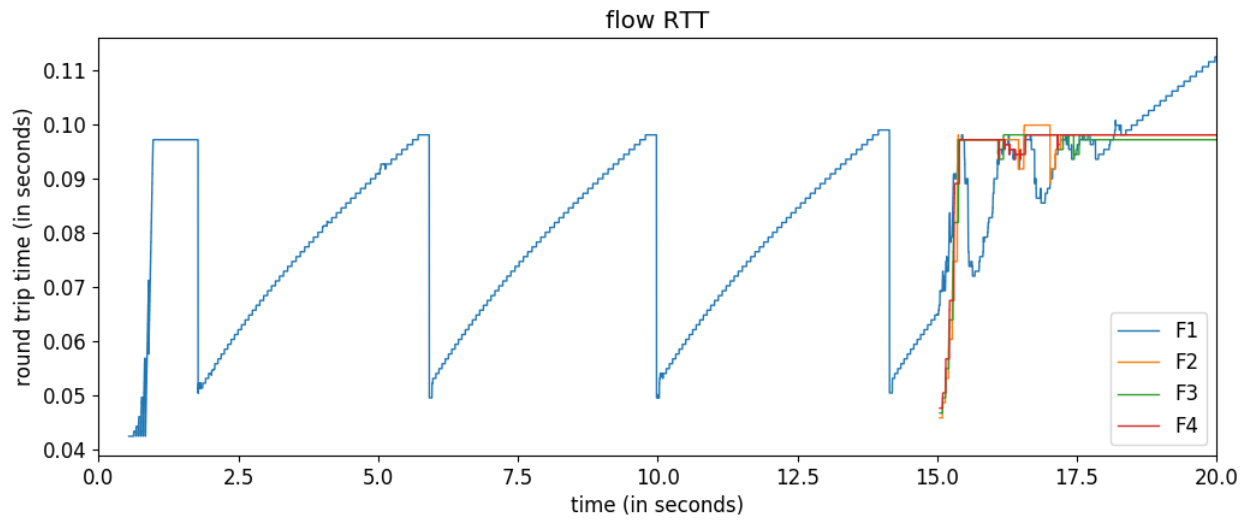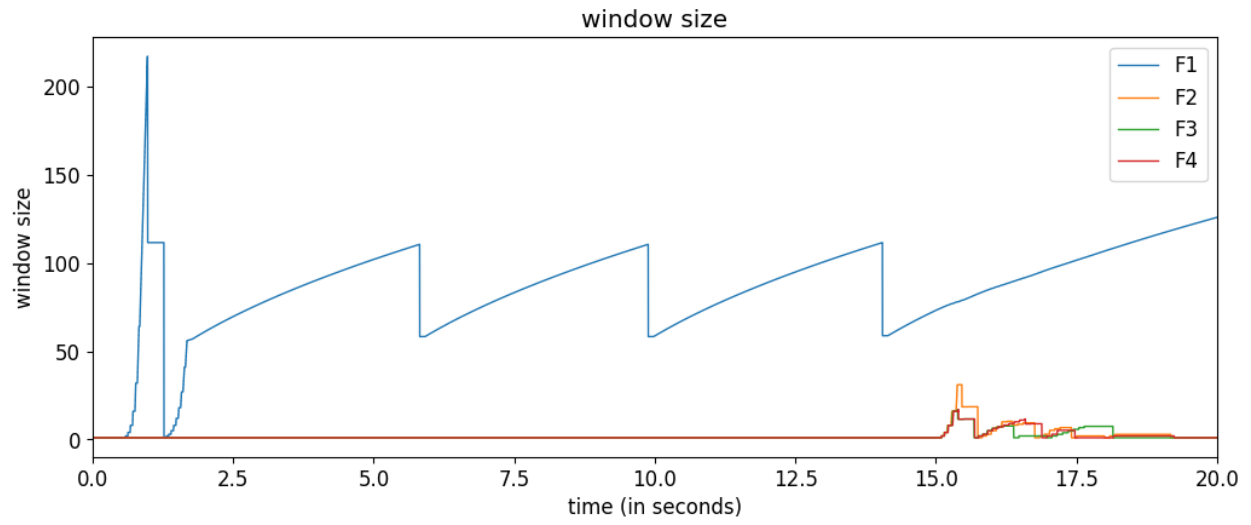

flow rate

flow RTT

link rate

packet loss

window size
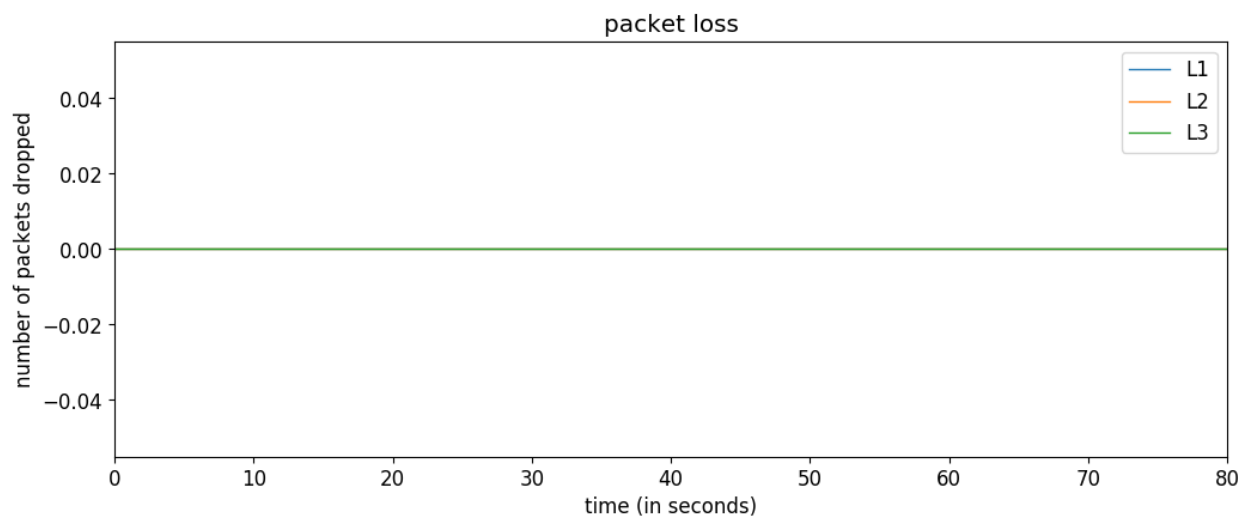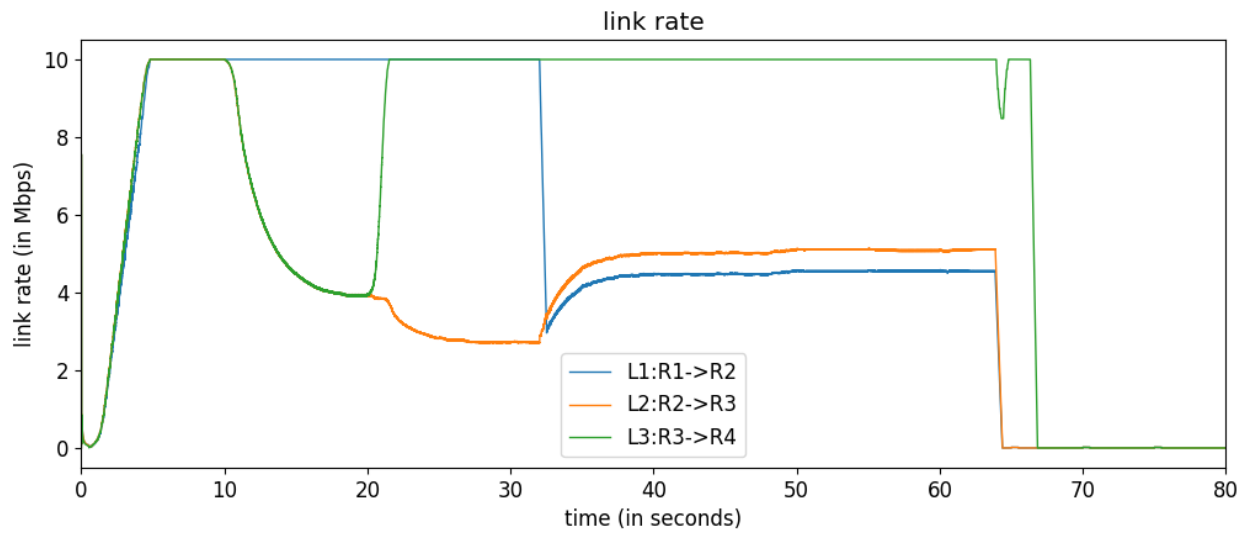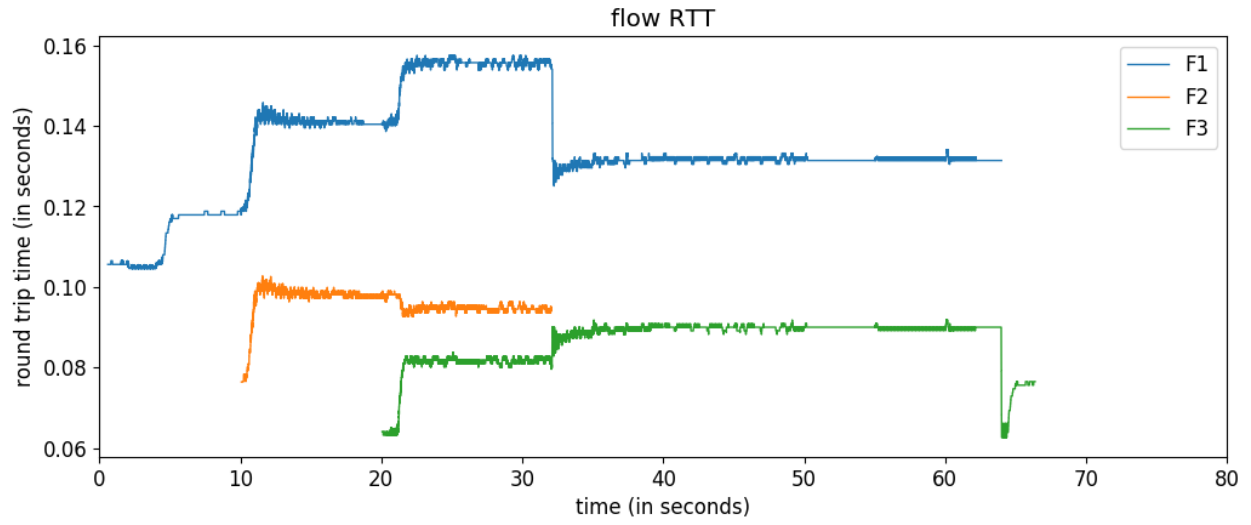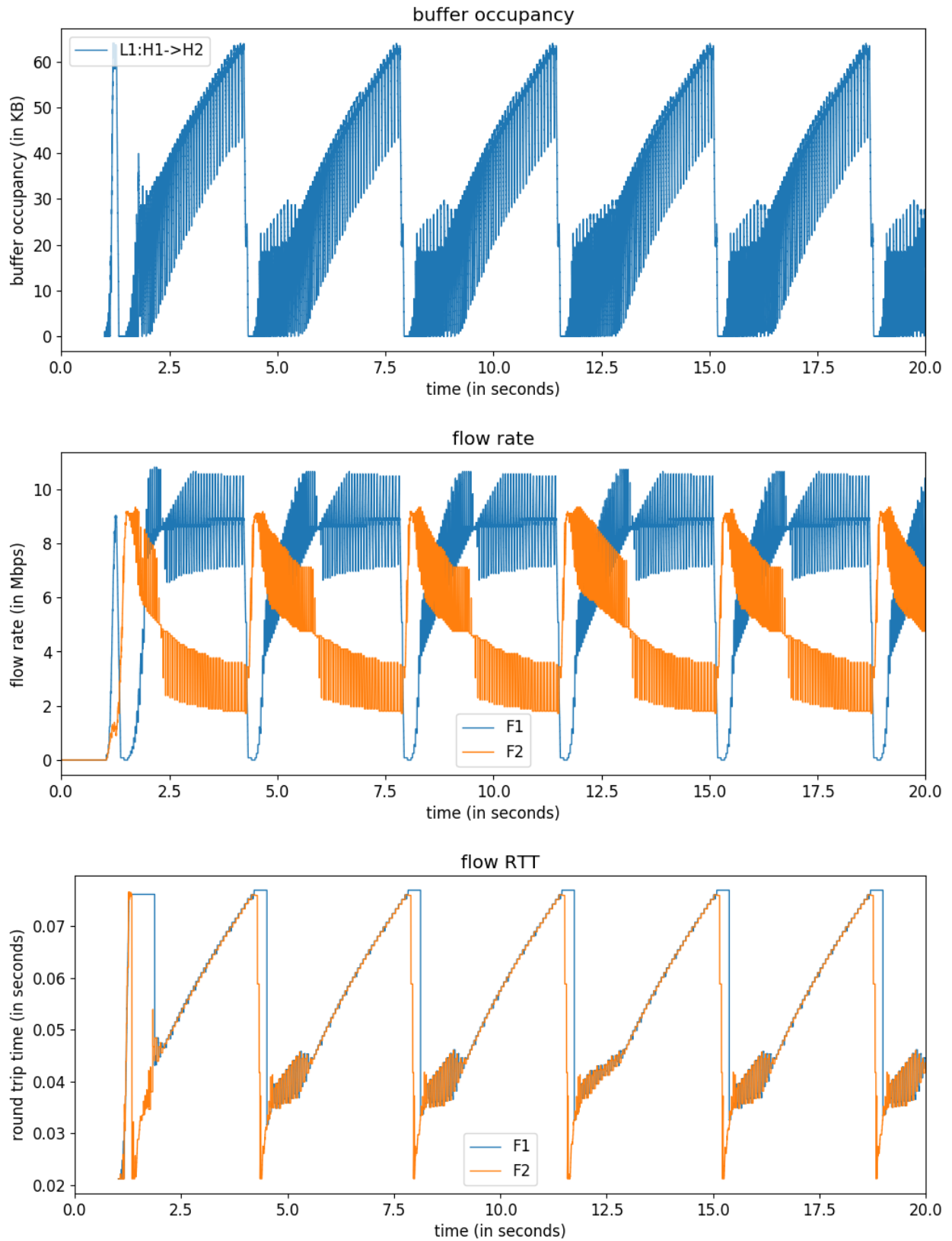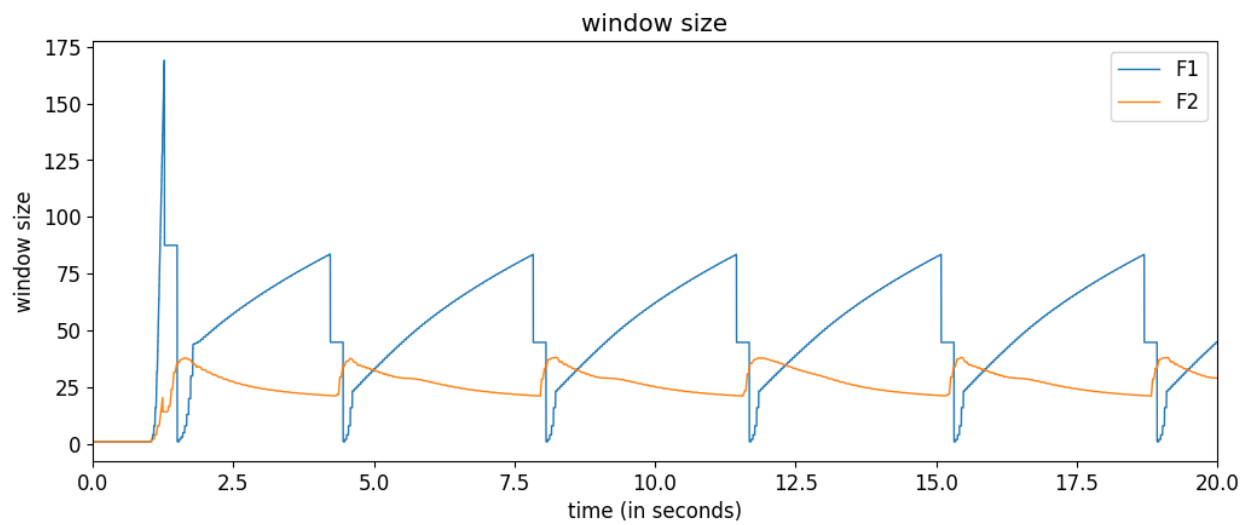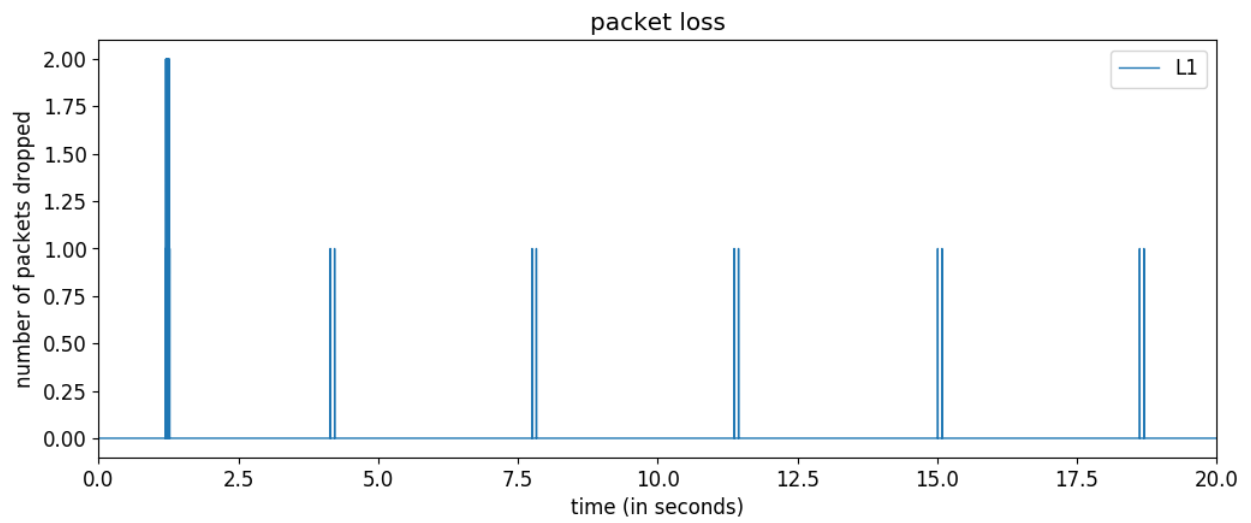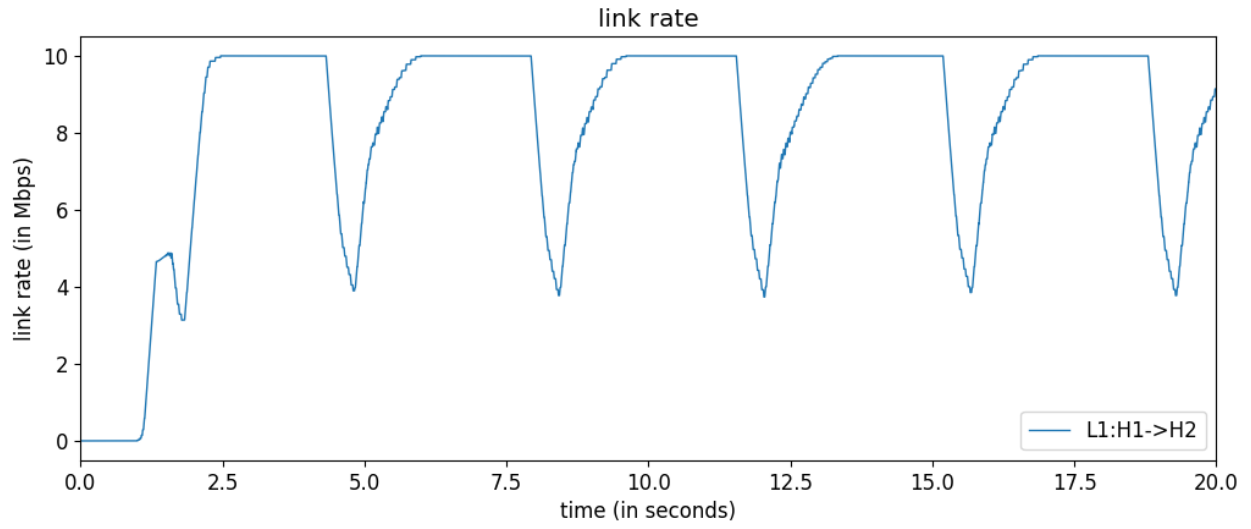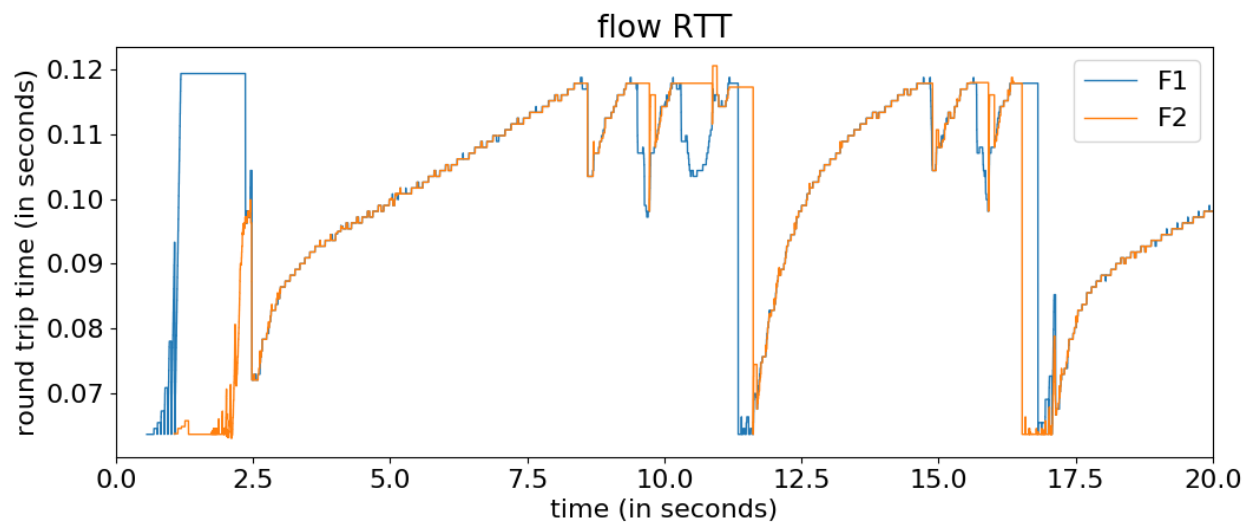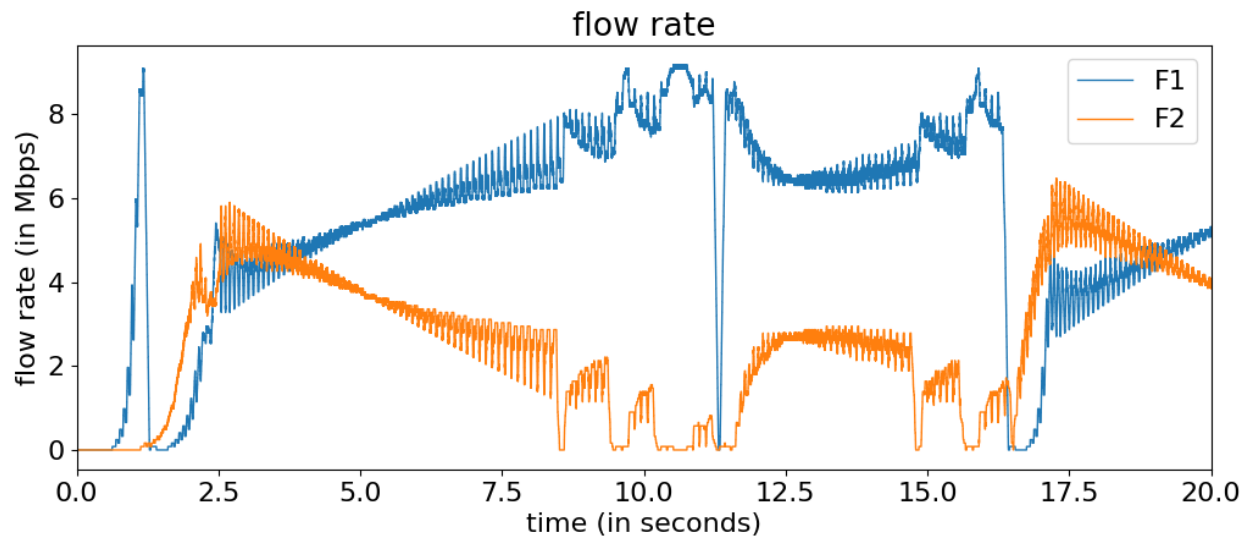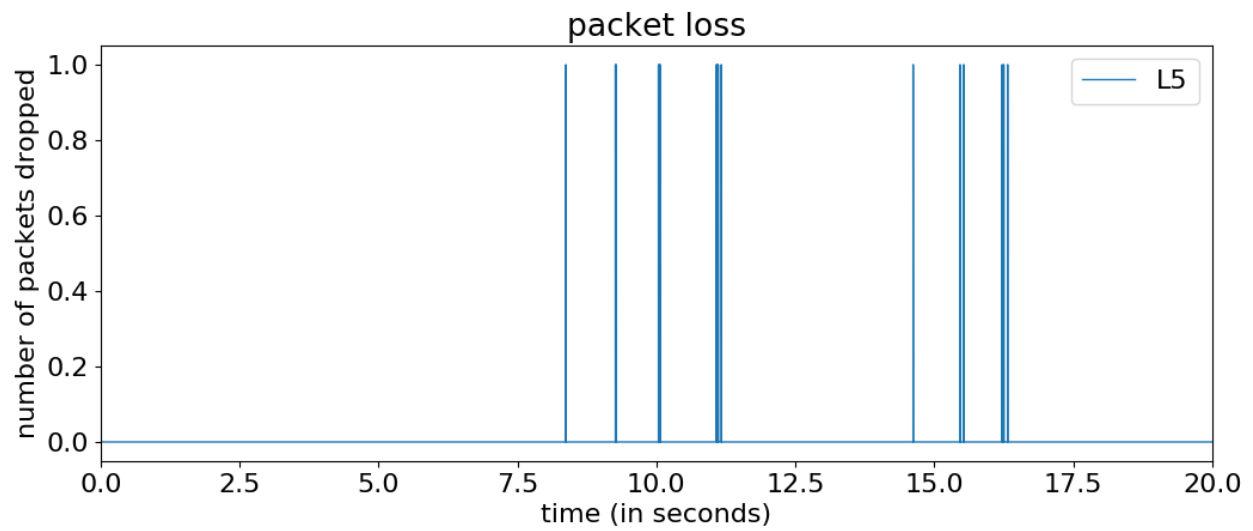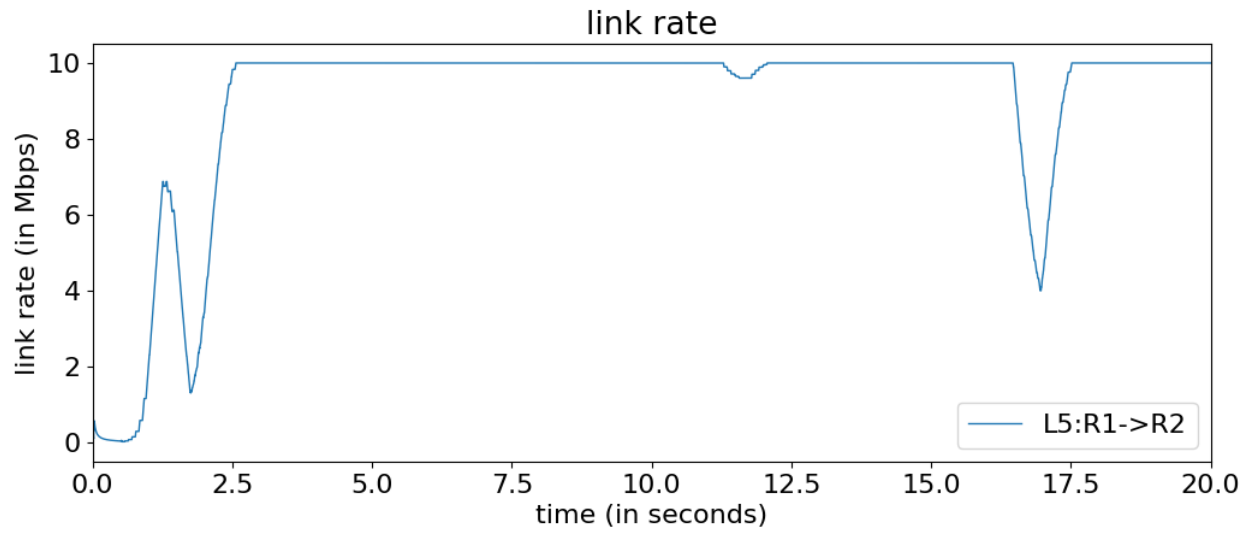
7.6   Test Case 5

7.6.1   TCP Reno

buffer occupancy

flow rate

flow RTT

link rate

packet loss

window size

7.6.2 TCP Fast

buffer occupancy

7.7 Our Test Case

buffer occupancy

legend:
- L1:H1->R1
- L2:H2->R1
- L3:R1->H3

buffer occupancy (in KB) vs time (in seconds)

flow rate

legend:
- F1
- F2

flow rate (in Mbps) vs time (in seconds)

flow RTT

legend:
- F1
- F2

round trip time (in seconds) vs time (in seconds)
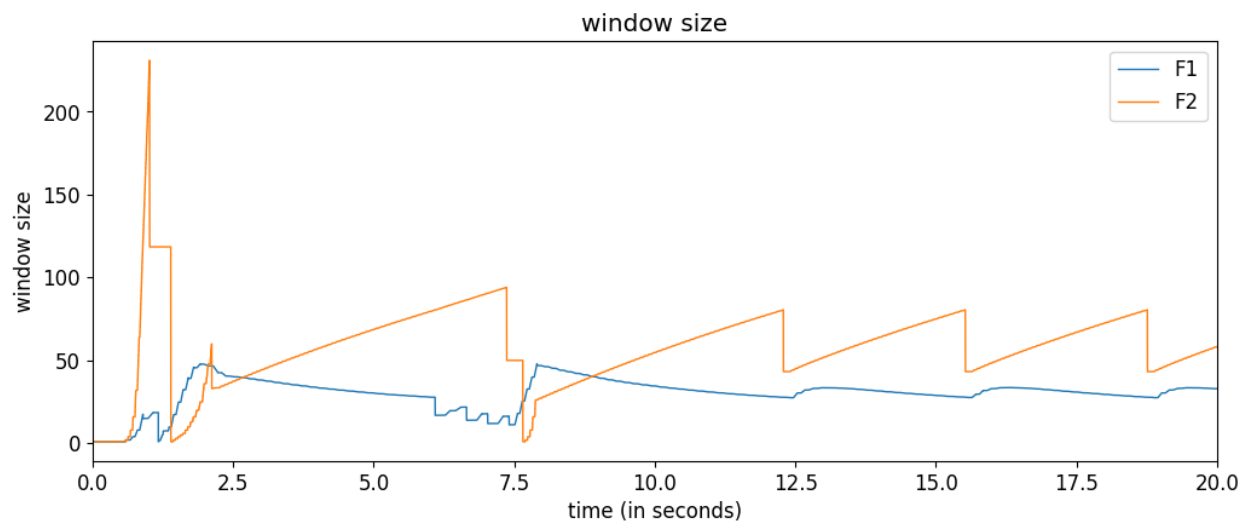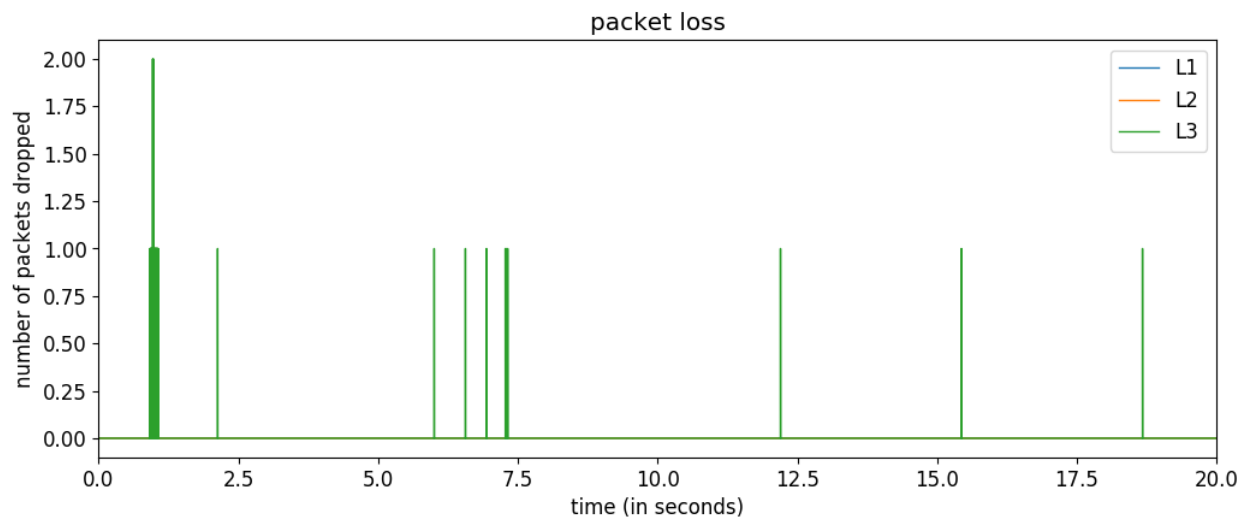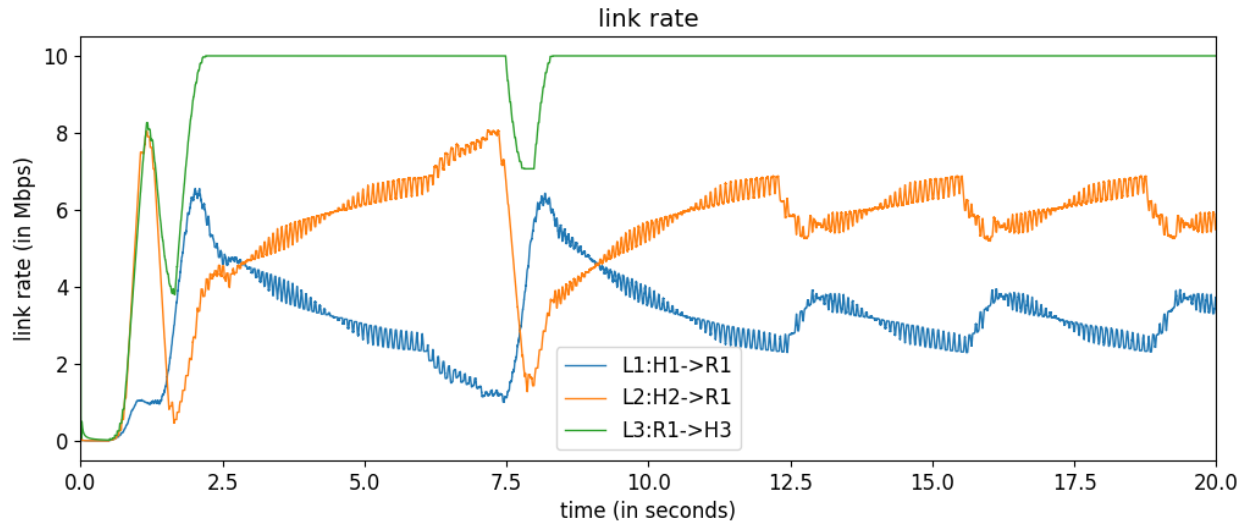
29
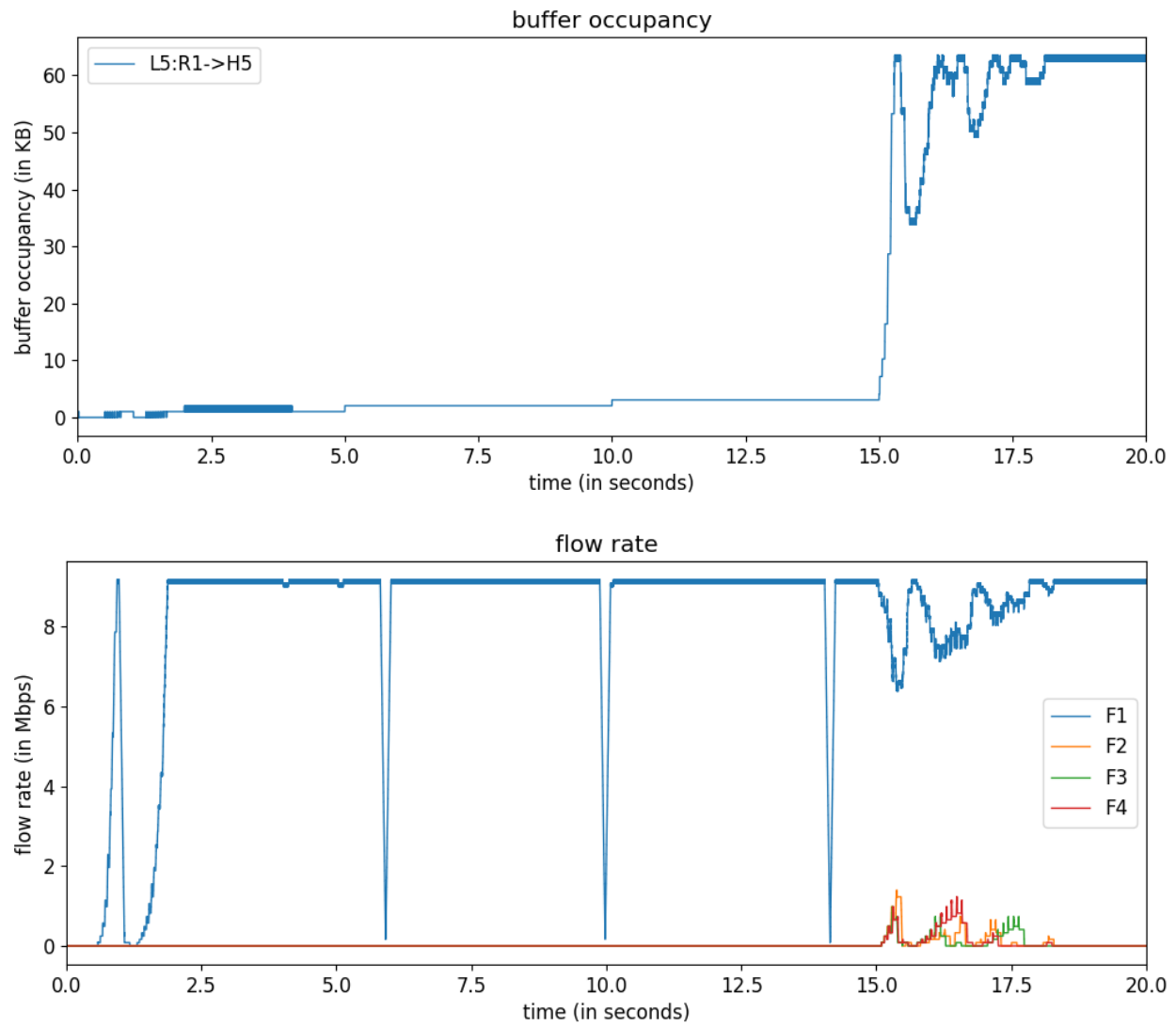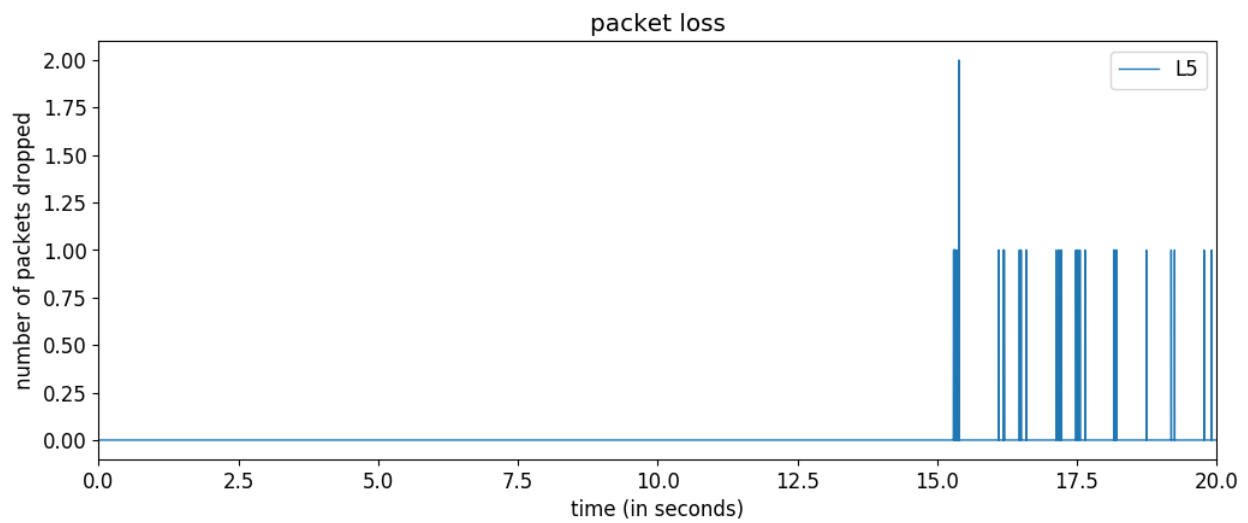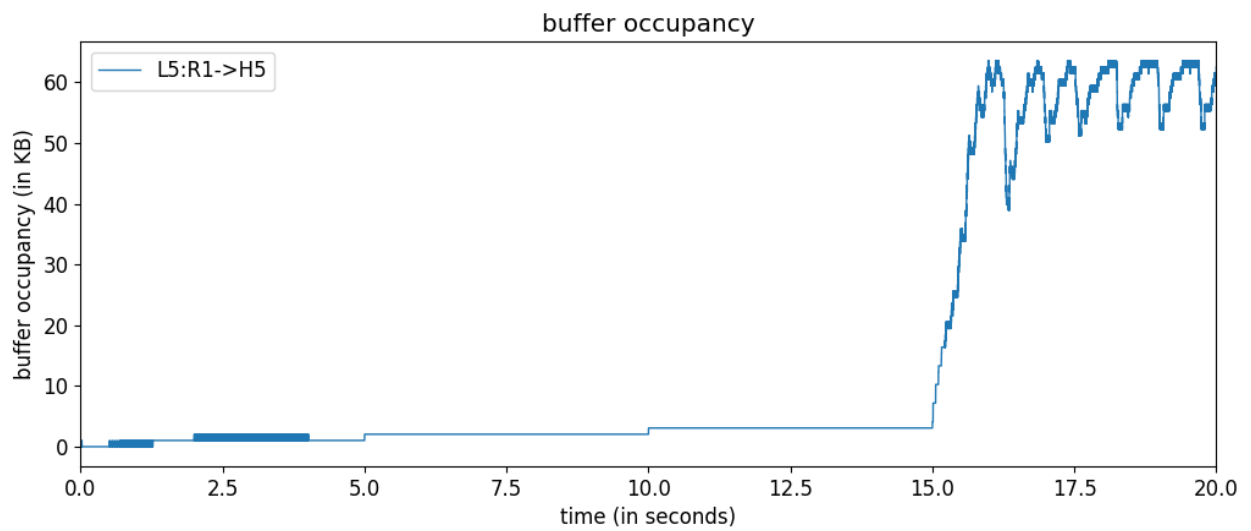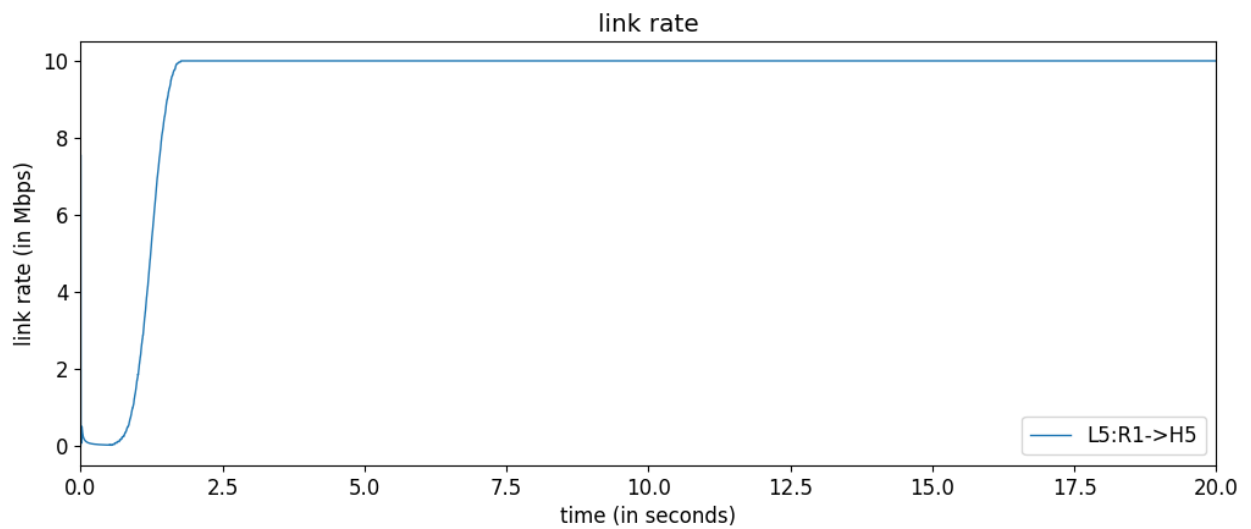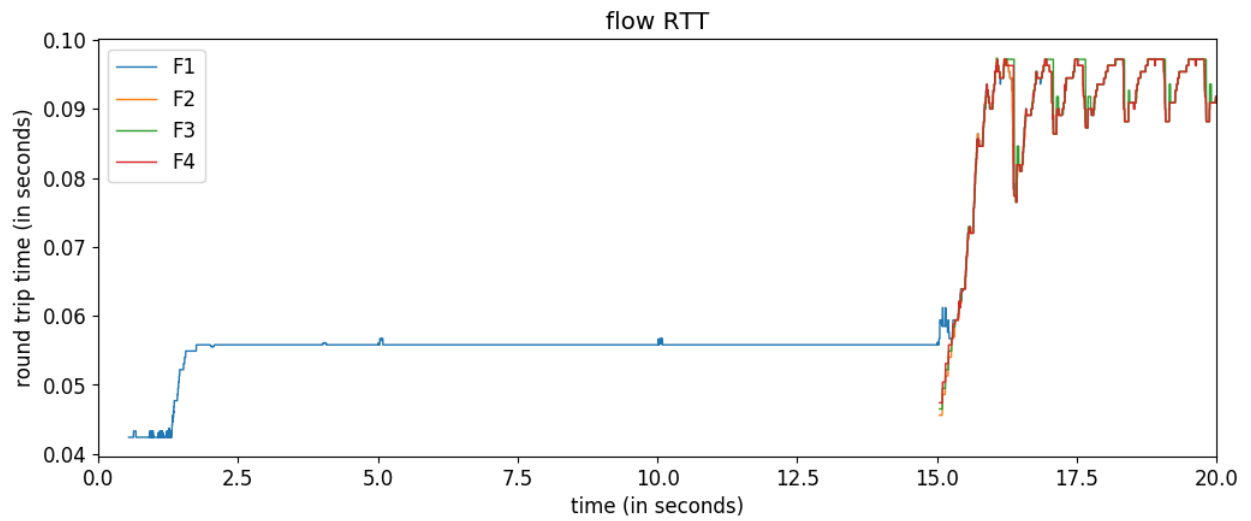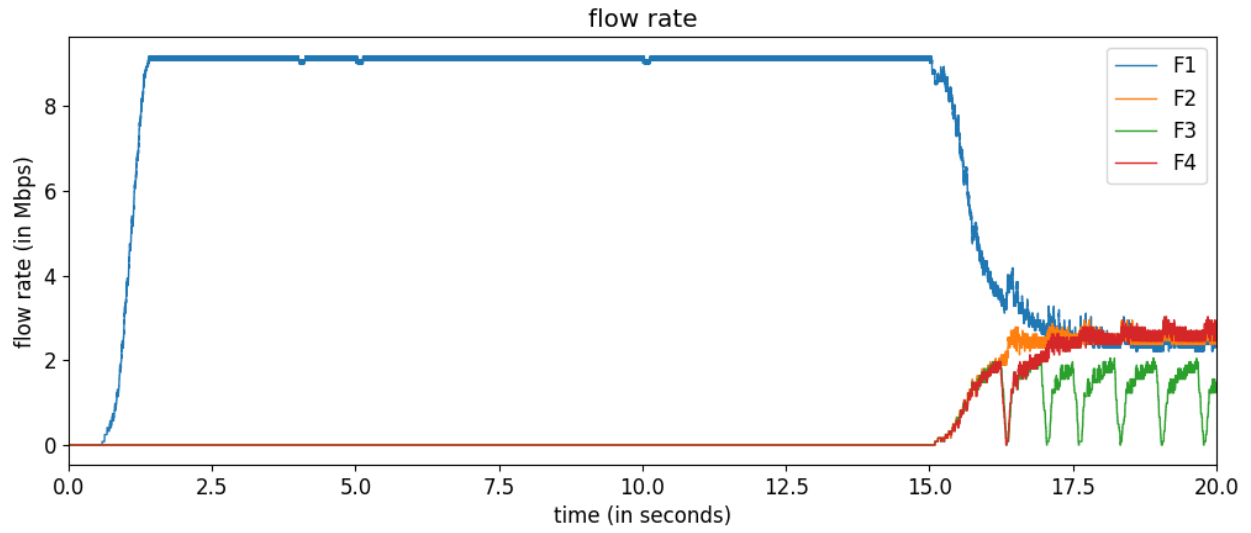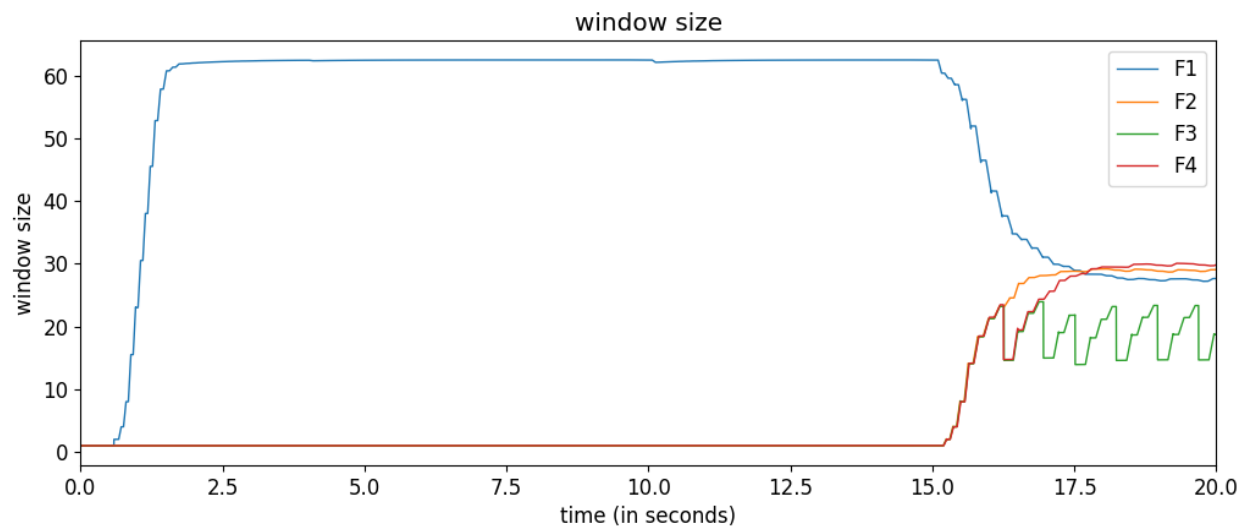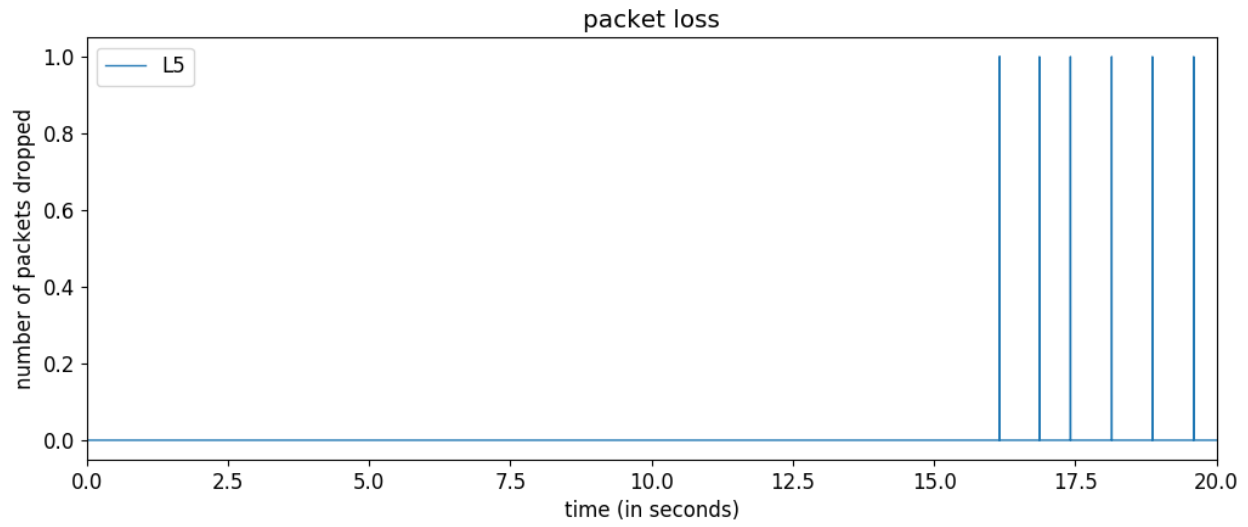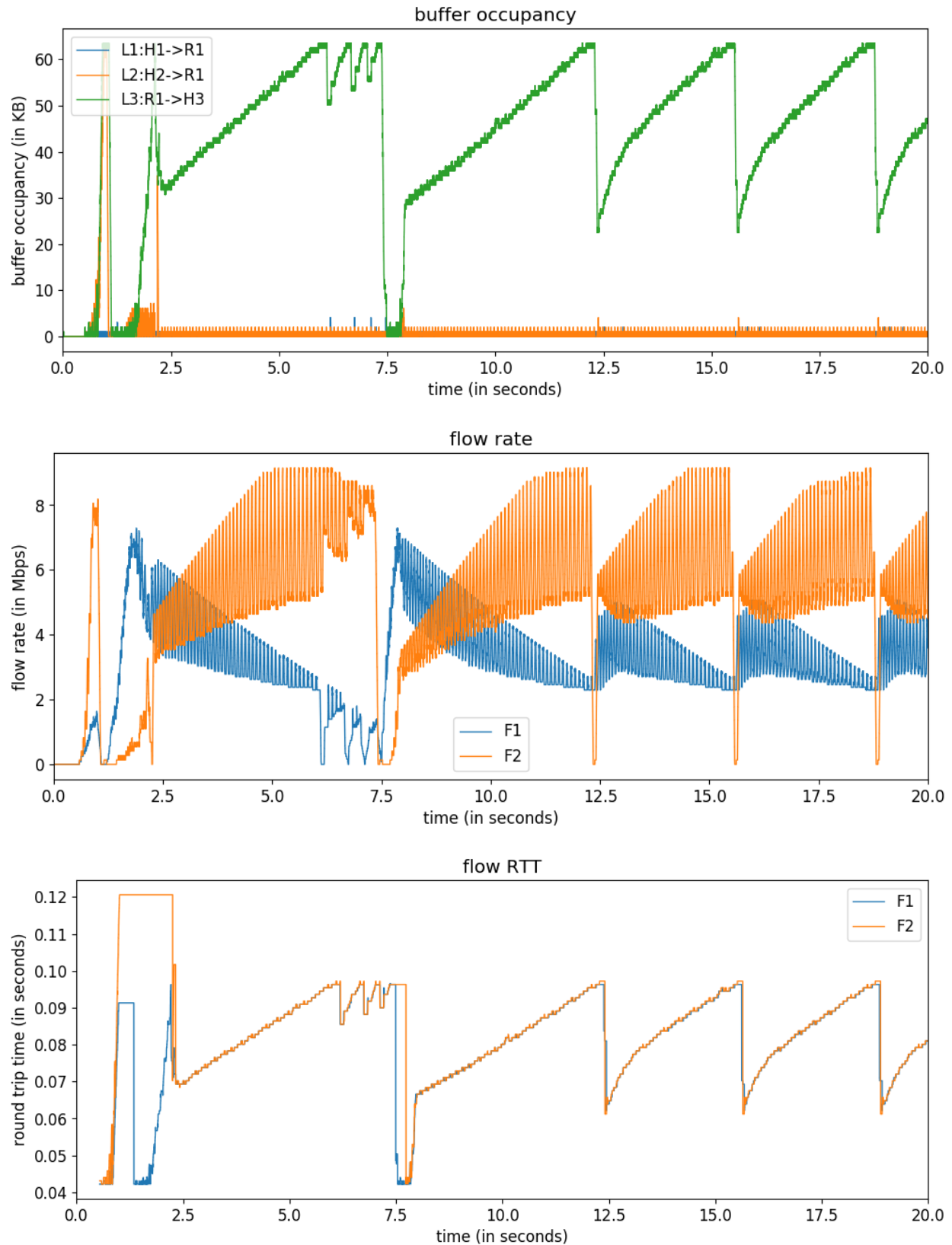
8 Timeline & Division of Labor

8.1 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

8.2 Division of Labor

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