BlackWidow TCP Network Simulator

Rahul Bachal, Justin Leong, David Qu, and Nancy Wen

Abstract—Transmission Control Protocols (TCPs) are distributed congestion control algorithms that prevent internet communications from having too much interfering traffic. For our project, we created ”abstract network simulator” that models the transmission of packets through a specified network including hosts, routers, flows and links.

Index Terms—Transmission Control Protocol (TCP), Distributed Algorithms, Abstract Network Simulator

1 Introduction

OUR project is an ”abstract network simulator” that models the transmission of packets through a specified network including hosts, routers, flows and links. Our project is able to successfully simulate the three provided test cases and produce plots for each simulation.

1.1 Tools and Methods

We use the Python 2.7 programming languages to simulate the network. The simulation is discrete event-based. For testing our code, we use the nose package for unit testing. We plot the results of the simulation using the matplotlib library. Finally, we use Sphinx for documentation and Github for version control.

1.2 Overall Program Architecture

The blackwidow module (located in the src folder) includes the parser, network, and graph submodules. It configures and runs a simulation, and then produces graphs for the simulation. For example, to run the module on case 0, type the following in the Python interpreter:

```
>>> from blackwidow
import BlackWidow
>>> bw = BlackWidow()
>>> bw.run('case0.json')
```

We also have a run_simulator.py script that parses the arguments and calls the blackwidow module:

```
>>> python run_simulator.py [-v]
cases/case*.json
```

1.3 Parser

Each test case is represented by a JSON file that has lists of hosts, routers, flows, and links along with associated link rates, link delays, link buffer sizes, flow start times, and flow data amounts.

1.4 Elements of the Network

The elements of the network include the hosts, routers, flows and links. In our simulator, we implement the elements of the network using object oriented programming.

1.4.1 Hosts

In an actual network, the hosts are the individual endpoint computers like, desktop computers or servers. We assume that every host can

• *R. Bachal, J. Leong, and D. Qu are undergraduates studying Computer Science at the California Institute of Technology, Pasadena, CA, 91125.*

• *N. Wen is an undergraduate studying Computation and Neural Systems at the California Institute of Technology, Pasadena, CA, 91125*

process an infinite amount of incoming data instantaneously. Hosts are connected to at most one link. We represent hosts in our network simulation as a Host class that extends a Device superclass. Host has these fields: network_id, links, and flows. Host has two methods: send() which calls link.receive() and receive() which calls flow.receive().

1.4.2 Routers

In an actual network, the routers are the network equipment that sits between hosts. We assume that every router can process an infinite amount of incoming data instantaneously. Routers can be connected to an arbitrary number of links. We represent routers in our network simulation as a Router class that extends a Device superclass. Router has three fields: network_id, links, and routing_table and three methods: send() which looks at routing table, receive() which checks if packet is routing packet, and update_route() which run either Dijkstra's or Bellman Ford.

1.4.3 Flows

In an actual network, flows are the active connections. Flows have a source and destination address, and generate packets at a rate controlled by the congestion. We represent flows in our network simulation a Flow superclass that has several methods: send_packet, send_ack, abstract receive, abstract timeout. The Flow superclass is extended by several different subclasses which represent the different congestion control algorithms. We have TahoeFlow extends Flow, RenoFlow extends Flow, and FASTFlow extends Flow.

1.4.4 Links

In an actual network, the links are the communication lines that connect hosts and routers together. Links connect the hosts and routers and carry packets from one end to another. Every link has a specified capacity in bits per second. Outgoing data from hosts and routers must sit on a link buffer until the link is free. The link buffer is first-in, first-out. All the links are half-duplex (data can flow in both directions, but only in one direction at a time). Each link has a static cost, based on the some intrinsic property of the link, and a dynamic cost, dependent on link congestion.

We represent links in our network simulation as a Link class with these fields: id, device_a, device_b, delay, rate, and capacity. Link as two methods: send() which enqueues packets to send to device and receive() which receives packets from devices.

1.4.5 Packets

Packets that try to enter a full buffer will be dropped. Flow generated data packets have a fixed size of 1024 bytes. Acknowledgment packets have a fixed size of 64 bytes. We represent packets in our network simulation as a Packet superclass with the subclasses: DataPacket and AckPacket. Packet has fields: packet_id, source, destination, packet_size (which is fixed for data packets and acknowledgment packets). There is an isAck method that checks to see if a packet is an acknowledgment packet.

1.4.6 Network

The Network class runs the simulation. It has these fields: devices, links, flows, ids, time and these methods: check_id, add_host, add_router, add_link, add_flow, and run.

1.5 Dynamic Routing Protocol

Our routers implement a dynamic routing protocol that uses link costs as a distance metric and route packets along a shortest path according to this metric. The dynamic routing protocol is decentralized, and thus uses message passing to communicate among routers. This message passing sends packets along the link during the simulation. Thus, our simulation implements the Bellman-Ford shortest path algorithm in a distributed manner.

1.6 Congestion Control Algorithms

1.6.1 General Flow Timeout

All the congestion control algorithms use the same calculation for the retransmission timeout time (RTO). The calculation for the retransmission timeout time is based on the RFC 6298 document on "Computing TCP's Retransmission Timer" which defines the standard algorithm that TCP uses. The only difference is that a maximum RTO of 5.0 seconds was used in our simulation whereas the document suggests a 60.0 second maximum. When the first RTT measurement is made, the flow sets $SRTT = last\_RTT$, $RTTVAR = last\_RTT/2$, $RTO = SRTT + max(G, K * RTTVAR)$ where K=4 and G=1s=1000 ms. When all the following RTT measurements are made, the flow sets $RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - lastRTT|$ and $SRTT = (1 - alpha) * SRTT + alpha * last\_RTT$ where alpha = 1/8 and beta=1/4. Then, retransmission timeout is calculated as $RTO = SRTT + max(G, K * RTTVAR)$.

1.6.2 TCP Tahoe

Our Flow class implemented all the features of the TCP Tahoe algorithm. The TahoeFlow class set the parameters for the TCP Tahoe algorithm. The Tahoe algorithm includes slow start where the window size is increased by 1 every time an ack packet is received. When the window size reaches the congestion threshold, the algorithm switches to a congestion avoidance phase where every time an ack packet is received, window size is increased by $1/cwnd$ where $cwnd$ is the previous window size. If a packet timeouts, which is usually due to packet drop, the congestion threshold is set to half the window size and the congestion window, $cwnd$ is set to 1. Then the flow begins the exponential slow start until it reaches the threshold again. When a packet timeout occurs, the packet is resent and then the flow continues sending packets which acknowledgements have not been received for.

1.6.3 TCP Reno

Our RenoFlow class extended the TahoeFlow class. Our RenoFlow class inherits slow start and congestion avoidance from the TahoeFlow class which inherits them from the Flow class. In addition, we implemented fast recovery and fast retransmit which makes up the TCP Reno algorithm. The RenoFlow class keeps track of the next packet expected when sending acks so that if it receives 3 duplicate acks, it will resend the packet that it got the duplicate acks for and continue sending packets that have not been received until it gets the ack that the packet has been received. When the 3 duplicate acks are received, the slow start threshold is set to half of the window size which is equal to half of the flightsize, where flightsize is the number of packets that have been sent but no ack has been received for. Then the congestion window is set to ssthresh + window inflation. $cwnd = ssthresh + ndup$ In this case, window inflation or ndup(number duplicate) is 3. Then once the window size exceeds the current flightsize, the packet is resent. Once the ack is received for this packet one round trip later, the congestion window is decreased to the slow start threshold. $cwnd = ssthresh$ If a packet timeout occurs, then window size is set to 1 and slow start occurs again until the slow start threshold is reached.

1.6.4 FAST-TCP

The FastFlow class implemented all the features of the proprietary algorithm FAST TCP under the suggestion of one of the developers of the original algorithm, Professor Low. Every 20 ms, the interval which was suggested by Professor Low, the window size updates as follows: $W' = min(2 * W, (1 - \gamma)W + \gamma(baseRTT/RTT * W + \alpha)$ where $\gamma = 0.8$ and $\alpha = 20$. While Professor Low suggested $\gamma = 1$ for simplicity, a lower γ factor smoothed out the window size over time. The α was chosen based on the link rate. From Professor Low's expertise, $\alpha = 3$ for link rates less than 1 Mbps, 10 for link rates between 1 and 10 Mbps, 20 for link rates between 10 Mbps and 1 Gbps, and 50 for link rates above 50 Gbps. Since the link rates were between 10 and 12.5 Mbps, $\alpha = 20$ was chosen for this project. In addition, simulation tests showed that using $alpha = 10$ did not fully utilize the link capacities. This algorithm was notably faster than TCP Reno and TCP Tahoe for the simulation test cases.

1.7 Test Cases

1.7.1 Case 0

This is a basic test case consisting of two hosts, a single link, and a single flow. There is no routing involved, and simply tests some basic properties of our simulation.

1.7.2 Case 1

This case tests the dynamic routing capability of the simulation. It provides two different paths from Host 1 to Host 2. It is expected that the path will alternate due to the dynamic routing.

The TCP properties should be similar to Case 0 because the time scale of the routing path changes is larger than the TCP adjustment time scale.

1.7.3 Case 2

This case tests the behavior of the simulation on a more complex network involving multiple hosts and flows.

This case gives insights on the behavior of various TCP algorithms as flows enter and exit the system. The mathematical methods discussed in class to model TCP algorithm equilibria as network utility maximization (NUM) optimization problems give a rigorous way to reason about the behavior of the system as flows enter and exit.

1.8 Metrics and Graphs

The metrics we graphed included: per-link buffer occupancy, packet loss, flow rate, and packet delay. Our graphs display all the flow rates on the same graph, all the link rates on the same graph, etc. for comparison purposes.

Fig. 1. Case 0

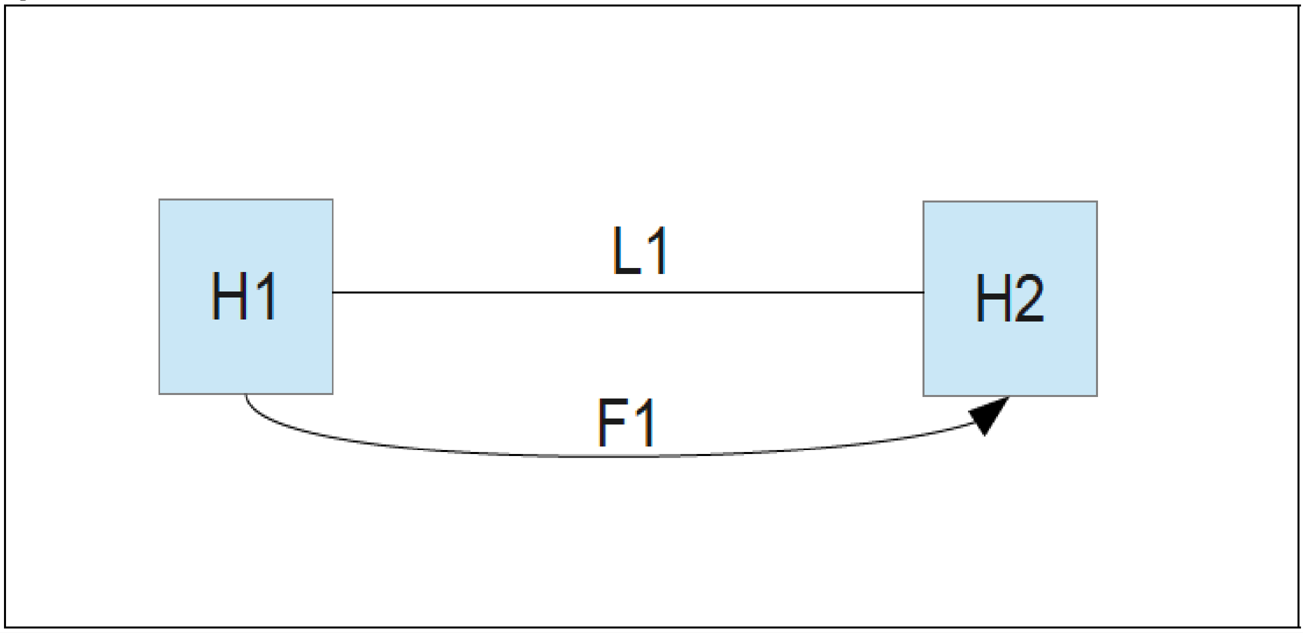


Fig. 2. Case 0 Reno Flow Rate and Link Rate

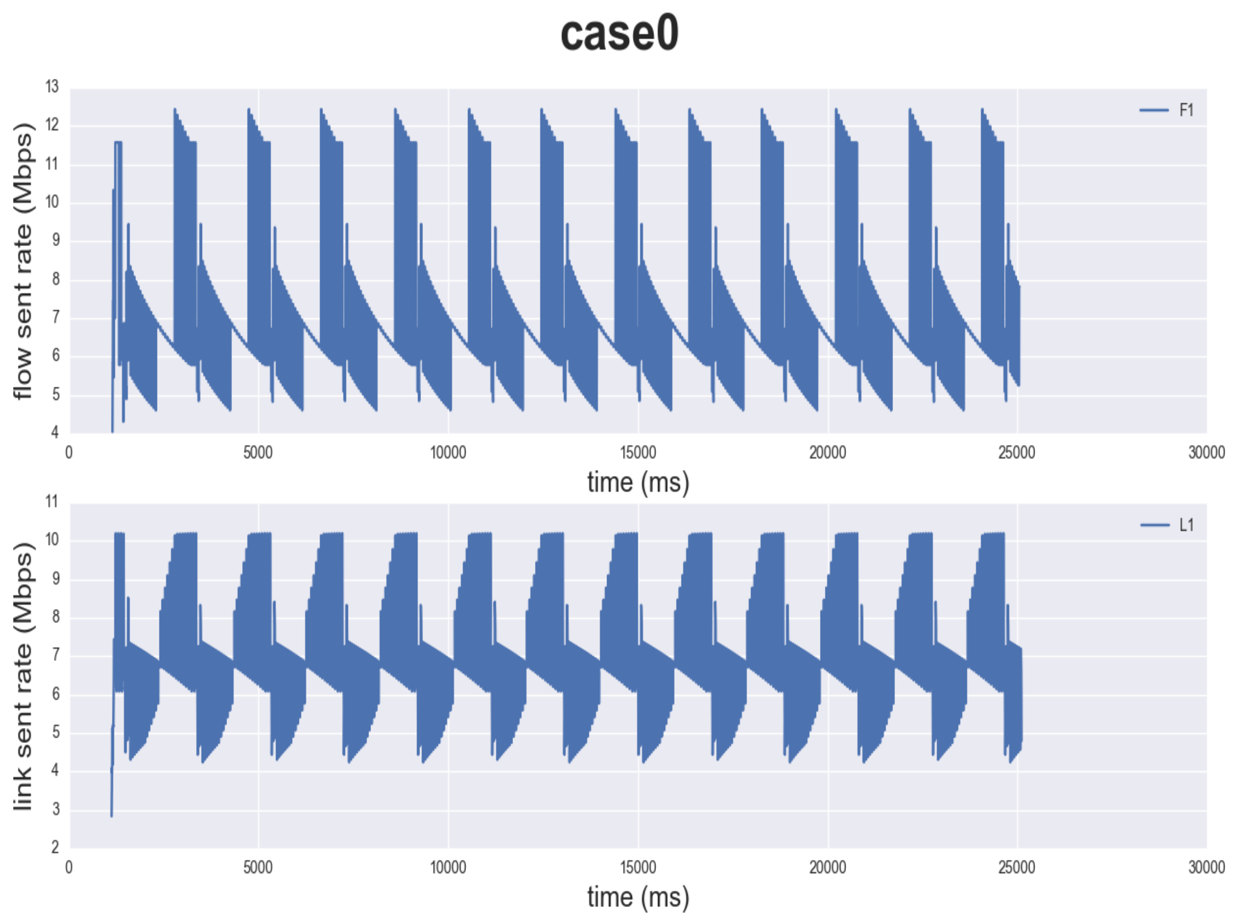


Fig. 3. Case 0 Reno Buffer Data

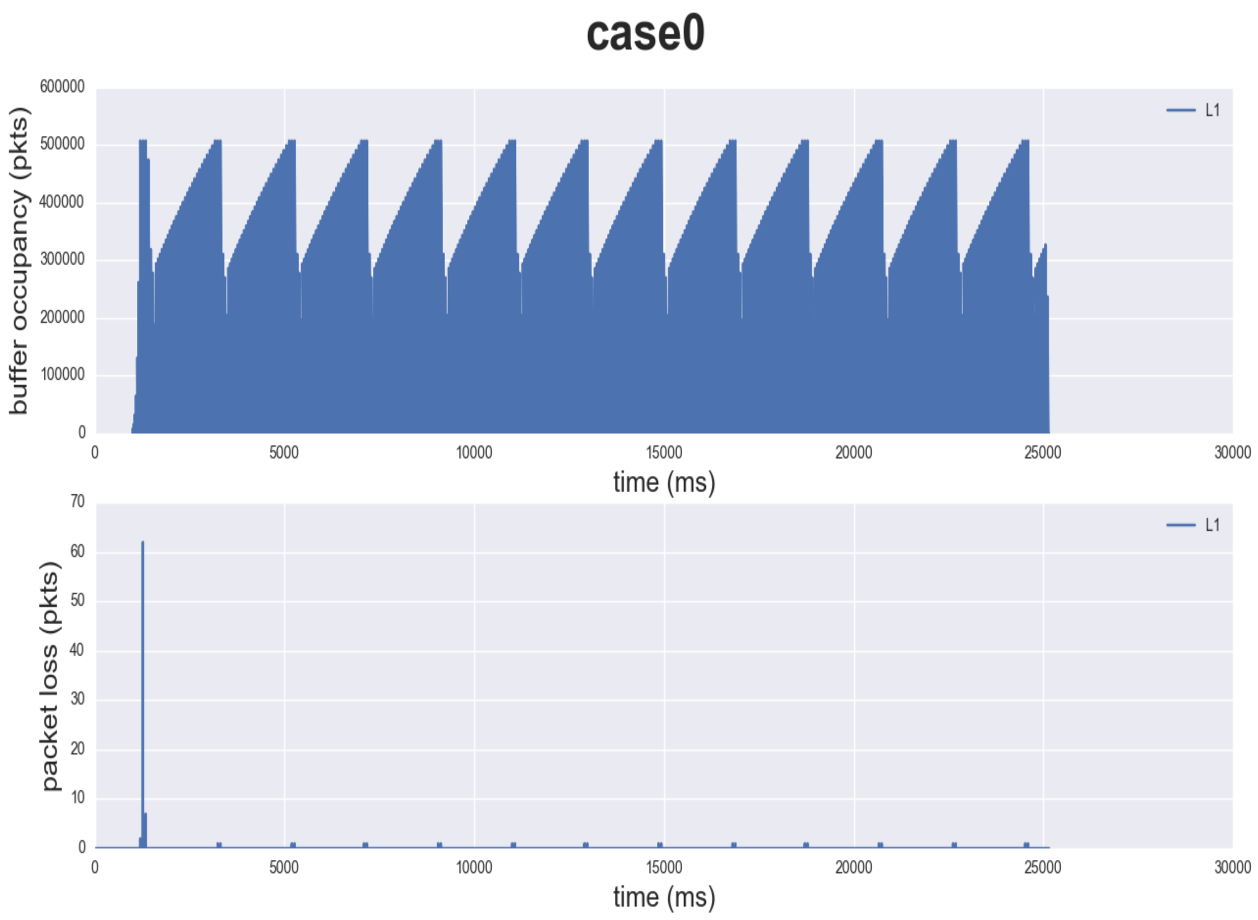


Fig. 4. Case 0 Reno Window Data

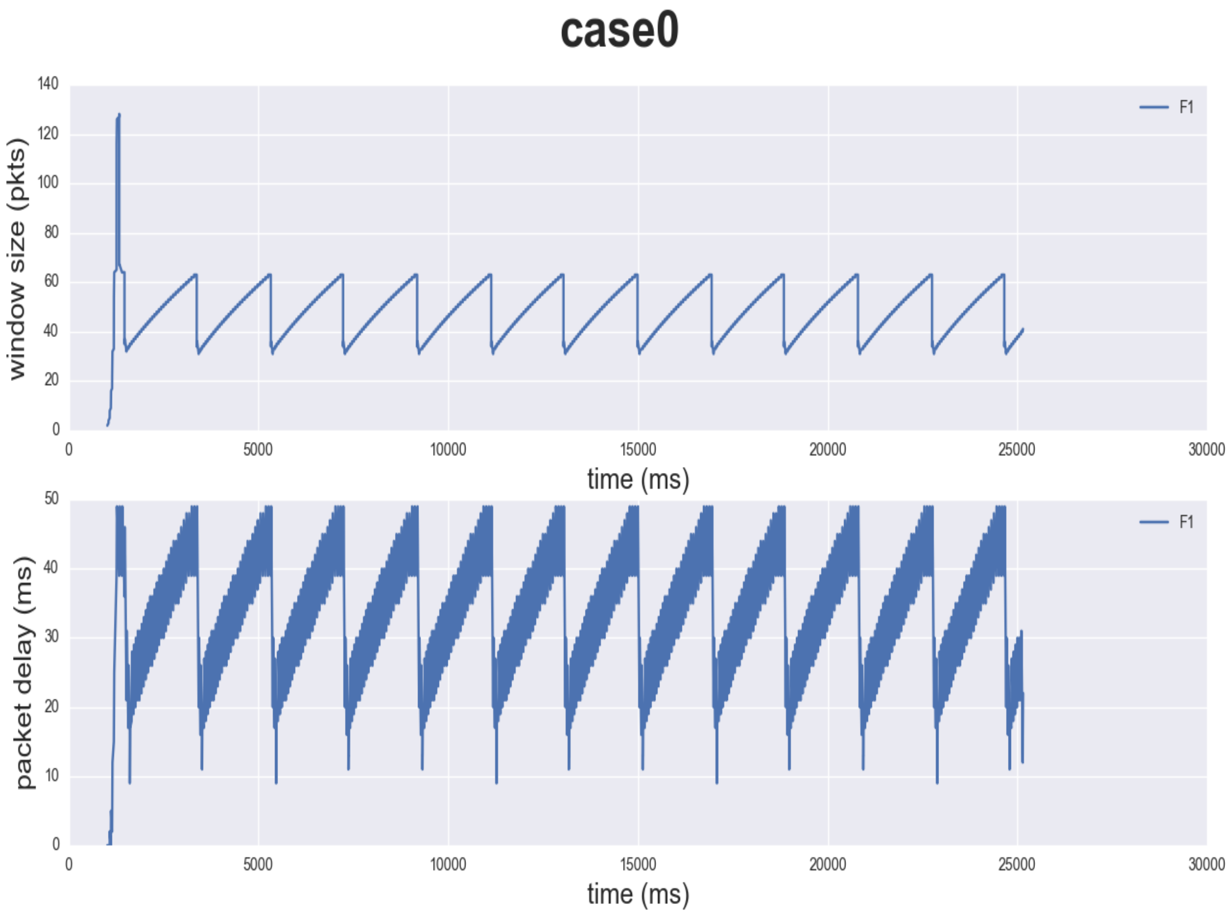


Fig. 5. Case 0 FAST Flow Rate and Link Rate

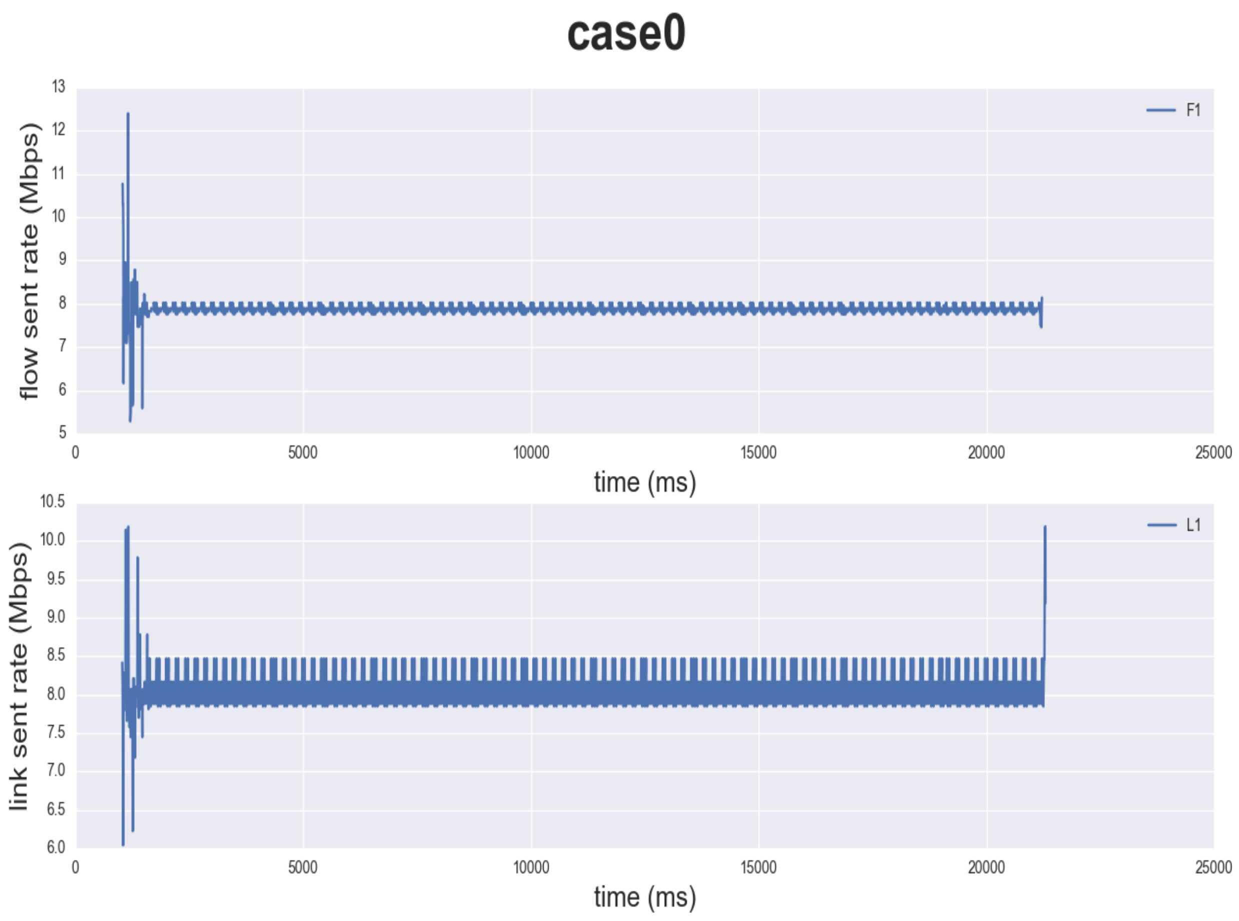


Fig. 6. Case 0 FAST Buffer Data

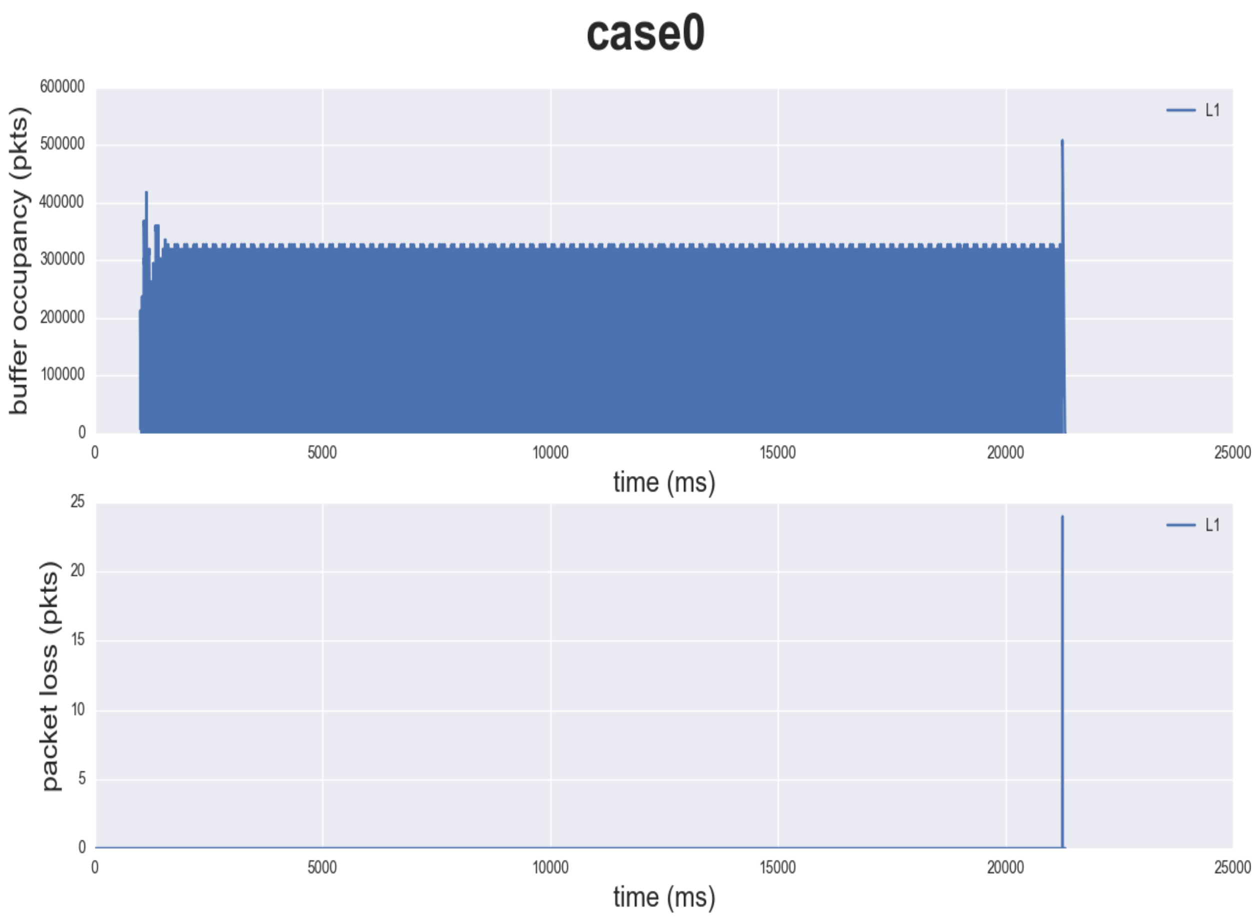


Fig. 7. Case 0 FAST Window Data

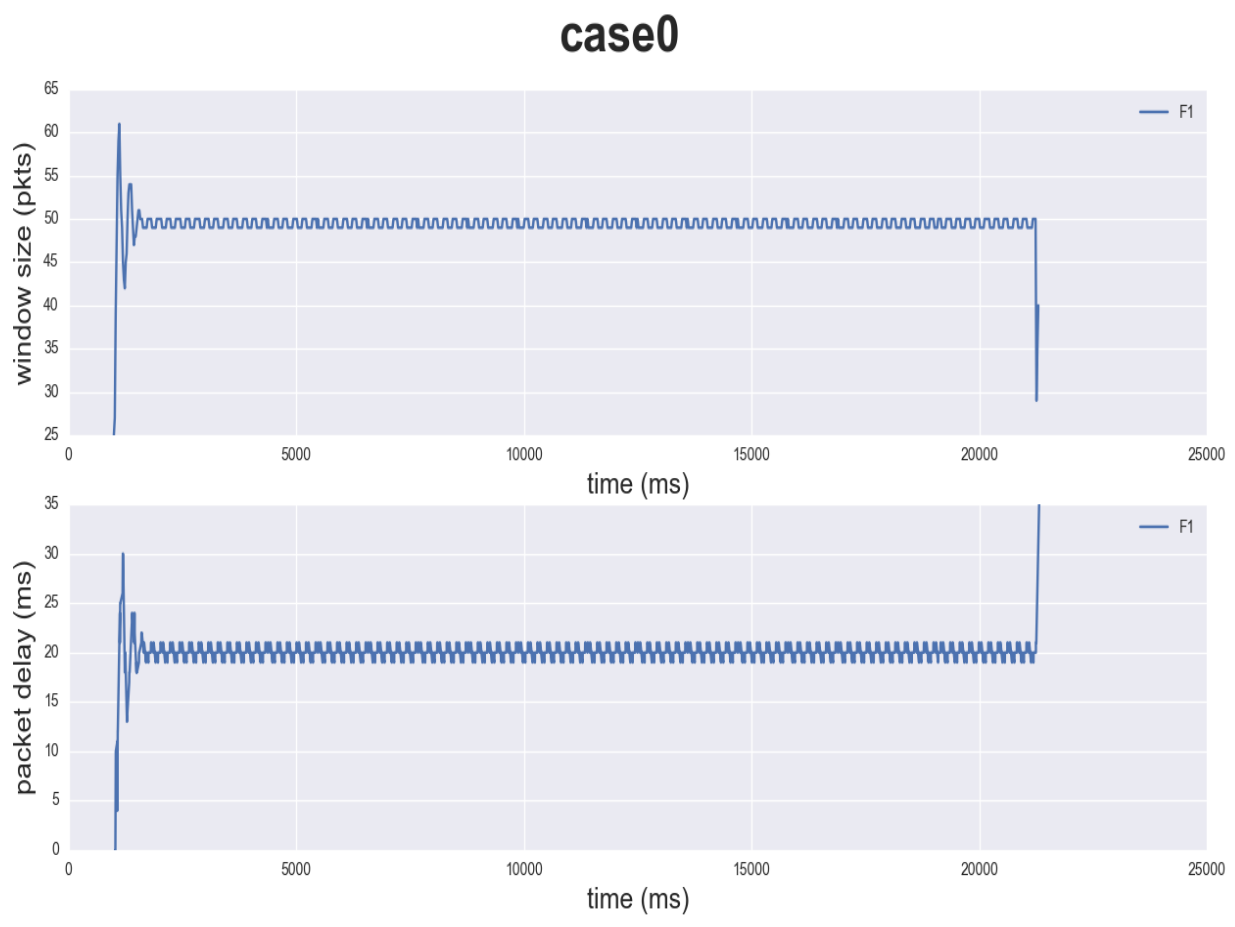


Fig. 8. Case 1

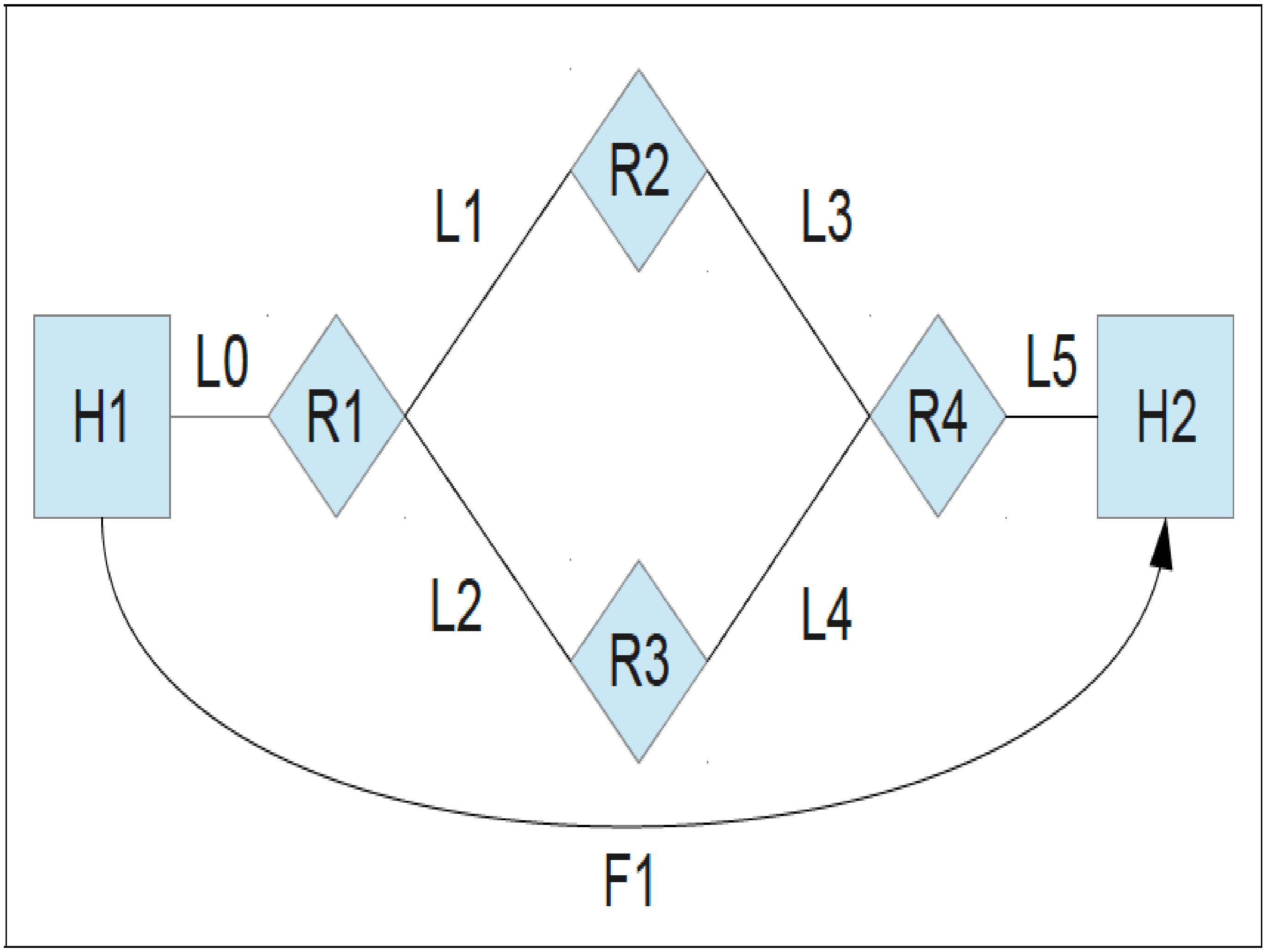


Fig. 9. Case 1 Reno Flow Rate and Link Rate

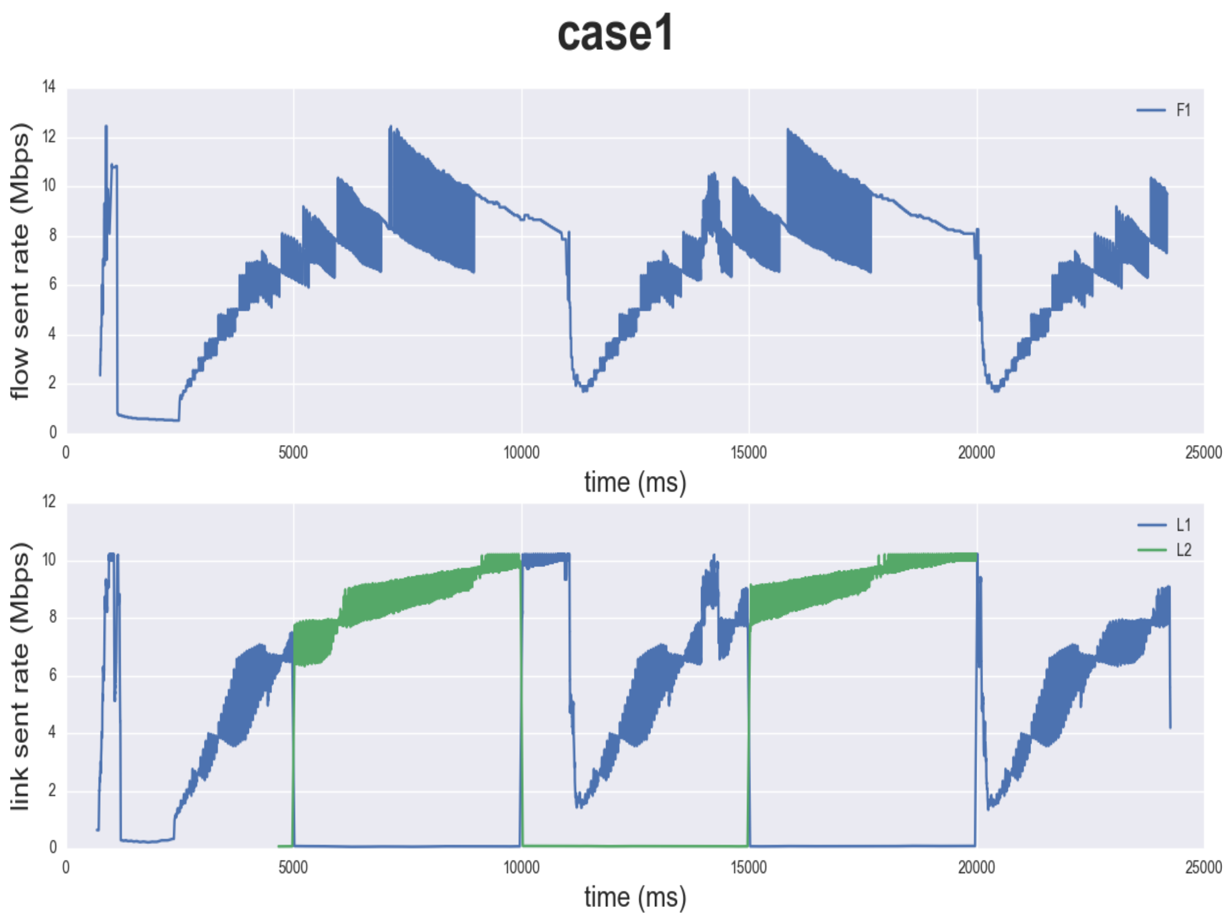


Fig. 10. Case 1 Reno Buffer Data

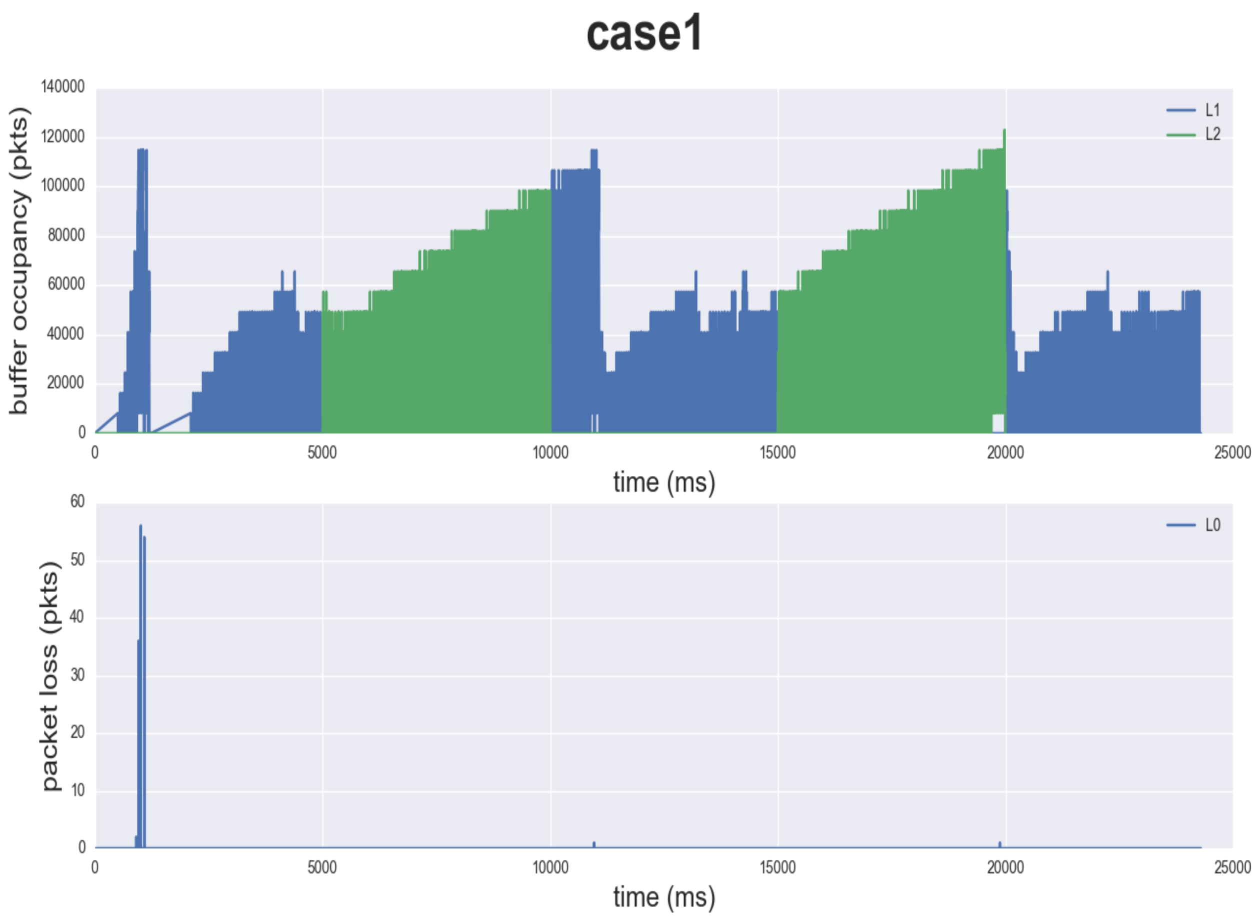


Fig. 11. Case 1 Reno Window Data

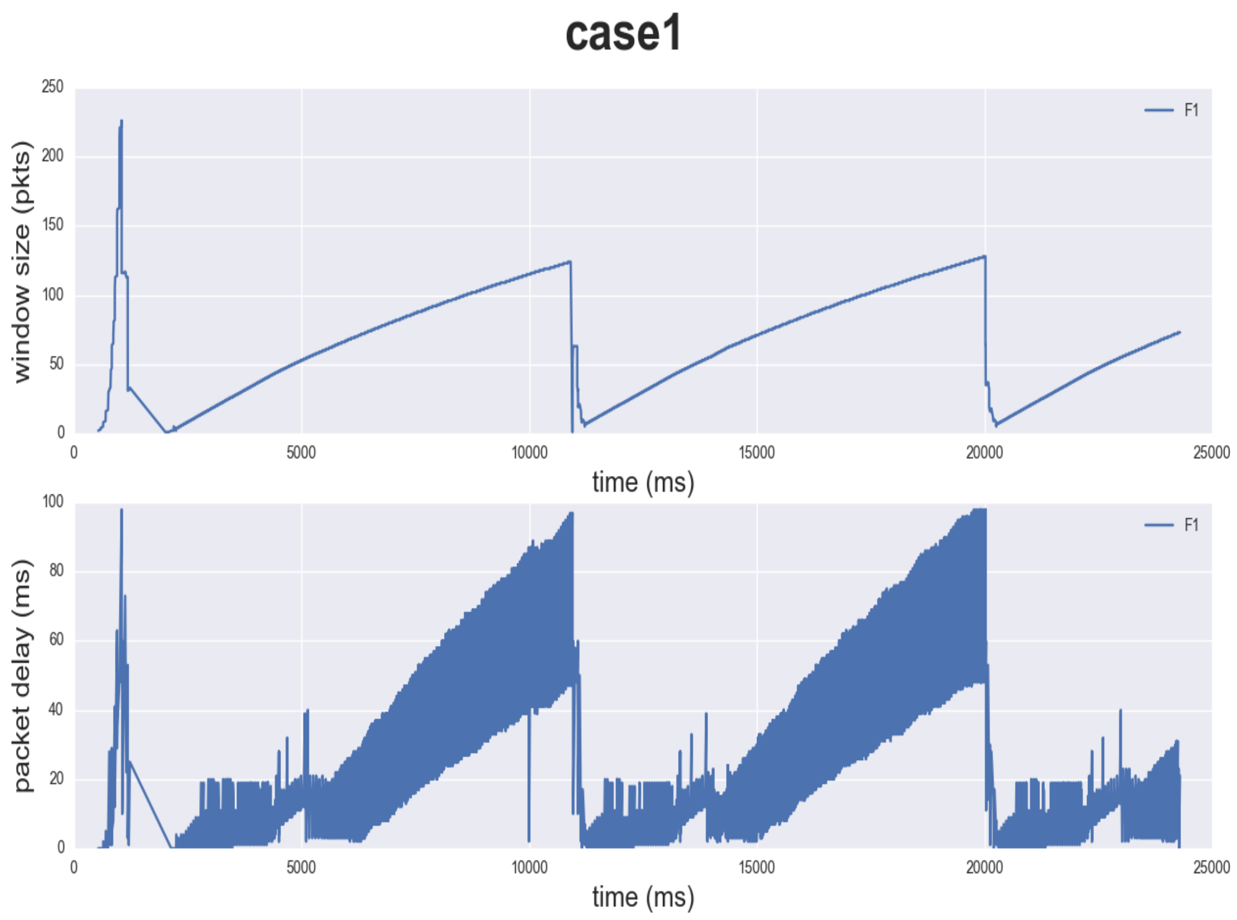


Fig. 12. Case 1 FAST Flow Rate and Link Rate

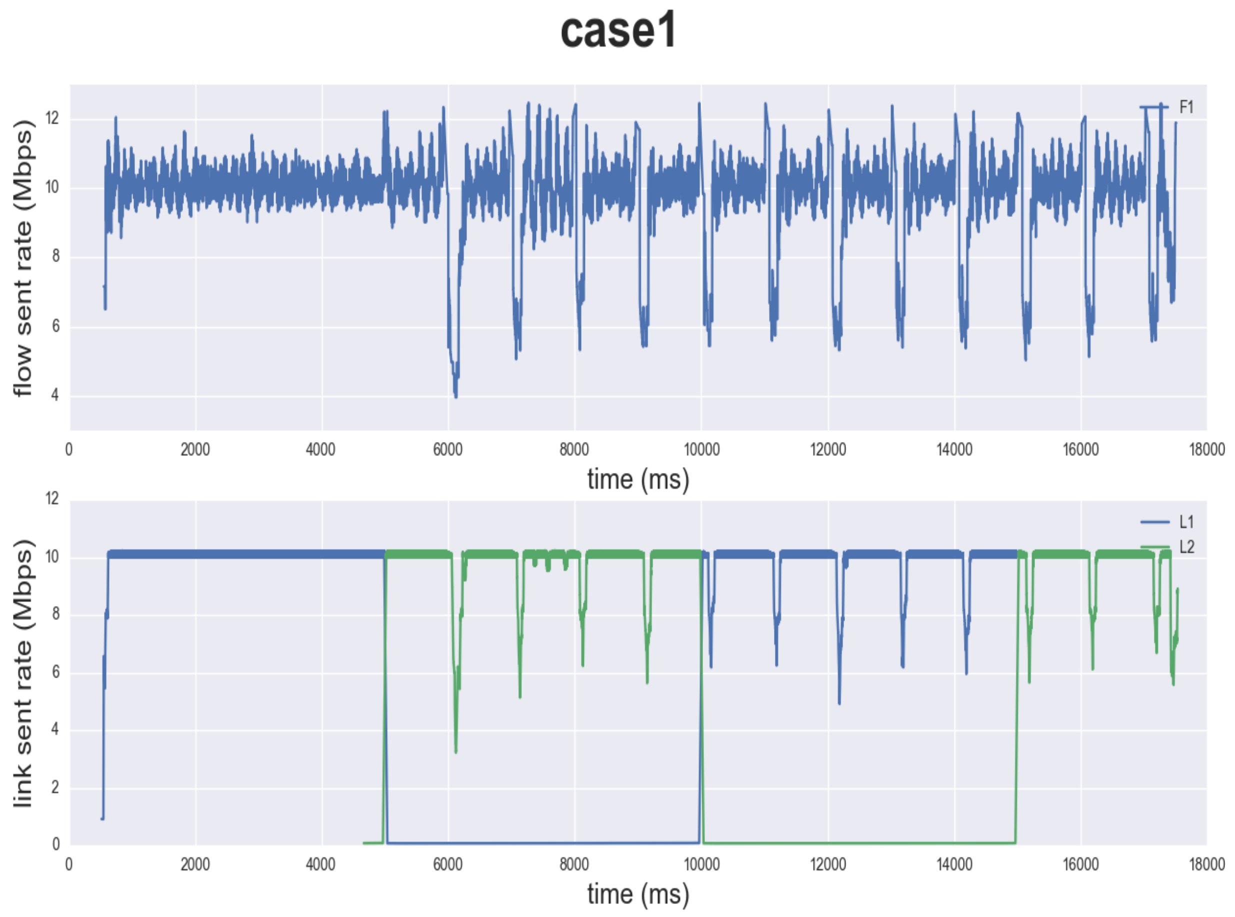


Fig. 13. Case 1 FAST Buffer Data

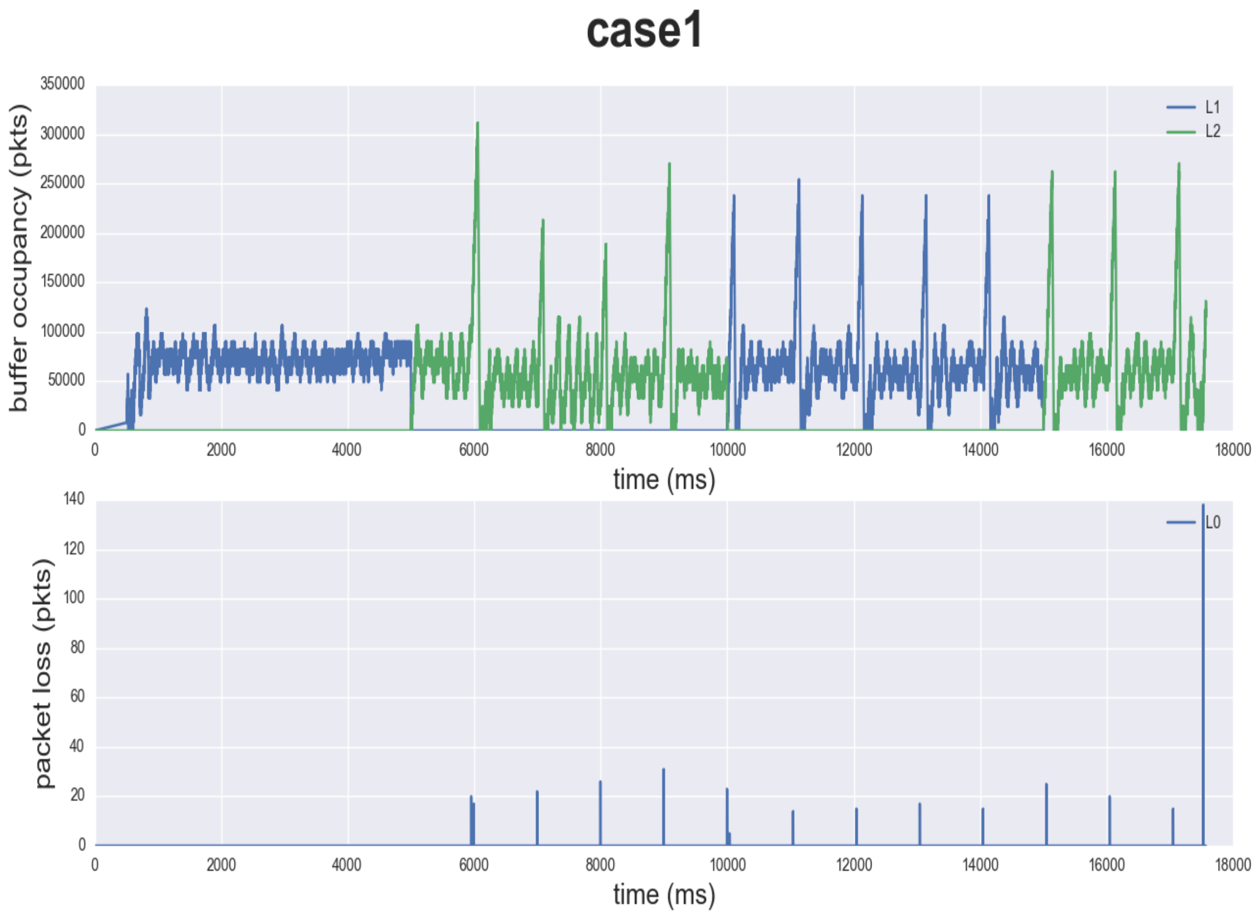


Fig. 14. Case 0 FAST Window Data

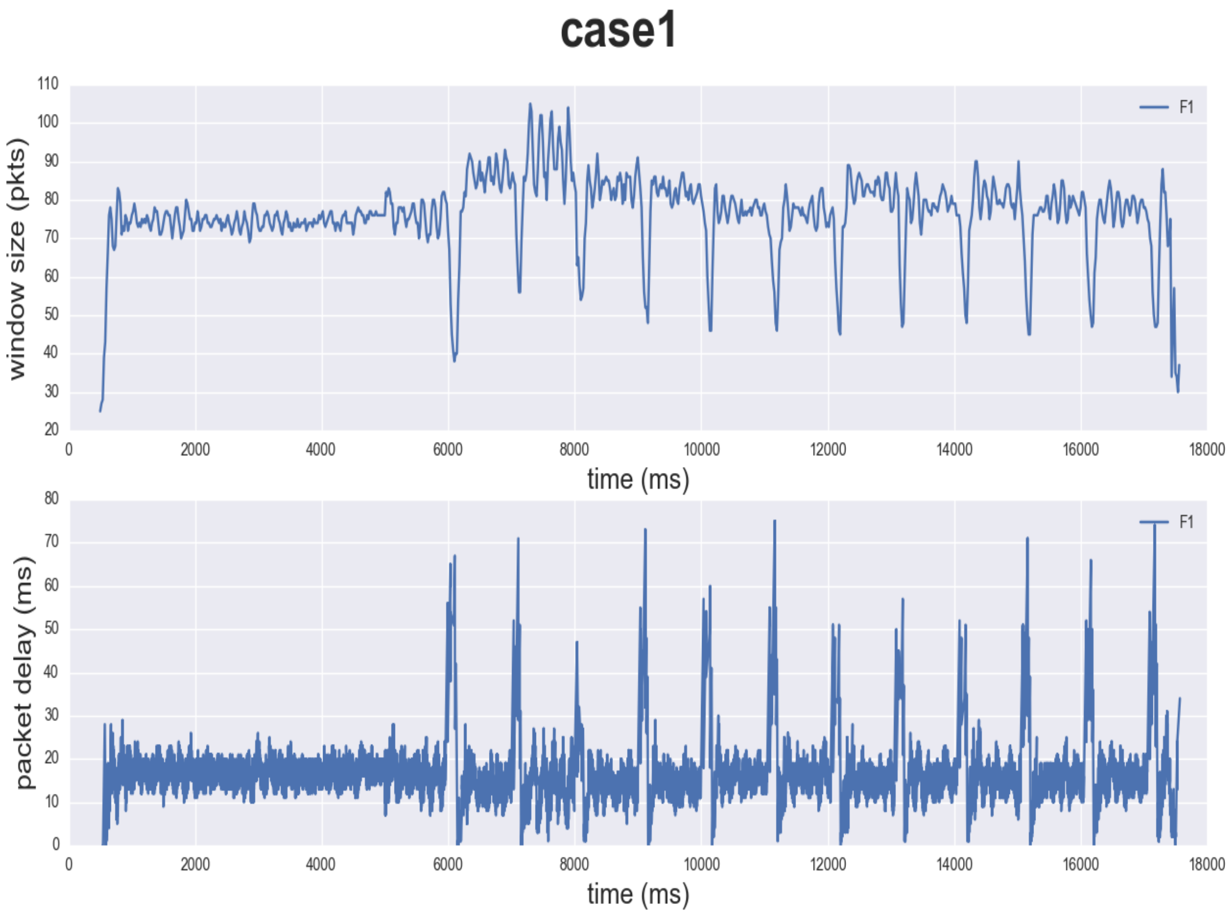


Fig. 15. Case 2

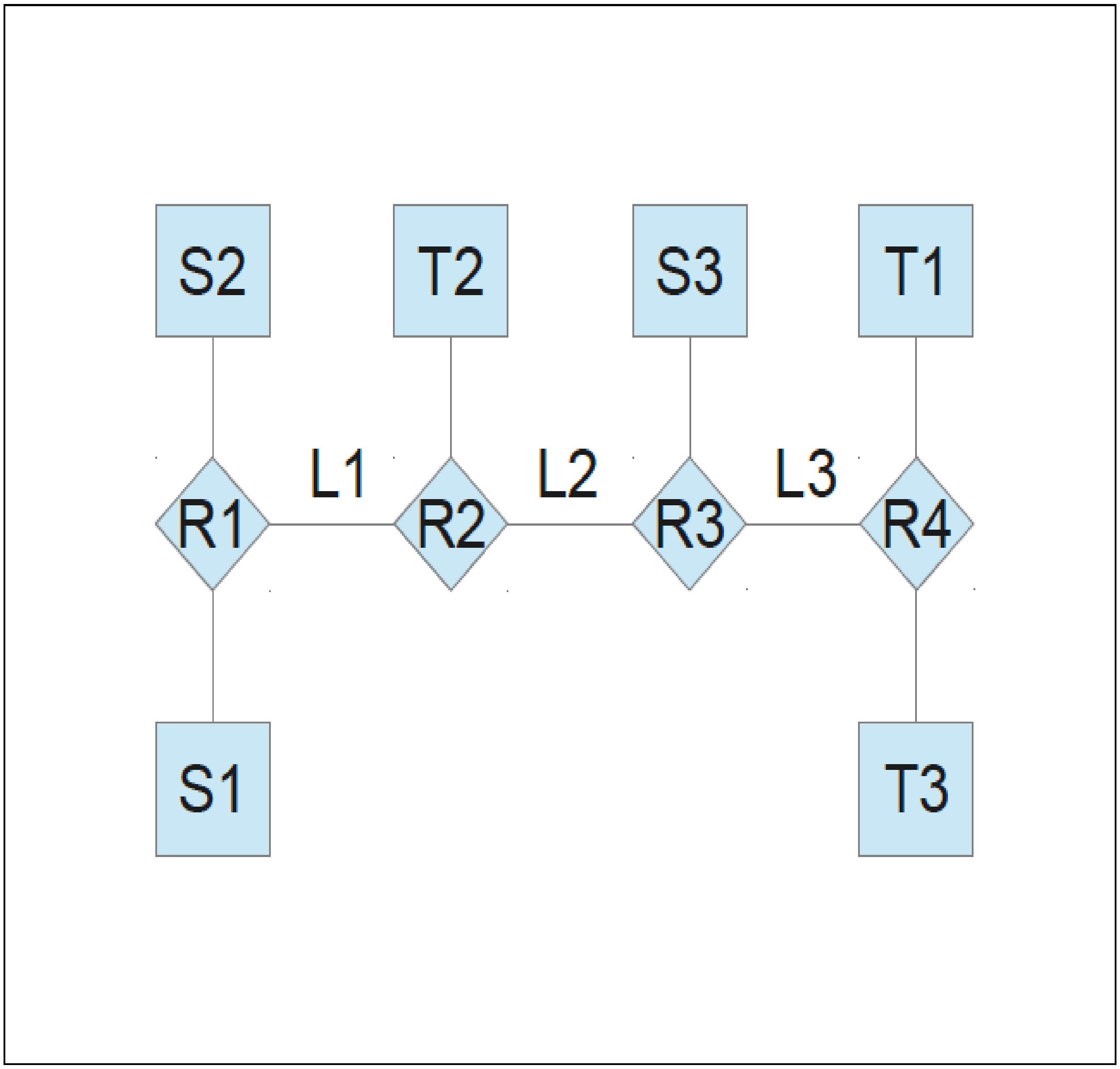


Fig. 16. Case 2 Reno Flow Rate and Link Rate

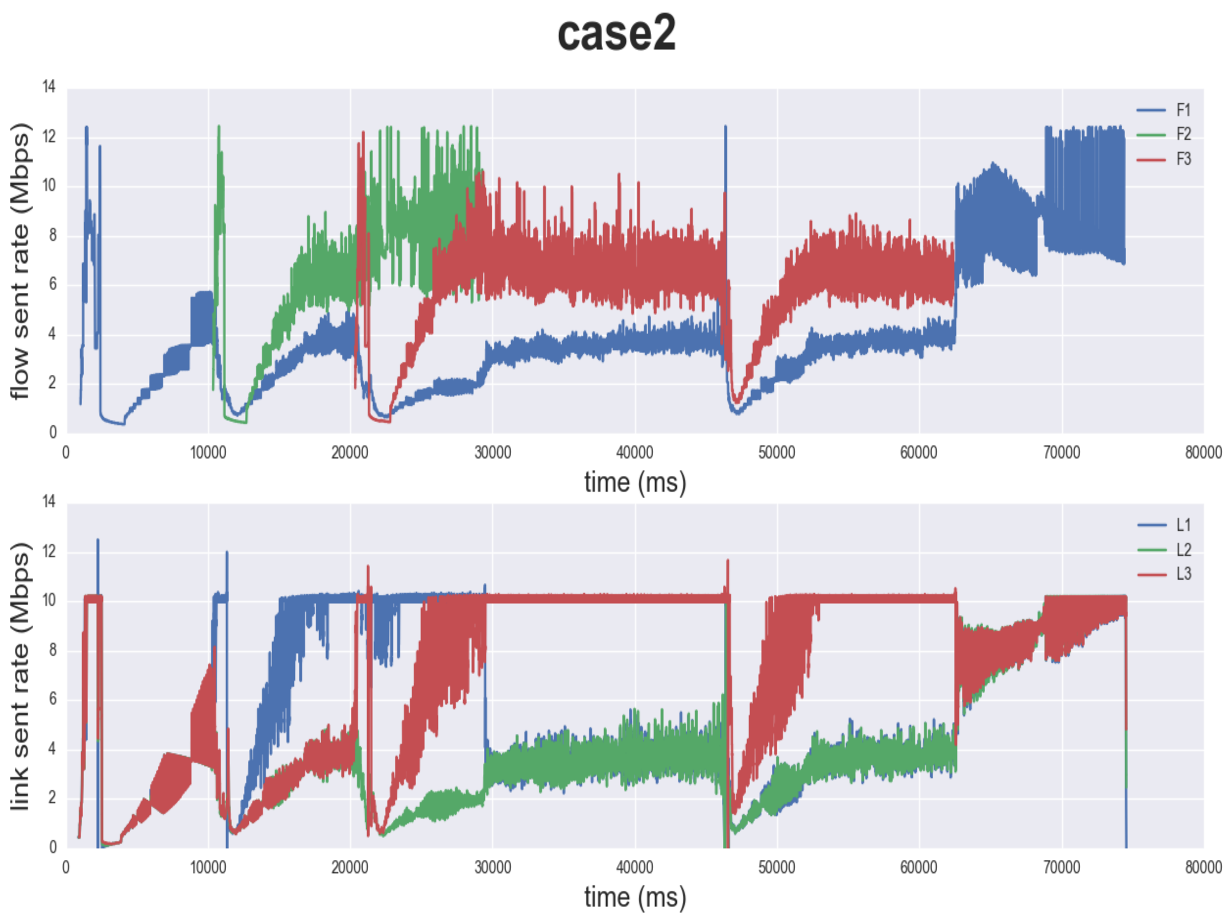


Fig. 17. Case 2 Reno Buffer Data

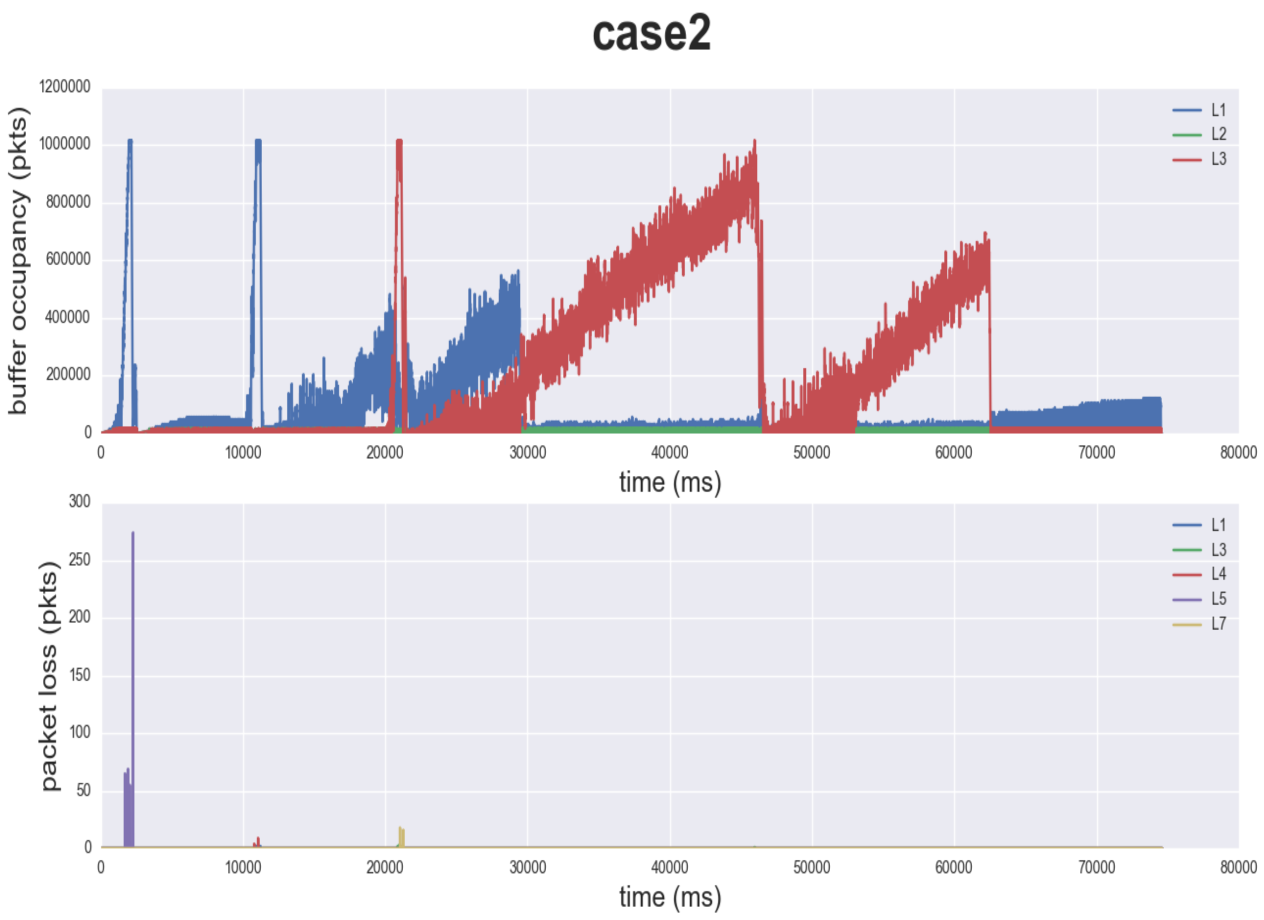


Fig. 18. Case 2 Reno Window Data

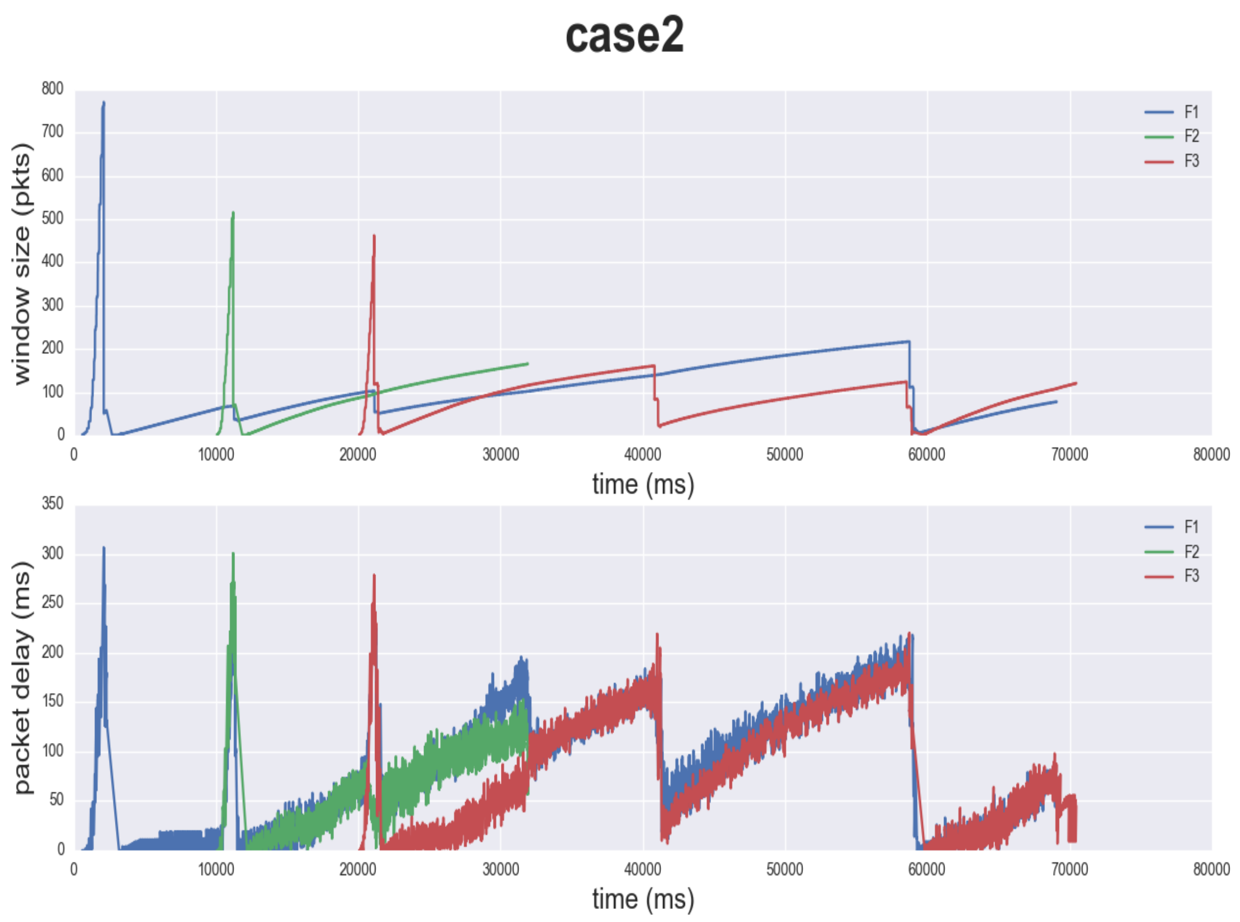


Fig. 19. Case 2 FAST Flow Rate and Link Rate

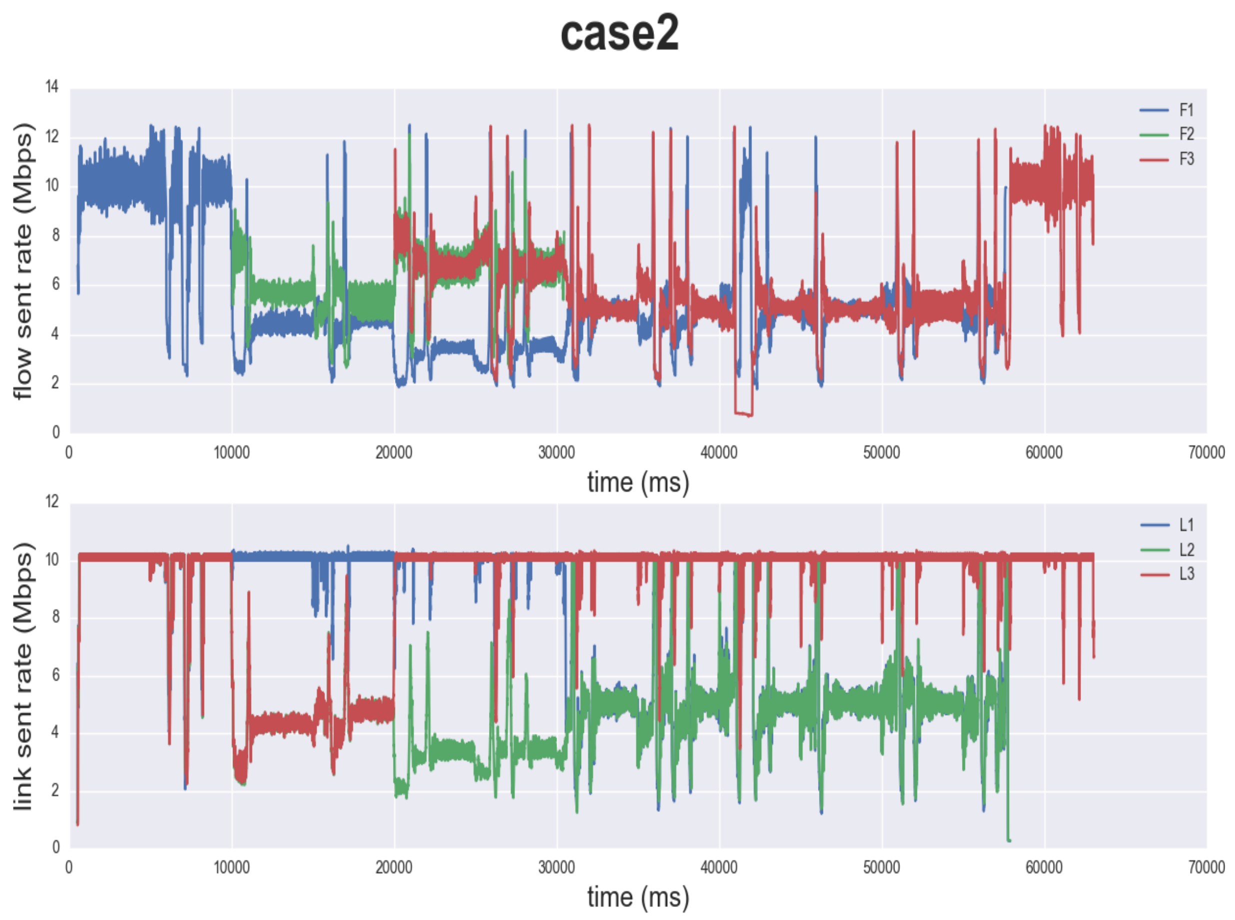


Fig. 20. Case 2 FAST Buffer Data

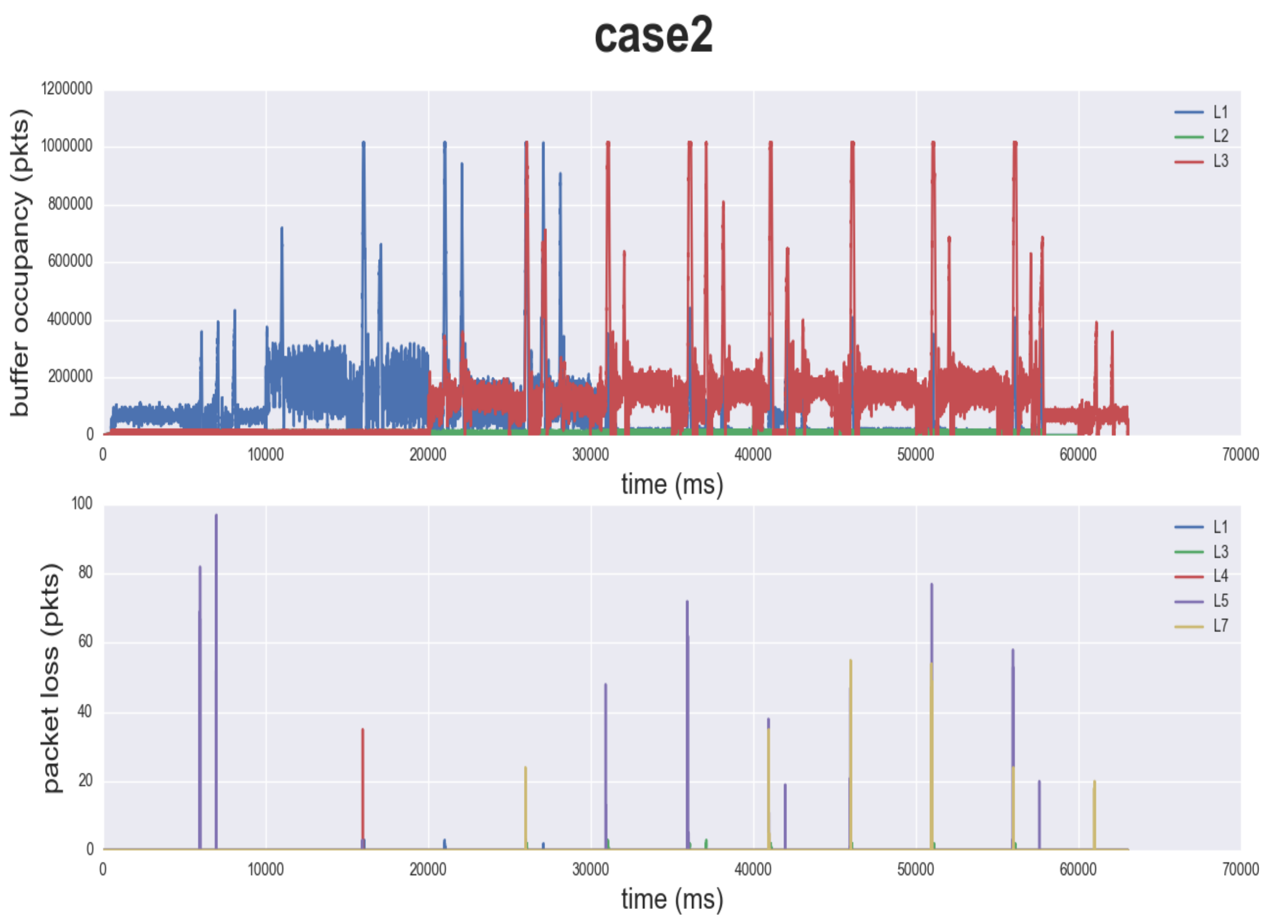
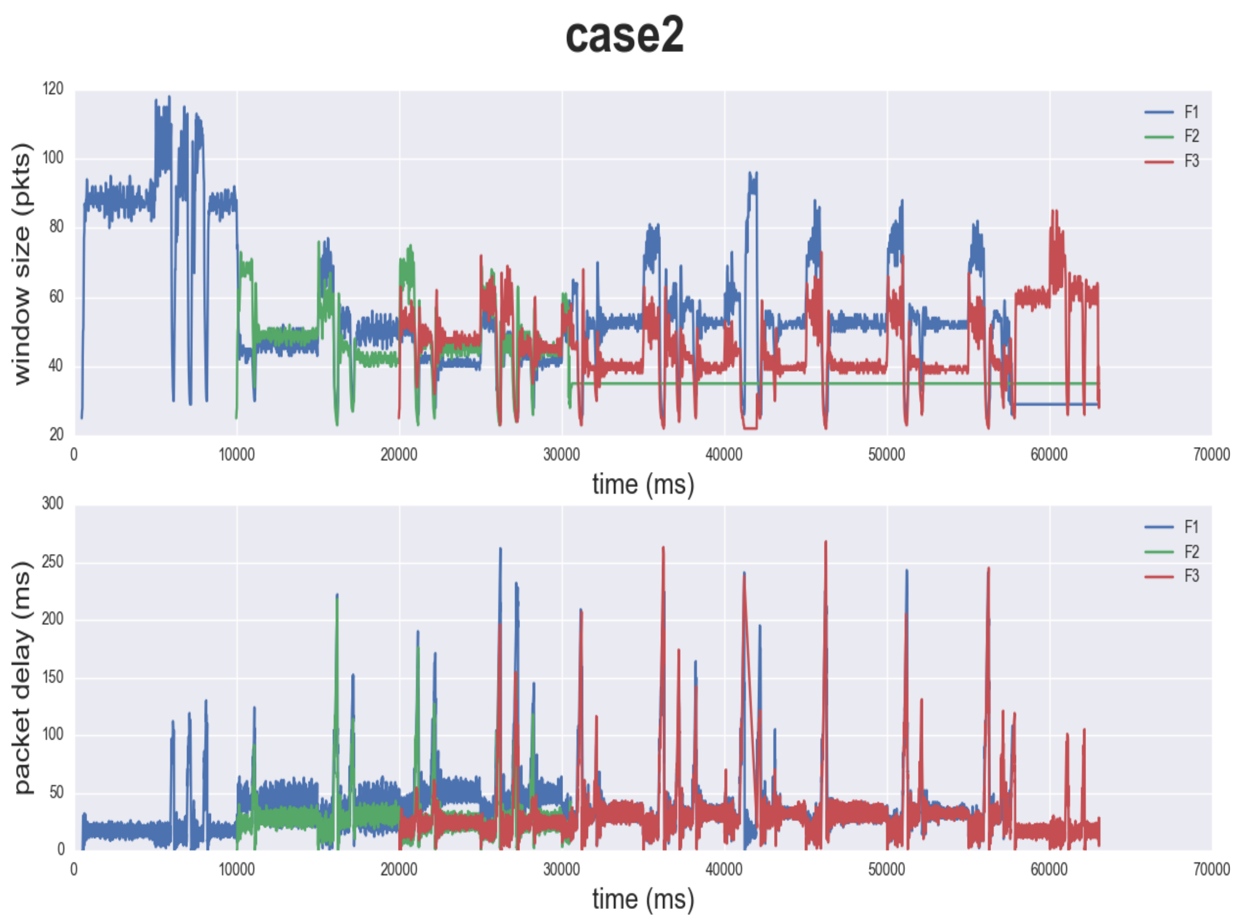


Fig. 21. Case 2 FAST Window Data



1.9 Simulation Results

1.9.1 Theoretical Expectations

The theoretical expectations for cases 0 and 1 are simple: since there is only one flow, it will use the link to its maximum capacity. Thus, we expect an equilibrium flow rate of $10\,\mathrm{Mbps}$, which would take $16.0\,\mathrm{s}$ to transfer $20\,\mathrm{MB}$. Note that this expectation ignores ACK packets, so the actual time should be slightly higher. Since case 0 starts at $1.0\,\mathrm{s}$, we expect its flow to finish around $17.1\,\mathrm{s}$ (adding in the propagation delay of $0.1\,\mathrm{s}$) Similarly, case 1 starts at $0.5\,\mathrm{s}$, so we expect its flow to finish around $16.9\,\mathrm{s}$ (with a $0.4\,\mathrm{s}$ total propagation delay from Host 1 to Host 2).

Case 2 involves multiple flows, which requires the NUM interpretation of the TCP network to solve. A detailed analysis of this case for the FAST TCP algorithm is discussed in Appendix A. In summary, for the FAST TCP algorithm we expect Flow 2 to finish first at $t_2 = 27.95\,\mathrm{s}$ Flow 1 to finish next at $t_1 = 53.05\,\mathrm{s}$ and Flow 3 to finish last at $t_3 = 58.68\,\mathrm{s}$.

Note that these expectations are optimal expectations that assume all flows instantly reach equilibrium. This is of course not valid in practice, but gives a good lower bound on the simulation time.

1.9.2 Observed Results

For Case 0, our Reno simulation took approximately $23\,\mathrm{s}$ to finish. Our FAST TCP simulation took $17.621\,\mathrm{s}$ to finish, which is close to our optimal expectation of $17.1\,\mathrm{s}$. The Reno number is also close, but even at this small scale the difference in efficiency between the Reno and TCP algorithms is quite noticeable.

For Case 1, our Reno simulation took approximately $29\,\mathrm{s}$ to end while our FAST TCP simulation took $17.421\,\mathrm{s}$. This is close to our optimal expectation of $16.9\,\mathrm{s}$. Note that the simulation time for Case 1 was less than the time for Case 0, as theoretically expected.

For Case 2, our Reno simulation took approximately $67\,\mathrm{s}$ to end, and the flows finished in the order F2, F3, F1, which matches the sample time traces provided. In our FAST TCP simulation, F2 finished at $30.426\,\mathrm{s}$, F1 finished at $57.511\,\mathrm{s}$, and F3 finished at $62.991\,\mathrm{s}$. These numbers are close to the optimal expectation.

Additionally, the flow rate graphs show that when F2 joins, F2 takes more of the capacity because it absorbs F1's initial queuing delay into its RTT_min. After F3 enters, F3 and F2 have the same share of capacity and take up even more of the capacity. This matches our expectations.

1.9.3 Subtle Nuances Observed

Initially, we had some time traces that differed significantly from those provided in the assignment packet. In addressing these differences, we discovered that our discrepancies came from accounting for the effect of non-zero sized acknowledgment and routing packets, and particular parameters in our algorithm implementations.

1.10 Additional Features

1.10.1 Interactive Input

We created an interactive method of input as opposed to reading in JSON files.

1.10.2 Live graphing

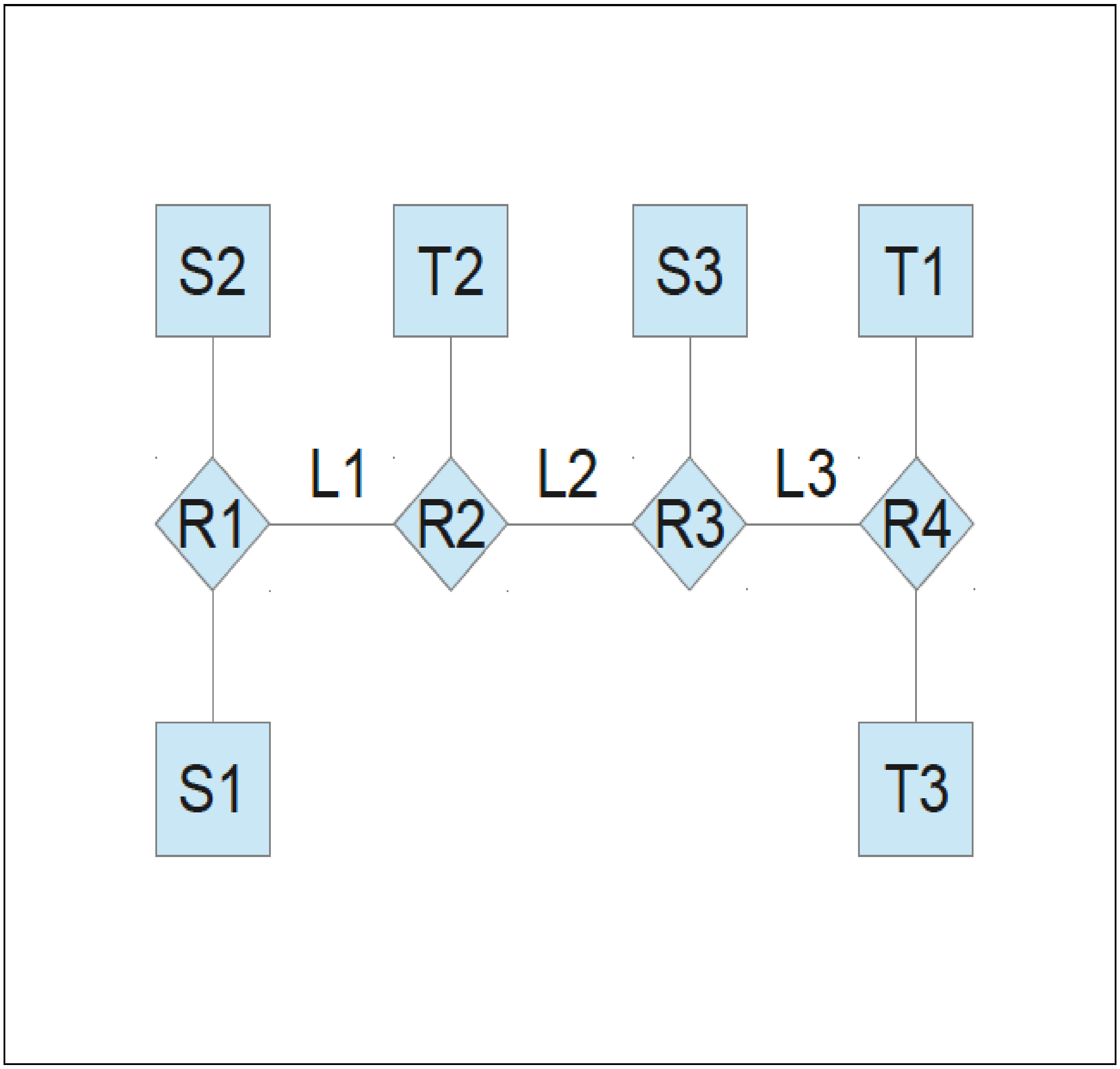
In addition to the default generation of plots after the simulation has ended, we have also implemented live graphing. There is the option of having the results graphed as they are being generated and display them on the screen.

2 Conclusion

The BlackWidow module allows for a customizable simulation of a TCP network and provides tools for visualizing the network and simulation data. Our script provides a way to interactively build up a network to simulate to easily test various nuances of TCP congestion control. The general BlackWidow module can be imported in any Python script, so our simulation can be used to fine tune various parameters.

Future work can be done to graph the simulation time as a function of various parameters including routing update period, timeout calculation parameters, re-transmission times, and even routing and acknowlegment packet size.

Fig. 22. Case 2 Network Graph (Reference for Appendix A)



APPENDIX A
CASE 2 FAST-TCP ANALYSIS

In the FAST-TCP algorithm, the equilibrium window size, W^*, is given by the relationship $W = \frac{\alpha RTT}{RTT - RTT_{min}}$ where RTT is the equilibrium round trip time, and RTT_{min} is the minimum experienced round trip time, which is a flow's best estimate of its propagation delay. α is a parameter determined by algorithm design. In our case $\alpha = 20$ packets.

A.1 Case 2 Parameters

For case two, we have 3 links of interest labeled L1, L2, and L3. These links all have a link rate of $c = 10$ Mbps. There are 3 flows, F1, F2, and F3. Flow F_i goes from S_i to T_i. Flow 1 starts at 0.5 s and has a total transfer amount of 35 MB = 280 Mb. Flow 2 starts at 10 s and has a total transfer amount of 15 MB = 120 Mb. Flow 3 starts at 20 s and has a total transfer amount of 30 MB = 240 Mb.

A.2 0.5 to 10 s equilibrium

This case is trivial since there is only one flow. Thus, at equilibrium $x_1 = c = \boxed{10\,\text{Mbps}} \Rightarrow q_1 = \frac{\alpha}{x_1} = \frac{20\,\text{packets}}{10\,\text{Mbps}} \cdot \frac{8192\,\text{bits}}{1\,\text{packet}} = \boxed{0.0164\,\text{s}}$.

Note the value of $8192\,\text{bits}$ doesn't take into account ACK packets. In general, the rigorous analysis of ACK packets is complex and requires detailed computation of algorithm details. Thus for the scope of this project we simply ignore ACK packets in our theoretical analysis.

A.3 10 to 20 s equilibrium

When Flow 2 enters, it measures the queuing delay from the Flow 1 in the previous case as part of its RTT_{min}. Let us denote the queuing delay of $0.0174\,\text{s}$ from the previous case as q^*. Then, at equilbrium $x_2 = \frac{W_2}{RTT} = \frac{\alpha}{RTT - RTT_{min}} = \frac{\alpha}{q_2 + d_2 - (q^* + d_2)} = \frac{\alpha}{q_2 - q^*} \Rightarrow q_2 = q^* + \frac{\alpha}{x_2}$.

By the NUM TCP theory discussed in class, we can view the equilibrium as the solution to the optimization problem

$$\max \sum_i U_i(x_i)$$

subject to the constraint $Rx \leq c$, where R is the routing matrix. In our case, $U_1(x_1) = \int q_1\,dx_1 = \alpha \log x_1$ and $U_2(x_2) = \int q_2\,dx_2 = q^* x_2 + \alpha \log x_2$. Our bottleneck link is L1, so we have the constraint $x_1 + x_2 = c$. Hence, we wish to maximize $U(x_1, x_2) = \alpha \log x_1 + \alpha \log x_2 + q^* x_2$. By substituting in the constraint, we reduce the problem to the single variable problem of optimizing $U(x_2) = \alpha \log(c - x_2) + \alpha \log x_2 + q^* x_2$. $\frac{dU}{dx_2} = 0 \Rightarrow$

$$0 = -\frac{\alpha}{c - x_2} + \frac{\alpha}{x_2} + q^*$$

$$0 = -\alpha x_2 + \alpha c - \alpha x_2 + q^*(c x_2 - x_2^2)$$

$$0 = q^* x_2^2 + (2\alpha - q^* c) - \alpha c$$

$$x_2 = \frac{q^* c - 2\alpha \pm \sqrt{(q^* c - 2\alpha)^2 + 4 q^* \alpha c}}{2 q^*}$$

Plugging in our numbers and eliminating the extraneous solution yields:

$$\boxed{x_1 = 3.82\,\text{Mbps}, q_1 = 0.0429\,\text{s}}$$

$$\boxed{x_2 = 6.18\,\text{Mbps}, q_2 = 0.0429\,\text{s}}$$

A.4 20 to 27.95 s equilibrium

Flow 3 is able to accurately measure its RTT, so its utility function is simply $U_3(x_2) = \alpha \log x_3$. Hence, we wish to maximize $U(x_1, x_2, x_3) = \alpha(\log x_1 + \log x_2 + \log x_3) + q^* x_2$. There are two bottleneck links, giving the constaints $x_1 + x_2 = c$ and $x_1 + x_3 = c \Rightarrow x_2 = x_3$. Substituting in these constraints yields the single variable problem $U(x_2) = \alpha \log(c - x_2) + 2\alpha \log x_2 + q^* x_2$ Setting $\frac{dU(x_2)}{dx_2} = 0$:

$$0 = -\frac{\alpha}{c - x_2} + \frac{2\alpha}{x_2} + q^*$$

$$0 = -\alpha x_2 + 2\alpha c - 2\alpha x_2 + q^*(c x_2 - x_2^2)$$

$$0 = q^* x_2^2 + (3\alpha - q^* c) - 2\alpha c$$

$$x_2 = \frac{q^* c - 3\alpha \pm \sqrt{(q^* c - 3\alpha)^2 + 8 q^* \alpha c}}{2 q^*}$$

Plugging in our numbers and eliminating the extraneous solution yields:

$$\boxed{x_1 = 2.68\,\text{Mbps}, q_1 = 0.0611\,\text{s}}$$

$$\boxed{x_2 = 7.32\,\text{Mbps}, q_2 = 0.0388\,\text{s}}$$

$$\boxed{x_3 = 7.32\,\text{Mbps}, q_3 = 0.0224\,\text{s}}$$

A.5 End behavior

Let t_i denote the end time of flow i. From the numbers, we see that Flow 2 will finish first at time $t_2 = 20 + \frac{120-61.8}{7.32} = \boxed{27.95\,\mathrm{s}}$ After Flow 2 finishes, Flow 1 and Flow 3 will share the links evenly because they both know their RTT accurately. Hence,

$$t_1 = 27.95 + \frac{280 - 95 - 38.2 - 7.95 \cdot 2.68}{5} = \boxed{53.05\,\mathrm{s}}$$

$$t_3 = 53.05 + \frac{240 - 7.95 \cdot 7.32 - 5 \cdot 25.1}{10} = \boxed{58.68\,\mathrm{s}}$$

Acknowledgments

The authors would like to thank Professor Stephen Low and Cody Han for their guidance on this project.

References

[1] J. Walrand and S. Parekh, *Communication Networks A Concise Introduction*, Morgan & Claypool, 2010.

[2] C. Jin, D. X. Wei, and S. H. Low, FAST TCP: Motivation, Architecture, Algorithms, Performance, Proc. IEEE INFOCOM, Mar. 2004, http://netlab.caltech.edu