Configurable Network Simulator with TCP Reno and FAST TCP Flow Congestion Control

Rushikesh Joshi, Ricky Galliani, Tim Menninger, Schaeffer Reed
Caltech, CS 143, Fall 2015

Overview

In this project, we designed and implemented a configurable network simulator. The program takes as input a network topology, simulates flows of data through the specified network, and returns as output a series of plots that report the state of the network during the simulation. In addition to a network topology, the program also takes as input a congestion control algorithm, either TCP Reno or FAST TCP, which specifies the protocol that the flows use to establish and continuously update their window size. This value can be changed by specifying *FLOW_TCP_RENO or FLOW_FAST_TCP* for the *DEFAULT_ALG* value in the source code (*constants.py)*. The simulator has built-in support for a network consisting of packets (data, ack, and routing), hosts, half-duplex links, routers, and flows, which abstractly represent the transfer of data from one host in the network to another. The networks simulated by this program also have dynamic routing functionality — routers within the network periodically update their routing tables using the Bellman-Ford algorithm to direct the flow of packets within the network away from built-up congestion.

Design

We implemented our program with a set of classes that represent network components and a set of complementary modules that contain procedures that ease the interaction between the network objects as well as store and plot the network data collected during the simulation. The general flow of the program begins at the command-line where a user invokes the program (*simulate*) with a network topology configuration file[1] passed as an argument. A *config_parser* module then instantiates a network of programmatically interconnected objects and enqueues the events to start the flow of data within the network. During the simulation, network data is written out to a set of data files that are read when the simulation terminates for aggregation, plotting, and display for the user.

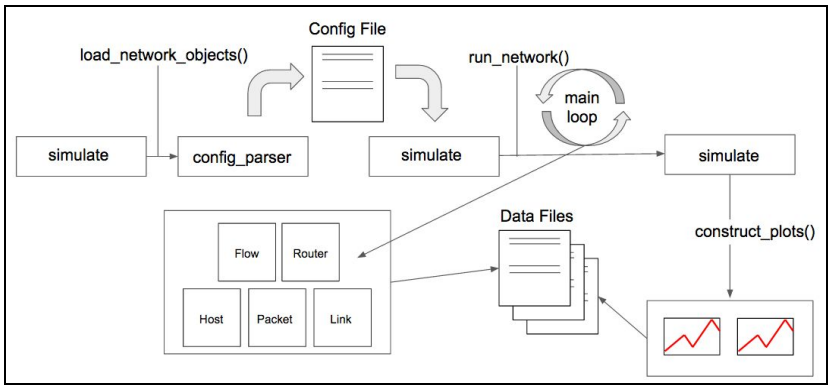

Figure 1. Diagram of the main components of the program and how they interface with each other.

Discrete Event Simulation

Our simulator was designed with the Discrete Event Simulation model in mind. In a discrete event network simulation, the complexity of the activity within the network is simplified by breaking it down into simple events and the corresponding times that the events must be executed. Within our program, network events, and the times they are to occur, are stored in a global priority queue. Typically, when an event is handled, the internal state of the network is changed in such a way that additional events must be enqueued in the priority queue as a follow up to the recent changes. In general, events represent a small interaction within the network, such as a host receiving an ack packet. Ultimately, when it is time for a particular event to be handled, the simulator executes the event, changes the internal state of the program, and enqueues any events that must be enqueued as a result of the current event. The simulation terminates when the priority queue is empty.

Flow

Most of the FAST TCP congestion control complexity within our network simulation is managed by the flow objects. After being initialized within our program, a flow is immediately in control of a dynamically updating window of packets flowing through the network. The FAST TCP algorithm utilizes three constants defined in our source code (*constants.py*). The first constant called *FAST_TCP_PERIOD* defines how often the *periodic_window_update()* function will be called when running the FAST TCP algorithm. We set this to 100 to allow for window updates every 100 milliseconds. The second constant called *ALPHA_VALUE* defines what alpha value our algorithm will use when dynamically updating window size. This alpha factor varies across the different test cases. The third constant utilized by this algorithm is *FAST_TCP_TIMEOUT_FACTOR* which is used in the calculation of our RTT, and thus implicitly used in the window update function of FAST TCP as well. When running the FAST TCP congestion control algorithm, the flow updates its window size by performing a periodic window size computation,

$$W(next) = W(current) * (\frac{Min\ RTT}{Avg\ RTT}) + \alpha.$$

Therefore, when the current packet delay is large with respect to the minimum packet delay recorded for this flow, the window size decreases because it is assumed that the packets currently flowing through the network are having to deal with more congestion. Conversely, when the current average packet delay is roughly equivalent or perhaps less than the minimum packet delay, the window size increases. In addition to directly managing window size for FAST TCP networks, flow objects can also create packets for flows, and enqueue a job to resend every in-flight packet, in the event that a packet was dropped.

[1] Consult the source code documentation for information about the accepted format for input network configuration files.

Router

At all times, a router object within a network simulated with this program manages two routing tables, an official routing table and an updating routing table. They update their routing tables by computing the shortest distances (in time) to other routers within the network in a non-telepathic manner with the periodic distribution of routing configuration packets and the Bellman-Ford shortest path algorithm. The primary procedures associated with router objects are packet reception, packet sending, and routing table updates.

Link

The implementation of link objects within this simulator program was complex because of their half-duplex design. That is, the links within the networks simulated by this program were restricted to only carry data in one direction at any given time. Therefore, link objects are required to manage when to efficiently dequeue packets from their buffers in order to control the flow of data and equitable use of the link. A simple approach to this problem is implemented for this program. When the link is "instructed" to transmit a new packet, it simply dequeues the packet that has been enqueued for the longest amount of time, regardless of whether dequeuing that packet will require the flow of data down the link to change direction, a switch that will incur a time delay.

Host

Host objects within this program manage the sending of packets (both data and ack), the receiving of packets (both data and ack), as well as checks for ack timeouts. Hosts handle the complexity built into the TCP Reno algorithm, which increments the window size of the flow using the host with the reception of ack packets. The only configurable option for the TCP Reno algorithm is a value called *RENO_TIMEOUT_TIME* which can be found in our source code (*constants.py*). We have this value set to 500, which means that in the event a packet loss is detected either by a timeout event, or by receiving three duplicate acknowledgements, all other dropped packets will be ignored for a time period of 500 milliseconds to avoid too many successive window size updates.

Packet

This network simulator program uses a packet object to represent data packets, ack packets, and routing packets, the pieces of information passed throughout a network. There is essentially no complexity built into this implementation of a packet other than that a packet object carries some data (an integer) and has a fixed size depending on its type.

Test Case 0

Flow Source / Destination	H1 / H2
Flow Data Size	20 MB
Flow Start Time	1.0 s
Link Rate	10 Mbps
Link Delay	10 ms
Link Buffer Size	64 KB

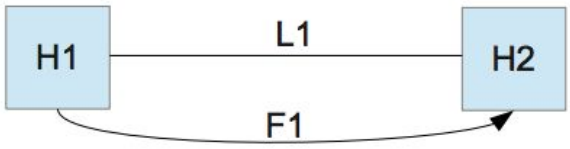

Figure 2. The input network specifications and network design diagram for Test Case 0.

Description

Test Case 0 is a test of the basic functionality of the components within a network. This test case tests our implementations of hosts, links, and flows, as well as both congestion control algorithms.

Discussion of Results

For test case 0, we observe the expected behavior for TCP Reno, as evidenced by the window size plot. As expected, we see a rapid increase in window size which corresponds to the slow-start phase, and then once a packet loss is detected, we see the window size decrease to threshold around 64 packets, the approximate size of the link buffers, when it enters the congestion avoidance phase. In this phase, the threshold is set around 64, and thus we see the window size oscillate between the threshold, and half of that which is around 32. This corroborates the implementation of our TCP Reno algorithm, because once the slow-start threshold is reached, the window size is supposed to be halved before it starts increasing again. This also validates our fast retransmit, as we avoid going back to slow-start phase, unless there is a packet timeout. The number of duplicate acknowledgements used to measure packet loss in our implementation of this algorithm was three.

In our plots for FAST TCP, we also see the behavior expected. The window size increases rapidly in the beginning, before the round trip times stabilize, and then we see an oscillation of window size between around 40 and 70. This could be tightened by increasing the sampling rate at which data is being averaged. The buffers fill as expected with acknowledgement packets and data packets, but since test case 0 is a pretty trivial case, the results are not as significant.

ALPHA_VALUE (*constants.py)*	***FAST_TCP_TIMEOUT_FACTOR*** (*constants.py)*
8	2

TCP Reno (Test Case 0)

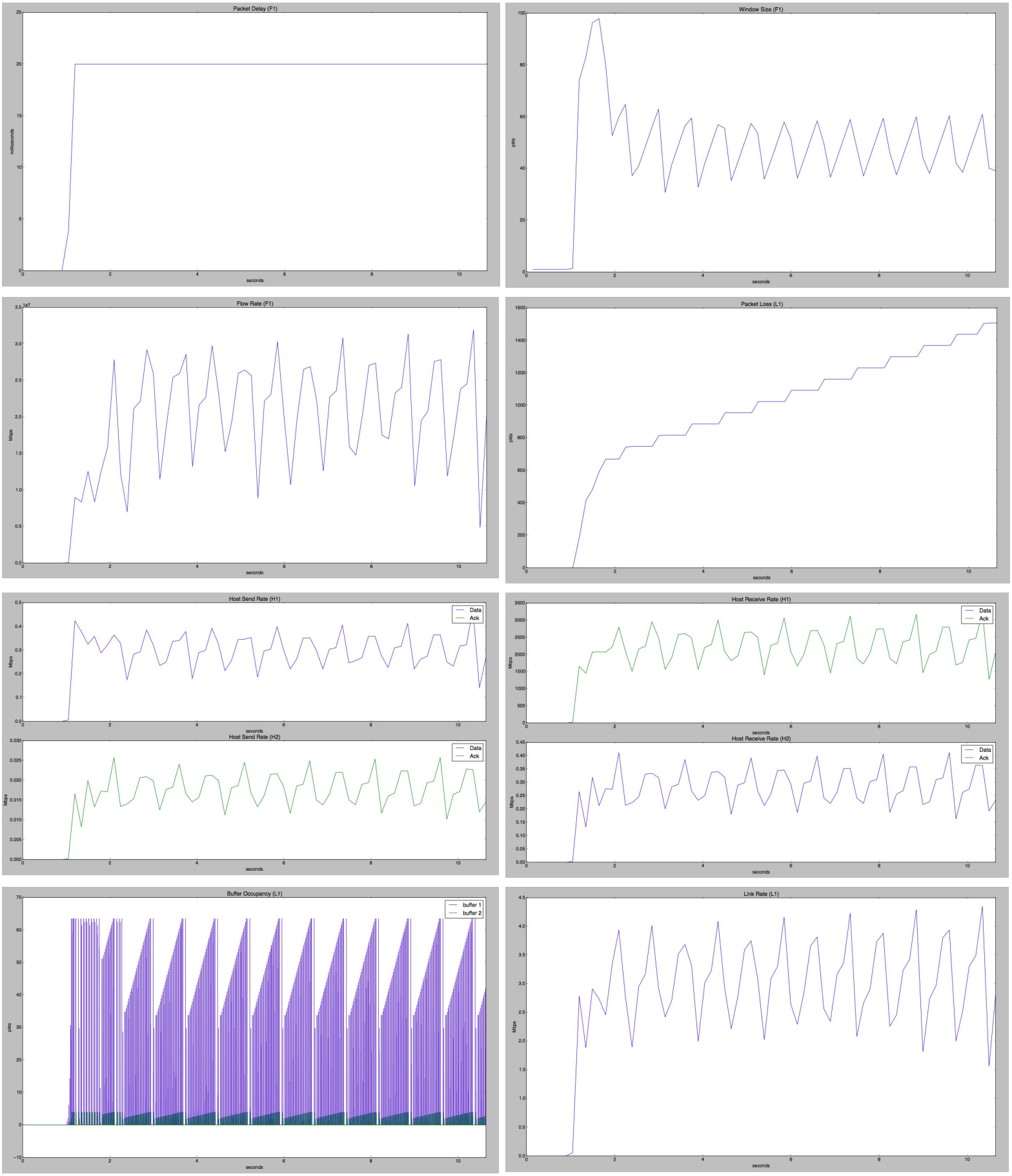

Figure 3. The output plots for the Test Case 0 simulation when the flow is configured to run the TCP Reno congestion control algorithm. The plots included are packet delay, window size, flow rate, packet loss (per-link), host send rate (per-host), host receive rate (per-host), buffer occupancy (per-host, per-buffer), and link rate (per-link).

FAST TCP (Test Case 0)

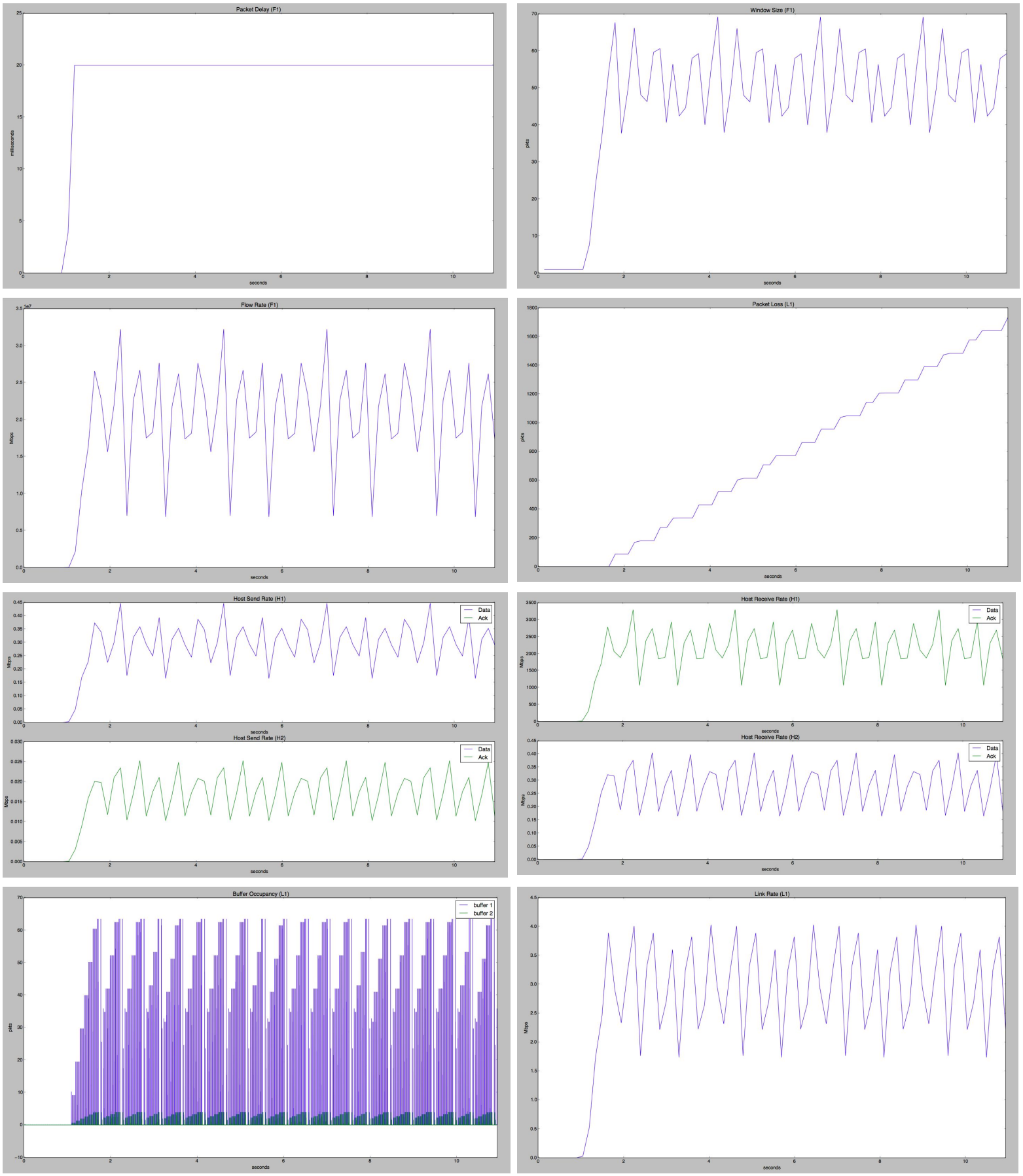

Figure 4. The output plots for the Test Case 0 simulation when the flow is configured to run the FAST TCP congestion control algorithm. The plots included are packet delay, window size, flow rate, packet loss (per-link), host send rate (per-host), host receive rate (per-host), buffer occupancy (per-host, per-buffer), and link rate (per-link).

Test Case 1

Flow Source / Destination	H1 /H2
Flow Data Size	20 MB
Flow Start Time	0.5 s
L0, L5 Link Rate	12.5 Mbps
L1-L4 Link Rate	10 Mbps
Link Delay	10 ms
Link Buffer Size	64 KB

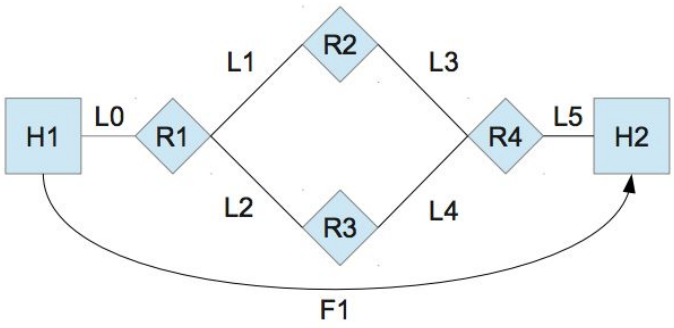

Figure 5. The input network specifications and network design diagram for Test Case 1.

Description

Test Case 1 introduced a new layer of complexity to our simulator by including routers in the network design. This case tests the implementation of the Bellman-Ford shortest path algorithm for dynamic routing built in to our network simulator.

Discussion of Results

Bellman Ford

From the network topology, one should expect router, R1, to naively route every packet to H2 through either L1 or L2, which is arbitrarily chosen at start because both would route equidistantly to H2. For argument's sake, say R1 routes every packet through L1. Therefore, no data is being sent over L2. When it updates, R1 should be able to recognize that L1 is congested and begin to reroute packets to H2 over L2. Upon closer inspection of graphs for buffer occupancy, one can find evidence of the progression of the Bellman Ford algorithm running on each individual router.

Every router ran in-place Bellman Ford every 500 milliseconds. Therefore, we should see the buffer occupancies for L1 and L2 alternate from being filled to empty every 500 ms, where if L1's buffers are occupied, L2's are not and vice versa. Looking at the buffer occupancy graphs after the flow is underway, this is exactly what we see. The buffers on L1 are filled partially or fully at 5 seconds network time for 500 milliseconds and then drops promptly to zero at 5.5 seconds. Meanwhile, the buffers on L2 are empty at 5 seconds and then pick up promptly at 5.5 seconds. All the while, the sum of the buffers on these two matches the shape of the buffer occupancy on L0, which further corroborates the success of Bellman Ford because in general, packets traverse L0 if and only if they traverse L1 or L2. Thus, we can conclude that our router self-configuration using Bellman Ford's algorithm operated successfully.

For TCP Reno, we once again see the proper implementation of the algorithm, as evidenced by the window size plots. Once again there is a rapid increase in window size which corresponds to the slow-start phase, and then once packed losses begin to be detected, we see the window size enter congestion avoidance, where with fast retransmit, the window size oscillates between the threshold of around 64, and half of that which corresponds to a window size of around 32. The FAST TCP plots also validate the implementation of the algorithm, as we see a rapid increase in window size, and then oscillation between 55 and 70 which corresponds with the stabilization of our round trip time.

ALPHA_VALUE (*constants.py)*	***FAST_TCP_TIMEOUT_FACTOR*** (*constants.py)*
8.5	2

TCP Reno (Test Case 1)

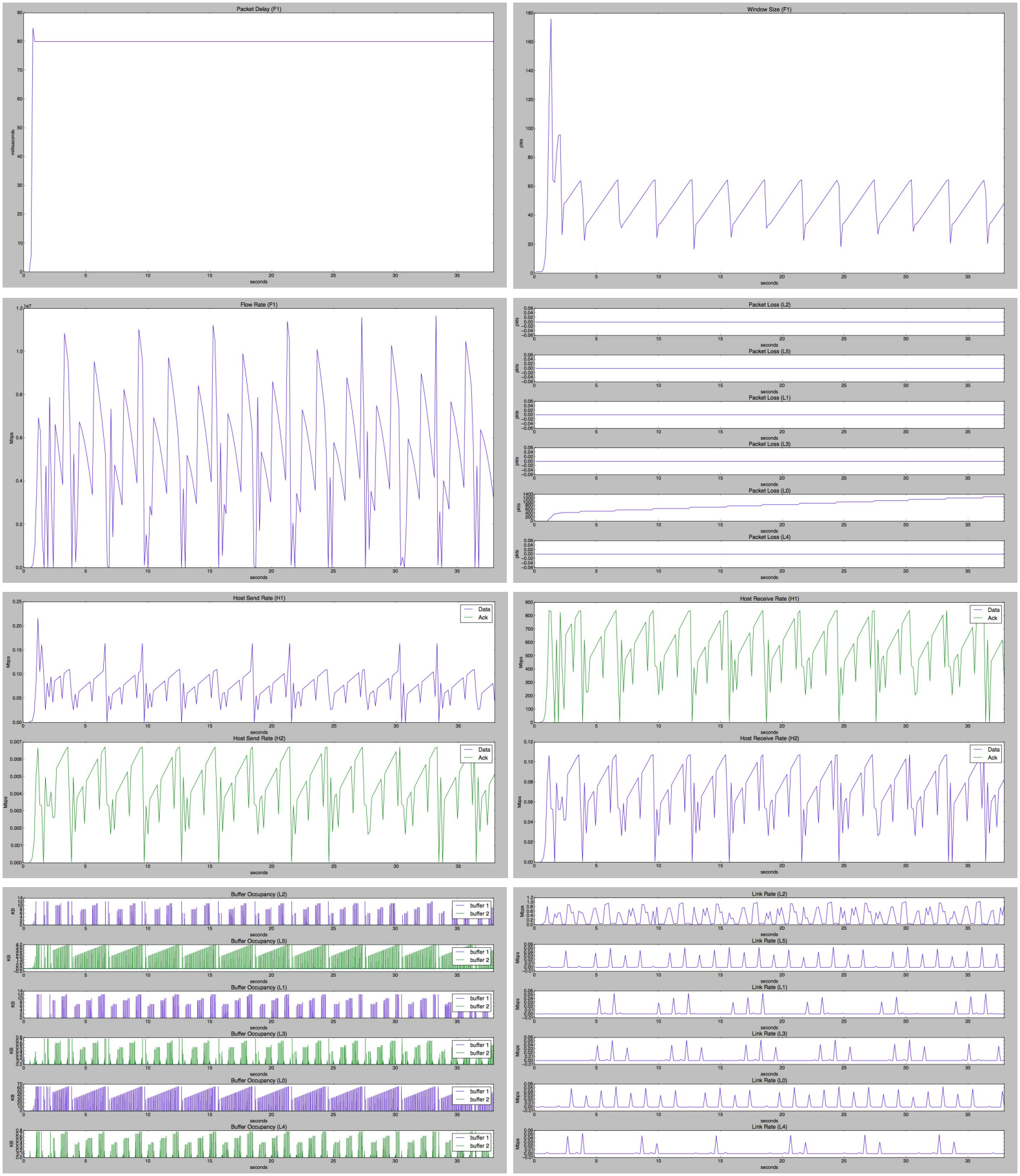

Figure 6. The output plots for the Test Case 1 simulation when the flow is configured to run the TCP Reno congestion control algorithm. The plots included are packet delay, window size, flow rate, packet loss (per-link), host send rate (per-host), host receive rate (per-host), buffer occupancy (per-host, per-buffer), and link rate (per-link).

FAST TCP (Test Case 1)

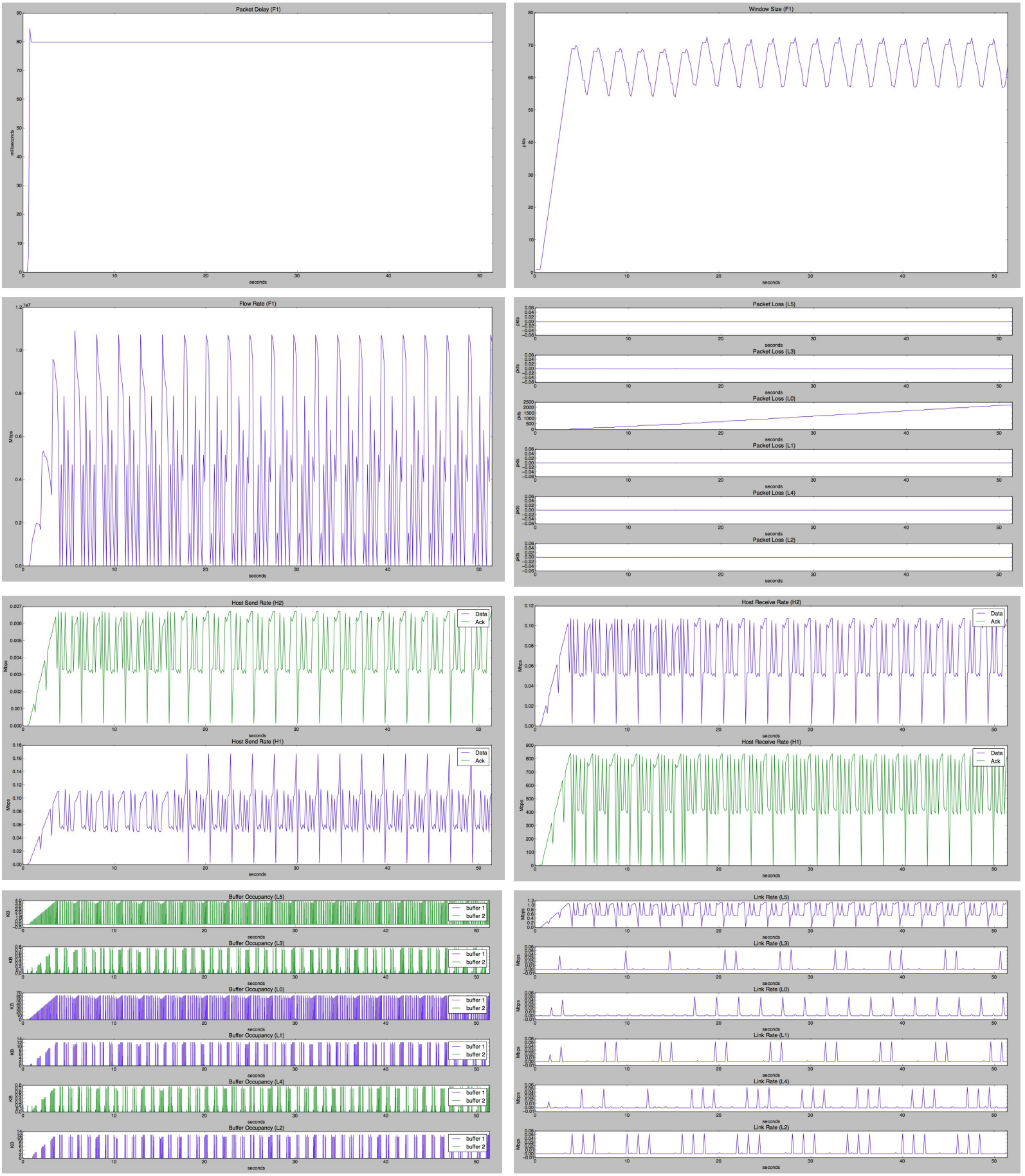

Figure 7. The output plots for the Test Case 1 simulation when the flow is configured to run the FAST TCP congestion control algorithm. The plots included are packet delay, window size, flow rate, packet loss (per-link), host send rate (per-host), host receive rate (per-host), buffer occupancy (per-host, per-buffer), and link rate (per-link).

Test Case 2

Flow 1 Source / Destination	S1 / T1
Flow 1 Data Size	35 MB
Flow 1 Start Time	0.5 s
Flow 2 Source / Destination	S2 / T2
Flow 2 Data Size	15 MB
Flow 2 Start Time	10.0 s
Flow 3 Source / Destination	S3 / T3
Flow 3 Data Size	30 MB
Flow 3 Start Time	20.0 s
L1, L2, L3 Link Rate	10 Mbps
Link Rate (All Other Links)	12.5 Mbps
Link Delay	10 ms
Link Buffer Size	128 KB

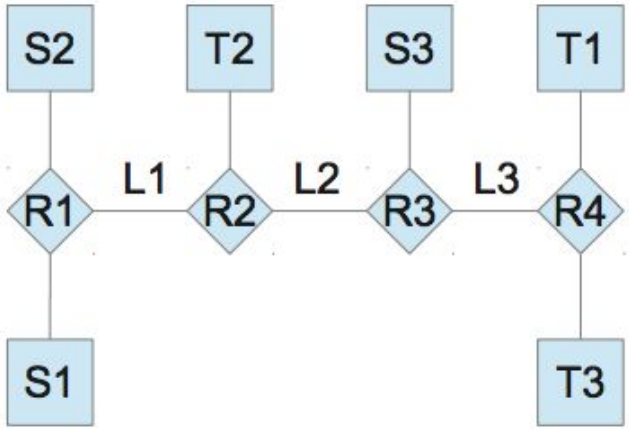

Figure 8. The input network specifications and network design diagram for Test Case 2.

Description

This test case adds even more complexity to the network. With three flows starting and stopping at different times, Test Case 2 not only tests the basic network component functionality tested in Test Case 0 and Test Case 1 and the dynamic routing functionality tested in Test Case 1, but also the simulated network's capability to simultaneously support multiple, congestion controlling flows.

Discussion of Results

Test case 2 with its added complexity, proved to be far more challenging than the other test cases. To get proper results, we tinkered with various parameters and calculations of RTT to arrive at our final iteration of the FAST TCP implementation, while TCP Reno was easier to corroborate.

As expected, we see a distinctive pattern in the window size plots for TCP Reno. In each of the flows, there is an initial increase in window size corresponding to the slow-start phase, and then the classic oscillatory behavior expected of the window size during the congestion avoidance phase. This plot corroborates our fast retransmit implementation, and passed test case 2 with the expected results.

For FAST TCP, we tested various alpha values and timeout factors (for the RTT calculation) to get the proper results. The plots for FAST TCP are not as pattern-oriented as the TCP Reno plots, but still follow the behavior expected from the FAST TCP algorithm. We see that the RTT stabilizes relatively quickly, and the sharp drops in window size correspond to sharp increases in the round trip time which result from increased congestion due to the bottle necked link in this test case.

ALPHA_VALUE (*constants.py)*	***FAST_TCP_TIMEOUT_FACTOR*** (*constants.py)*
20	2

TCP Reno (Test Case 2)

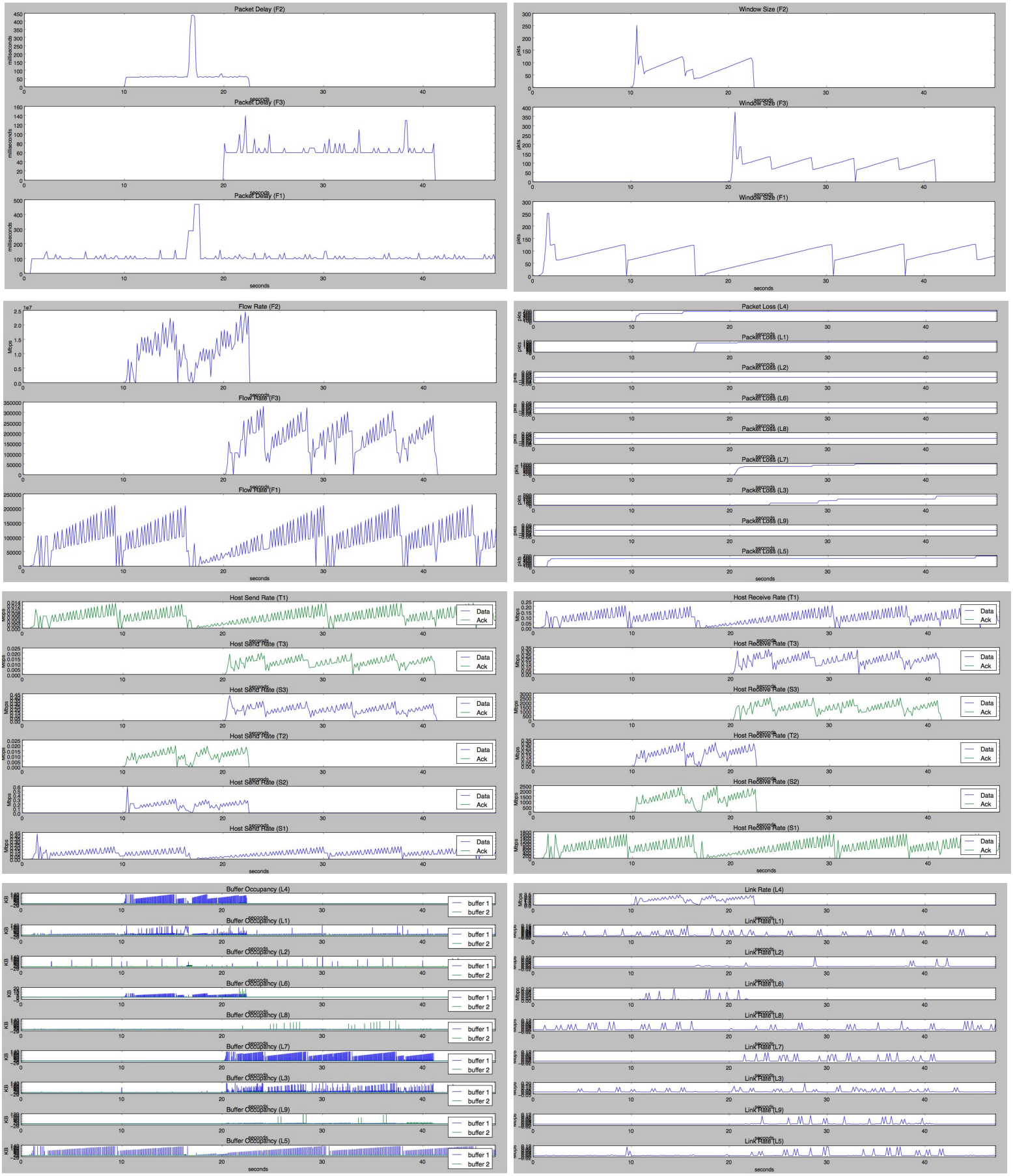

Figure 9. The output plots for the Test Case 2 simulation when the flow is configured to run the TCP Reno congestion control algorithm. The plots included are packet delay, window size, flow rate, packet loss (per-link), host send rate (per-host), host receive rate (per-host), buffer occupancy (per-host, per-buffer), and link rate (per-link).

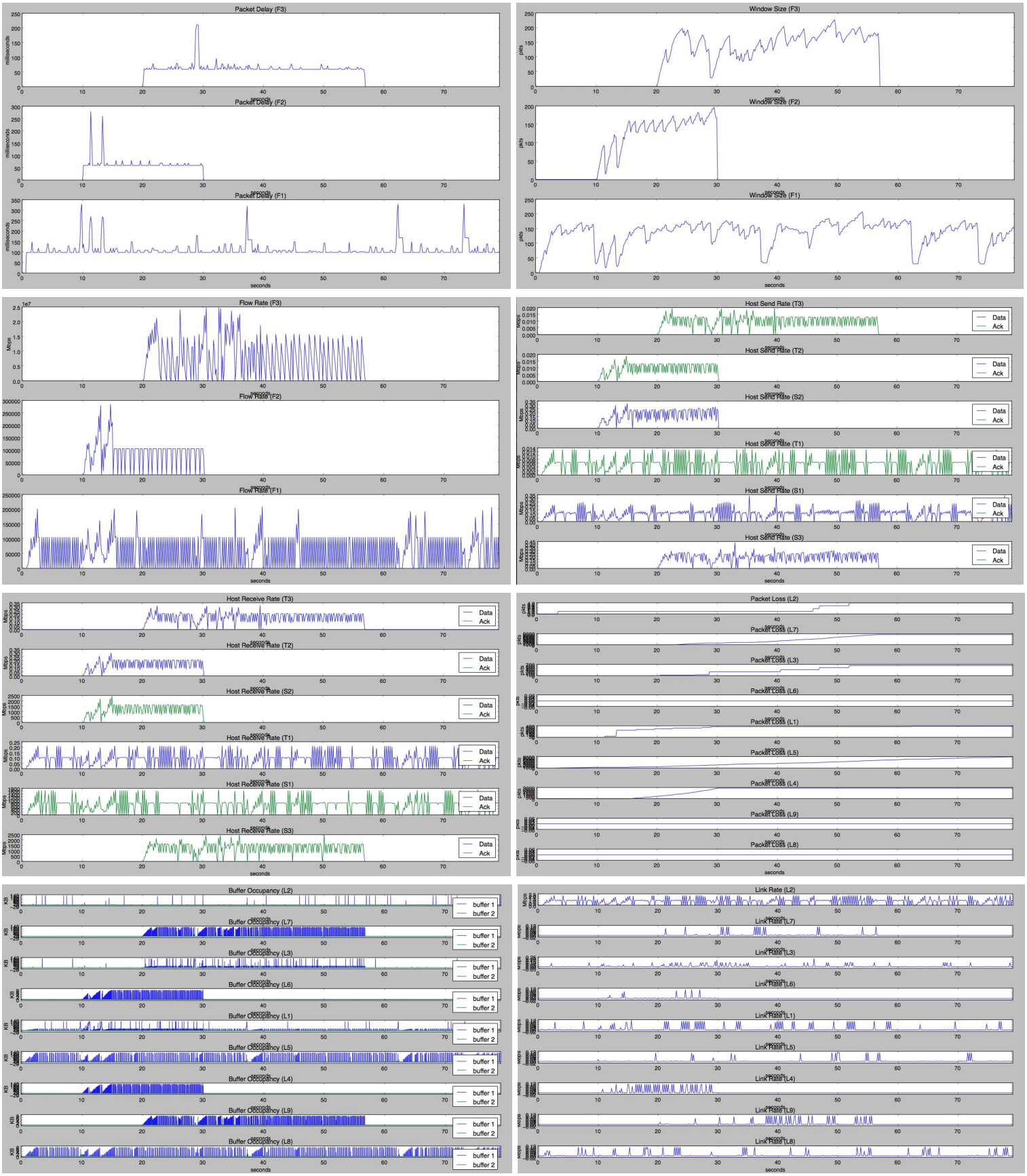

Figure 10. The output plots for the Test Case 2 simulation when the flow is configured to run the TCP Reno congestion control algorithm. The plots included are packet delay, window size, flow rate, packet loss (per-link), host send rate (per-host), host receive rate (per-host), buffer occupancy (per-host, per-buffer), and link rate (per-link).

Theoretical Results vs. Simulation Results (Test Case 2 - FAST TCP)

	0.5 - 10s (64s onwards)	10s - 20s	20s - 32.5s	32.5-64s
Measured F1 Throughput **Expected F1 Throughput**	**5 Mbps** **10 Mbps**	**7.5 Mbps** **12.5 Mbps**	**5 Mbps** **3.3332 Mbps**	**7.5 Mbps** **12.5 Mbps**
Measured F2 Throughput **Expected F2 Throughput**	**0 Mbps** **0 Mbps**	**10 Mbps** **12.5 Mbps**	**5 Mbps** **6.6664 Mbps**	**0 Mbps** **0 Mbps**
Measured F3 Throughput **Expected F3 Throughput**	**0 Mbps** **0 Mbps**	**0 Mbps** **0 Mbps**	**5 Mbps** **6.6664 Mbps**	**7.5 Mbps** **12.5 Mbps**
Measured L1 Buffer Occupancy **Expected L1 Buffer Occupancy**	**~30 pkts** **20 pkts**	**~35 pkts** **40 pkts**	**~30 pkts** **30 pkts**	**~15** **40 pkts**
Measured L2 Buffer Occupancy **Expected L2 Buffer Occupancy**	**0 pkts** **0 pkts**	**0 pkts** **(0 pkts)**	**0 pkts** **0 pkts**	**0 pkts** **0 pkts**
Measured L3 Buffer Occupancy **Expected L3 Buffer Occupancy**	**0 pkts** **0 pkts**	**0 pkts** **0 pkts**	**~40 pkts** **30 pkts**	**~40 pkts** **40 pkts**

Conclusion

Having completed this simulator, we noticed that FAST TCP consistently runs faster than TCP Reno. An exception to this might be Case 0, but Case 0 is somewhat trivial and should run about the same regardless of congestion control. Should we have continued, we would have also implemented algorithms such as TCP New Reno and TCP Vegas. This may help us further gauge what congestion control algorithms work best. We may also consider more intricate networks as some congestion control algorithms may be better suited under some conditions than others.

If we were to expand on this simulation, we may want to take into consideration processing delay, the time it takes a router or host to process the packet and react accordingly. In the network topologies we considered, this was too trivial to consider. However, if we were to consider larger networks in which flows traversed many routers from host to host, processing delay may become nontrivial.

Additionally, if we were to continue, it may be interesting to experiment with FAST TCP and how timed out packets are treated. Currently, we average packet round-trip times to update window size. However, when there is a timed out packet, the round trip time is not defined and we therefore have to define one. We found that how we handled this affected the outcome drastically and tried numerous ideas before agreeing that round-trip average “penalty” should rely on the current window size.

Finally, the way we plot our results is not as perfect as we would like. The shape of the plots are generally what we expect, but there seems to be some conversion errors causing our axes to display the wrong scale on some plots. Because the shapes were consistent with our expectation, we put this bug off to focus on congestion-control issues and ended up not having time to revisit the plotting bugs. Nonetheless, working on this simulator was a great collaborative experience, as we were able to successfully divide tasks that came together in the end. We learned how to effectively use version control to keep track of incremental changes, and ultimately, became more familiar with network intricacies and congestion control algorithms as well.

Acknowledgements

This project would not have been a success if not for the input and guidance of Ritvik Mishra (Caltech ‘16, Computer Science) throughout the design and development processes. Special thanks to Professor Steven Low as well, for teaching us in CS143. Please find our contact information below and feel welcomed to contact us for inquires about the project or permission to view and/or fork the source code.

Ricky Galliani	**Computer Science/Economics ‘17**	**pjgalliani@caltech.edu**
Tim Menninger	**Computer Science ‘17**	**tmenninger@caltech.edu**
Rushikesh Joshi	**Computer Science ‘16**	**rsjoshi@caltech.edu**
Schaeffer Reed	**Computer Science ‘17**	**jreed@caltech.edu**