CS/EE 143 Term Project: Network Simulator

Sharjeel Aziz, Chris Hazard, Ying-Yu Ho, Emily Shih

Introduction

The goal of the project was to implement a fully functional abstract network simulator. The program was required to take a network as an input, run the network simulation, record data about the simulation, and output graphs of the data for each run.

We were required to have working links, hosts, and routers, as well as flows that used congestion control.

To satisfy these requirements, we implemented the Bellman-Ford algorithm for dynamic routing and the Dijkstra algorithm for static routing (as well as Dijkstra for both static and dynamic through Sonar/Echo packets), and implemented TCP-Tahoe, TCP-Reno, and FAST-TCP for congestion control.

Methods

Development tools

All code was written in Python 2.7. The SimPy simulation environment was used to time and store events for all components in the network. GitHub was used for code sharing. Hackpad was used for tracking weekly tasks, communication between team members, and the architecture.

Architecture

Device

There is a base class Device implementing methods to

- add_port: connect to another device.
- send: send a packet to another connected device.
- send_except: broadcast a packet to all connected devices, optionally excluding one.

Additionally, Device defines an interface

- receive: actions upon receiving a packet.

Subclasses of Device includes

- Host: Interacts with flows to send data packets. Acknowledges incoming data packets.
- Link: Full-duplex. Forwards incoming packets to the other end. Has a buffer on each end. Models queuing and transmission delay.
- Router: Forwards data and acknowledgement packets. More complicated interaction with routing packets.

Packet

A base class Packet provides two main interfaces

- reach_router: visitor method call by the router receiving this packet.
- reach_host: visitor method call by the host receiving this packet.

The purpose is to achieve double dispatch (Device and Packet) without many conditional statements, making adding more types of packet relatively effortless.

Subclasses of Packet include

- DataPacket: Data to be sent. Each has size 1024 KB, ignoring header.
- AckPacket: Proof of receiving data packet.
- SonarPacket: Network discovery for Dijkstra routing.
- EchoPacket: Table construction for Dijkstra routing.
- RoutingPacket: Mixed Dijkstra/Bellman-Ford routing.

Flow

This is base class BaseFlow implementing basic functionality of a TCP flow. Another base class FlowState provides interfaces to model transition between congestion control states and details of various algorithm. A subclass of BaseFlow takes a dictionary of FlowState constructors to complete a congestion control algorithm.

BaseFlow defines interfaces

- event_timeout: Handler of timeout.
- event_dupack: Handler of duplicated acknowledgement.
- event_ack: Handler of normal acknowledgement.

Select implementation details

Link

Links were full-duplex, and thus were implemented using two BufferedCable objects, which represented a one-way connector between two devices, usually a host and a router. These BufferedCable objects were implemented to keep track of the buffers that were needed for packets being sent each way along a link.

Routing

We implemented two different ways to find the routing table. The initial method was to find a static routing table based on the time it takes for a host to send a routing packet to the closest router(s). However, since this table only showed the best way to go from host to router, it was not the best path finder. To fix this, we implemented the Bellman-ford method. The routers send out packets to all of the ports they are connected to and the packets come back with data. This data could be from a host(in which case, it will contain the time it took to get to the host and the host ID) or from a router(in which case, it will contain the tables for that router). When a packet arrives back to the initiating router from a router, we go through the table and add the hosts that were not in the initiating router(along with the port ID it came back from) and we change the port IDs of specific hosts for which it is faster to send the packet through that port ID.

However, though this method worked pretty well, we were not able to get the best graphs. Hence, we implemented a dijkstra method to explore the system and find the best paths through the system from each host. We used sonar/echo type packets. The host sends out packets(sonar packets) and other hosts send back an echo packet. Hence, we were able to dynamically keep updating the routing table based on when the echo packets return.

Flow

The base Flow class was responsible for sending data packets in the proper order, maintaining the transmission window, detecting timeouts, and correctly processing acknowledgement packets. Flow was implemented by dividing the simulation into the four states: Slow Start, Congestion Avoidance, Fast Retransmission, and Fast Recovery. We ended up with a Flow class that was extremely flexible, where all it took to implement a new congestion control algorithm was changing one or two of the state subclasses. This was a great improvement over our initial design, where the entire flow process was rewritten for

each type of flow, which resulted in a lot of repeated and disorganized code.

Graphics

We implemented graphics by creating a graphics object with an update function. Graphics object is responsible for making the gui that graphs the information about the network. The update function based on the input.update_string, which contains the information that is updated to the graphs. We pip in the updates after running the simulation. We also attempted a guigenerator object(for extra credit). It displays a gui that helps you build a network and prints it out to a user specified file.

Additionally, we wrote a program saveplot.py that generates plots as PNG files.

User Guide

Terminal commands

```
python networking.py sim_time [flow_alg=fast] < testcase.txt | python
process.py [point_per_sec=5] | python graphics.py

python networking.py sim_time [flow_alg=fast] < testcase.txt | python
process.py [point_per_sec=5] | python saveplot.py output [maxtime]

flow_alg = tahoe, reno, fast, or cubic

output = plot.png or "-" for displaying plots on screen
```

Test case format

A test case file has five sections separated by a link with single “-” (not an underscore “_”) as described below.

```
====Start of File====
[Host ID]
(more hosts)
-
[Router ID]
(more routers)
-
[Link ID] [Device ID] [Device ID] [Rate (Mbps)] [Delay (ms)] [Buffer (KB)]
(more links)
-
```

```
[Flow ID] [Source] [Target] [Data (MB)] [Start (s)]
(more flows)
-
[Type of Plot] [Device ID to Plot] [Device ID to Plot] ...
(more plots)
====End of File====
```

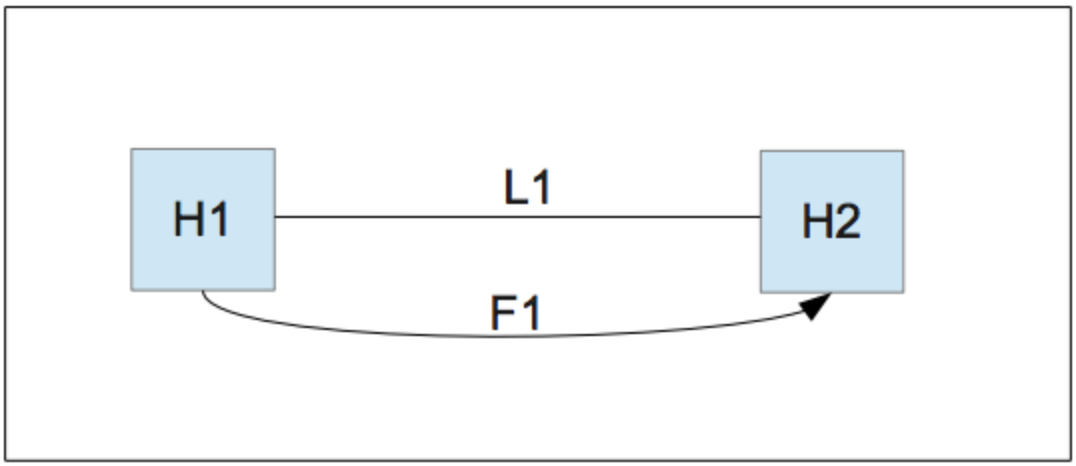
Sample test case

```
====Start of File====
S1
S2
S3
T1
T2
T3
-
R1
R2
R3
R4
-
L1 R1 R2 10 10 128
L2 R2 R3 10 10 128
L3 R3 R4 10 10 128
L4 R1 S2 12.5 10 128
L5 R1 S1 12.5 10 128
L6 R2 T2 12.5 10 128
L7 R3 S3 12.5 10 128
L8 R4 T1 12.5 10 128
L9 R4 T3 12.5 10 128
-
F1 S1 T1 35 0.5
F2 S2 T2 15 10
F3 S3 T3 30 20
-
link_flow_rate L1 L2 L3
buf_level L1 L2 L3
packet_loss_rate L1 L2 L3
flow_send_rate F1 F2 F3
window_size F1 F2 F3
packet_rtt F1 F2 F3
====End of File====
```

Results

All simulations used Dijkstra's algorithm for dynamic routing with a 5-second interval.

Test case 0

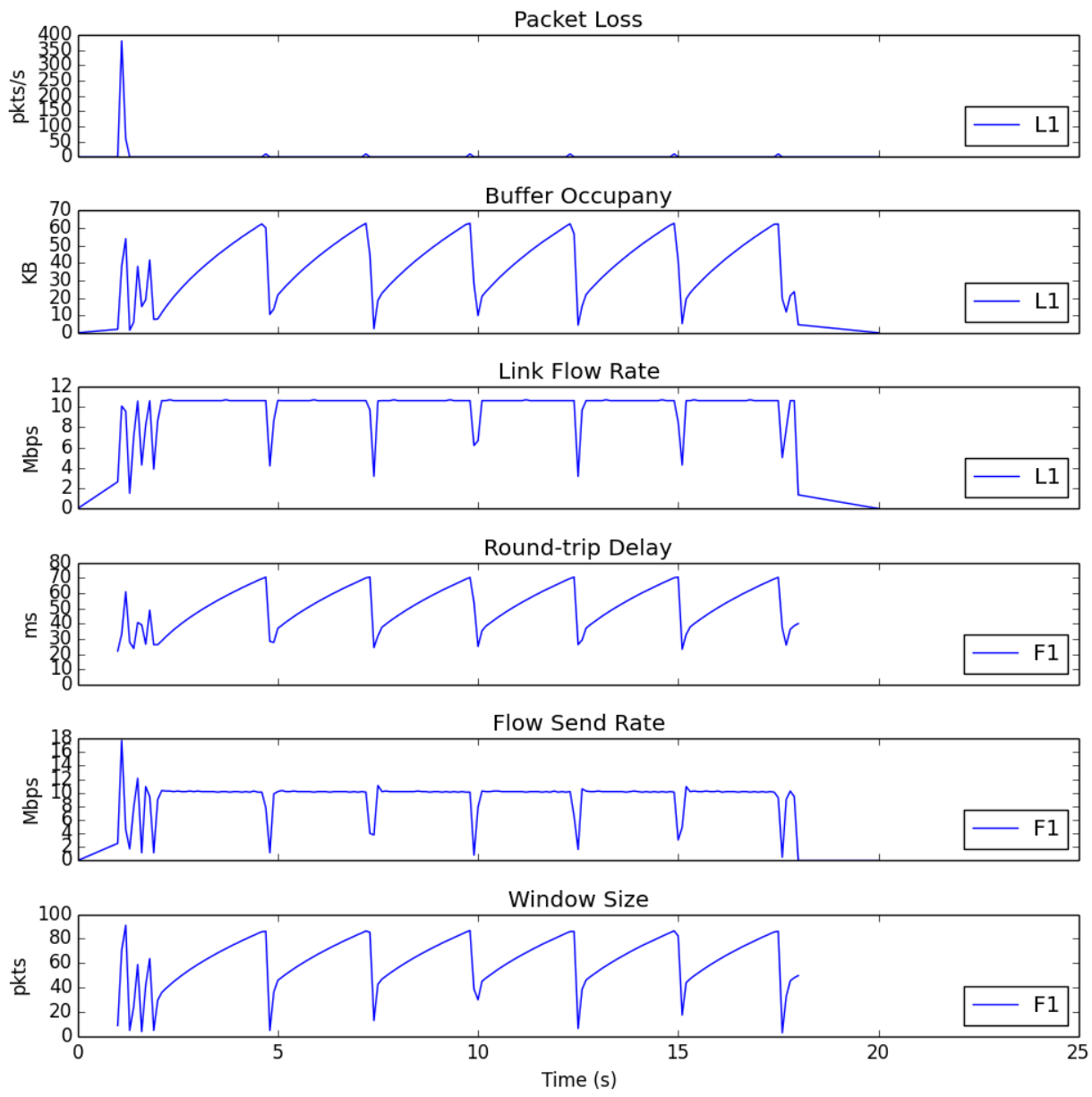


Analysis

For test case 0, we have the simplest network: one flow that uses one link to send packets from one host to another.

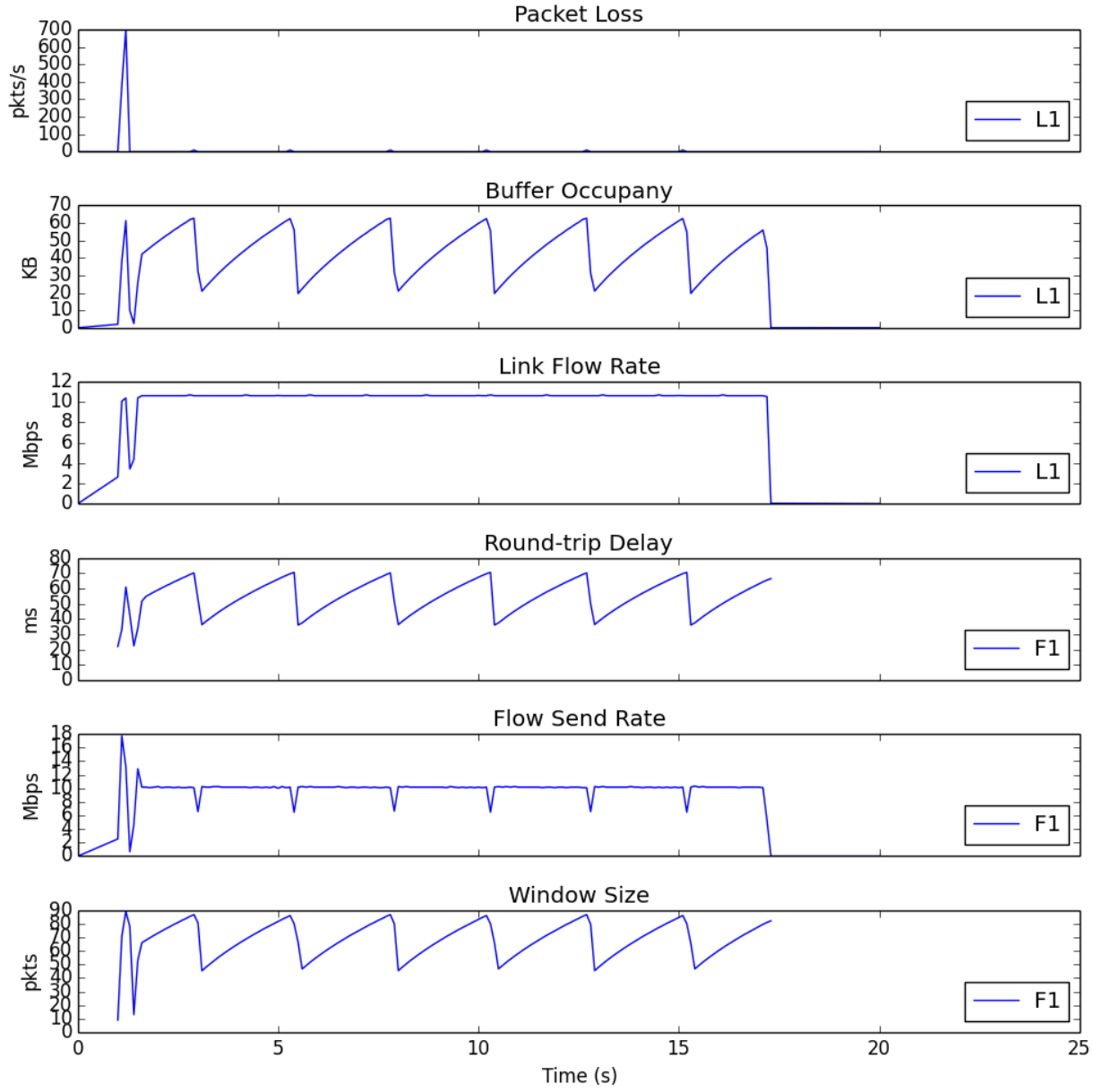
Thus, since there will be no interference from additional links or additional flows, we expect no packet loss, and a relatively constant link rate throughout the simulation. Patterns will be consistent for the other collected statistics as well.

TCP Tahoe



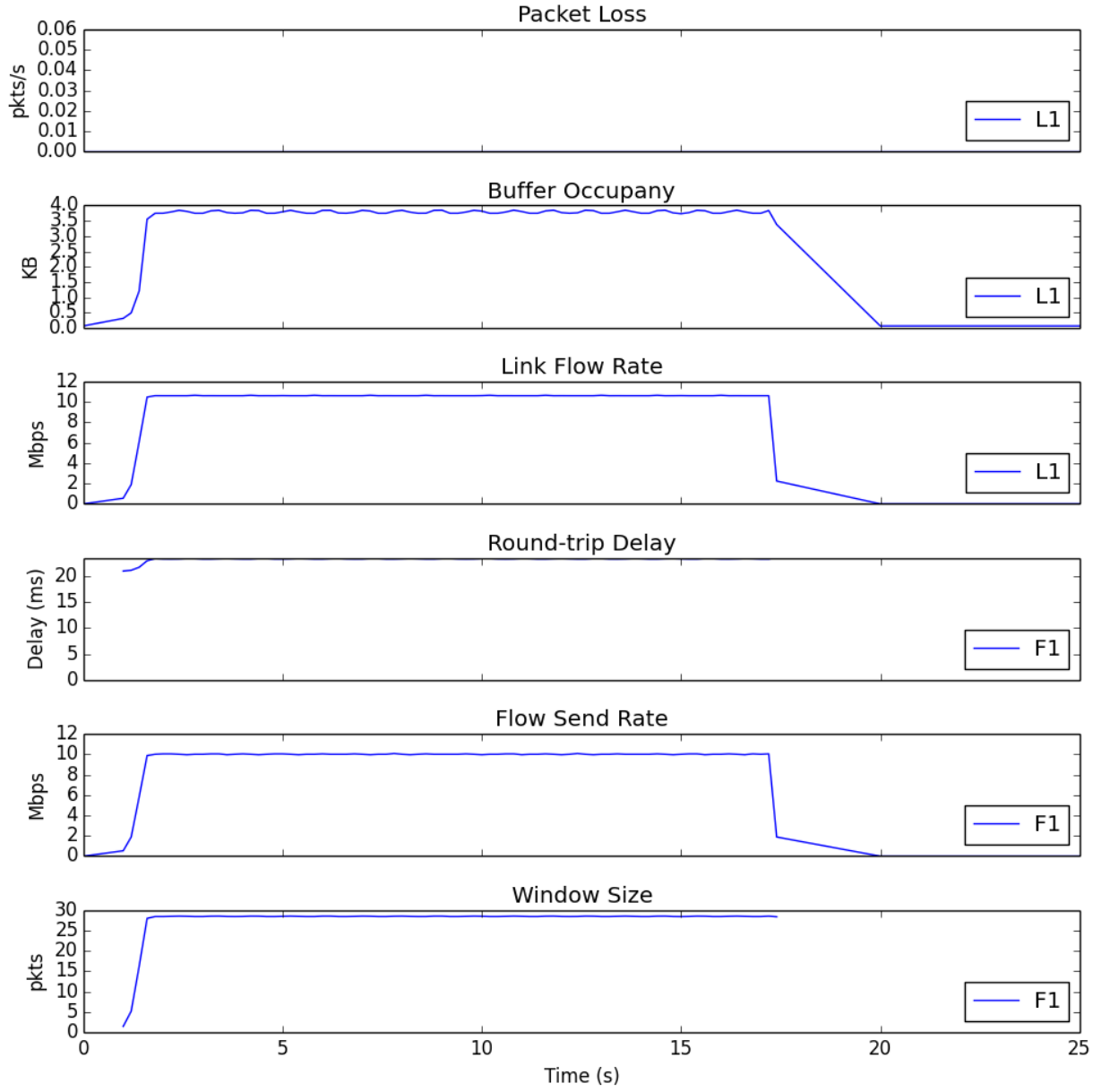
For test case 0, our implementation of TCP-Tahoe appears to be correct. With no conflicting flows, the window size behaves as we would expect for Tahoe. Flow rate stays mostly constant for the single flow.

TCP Reno



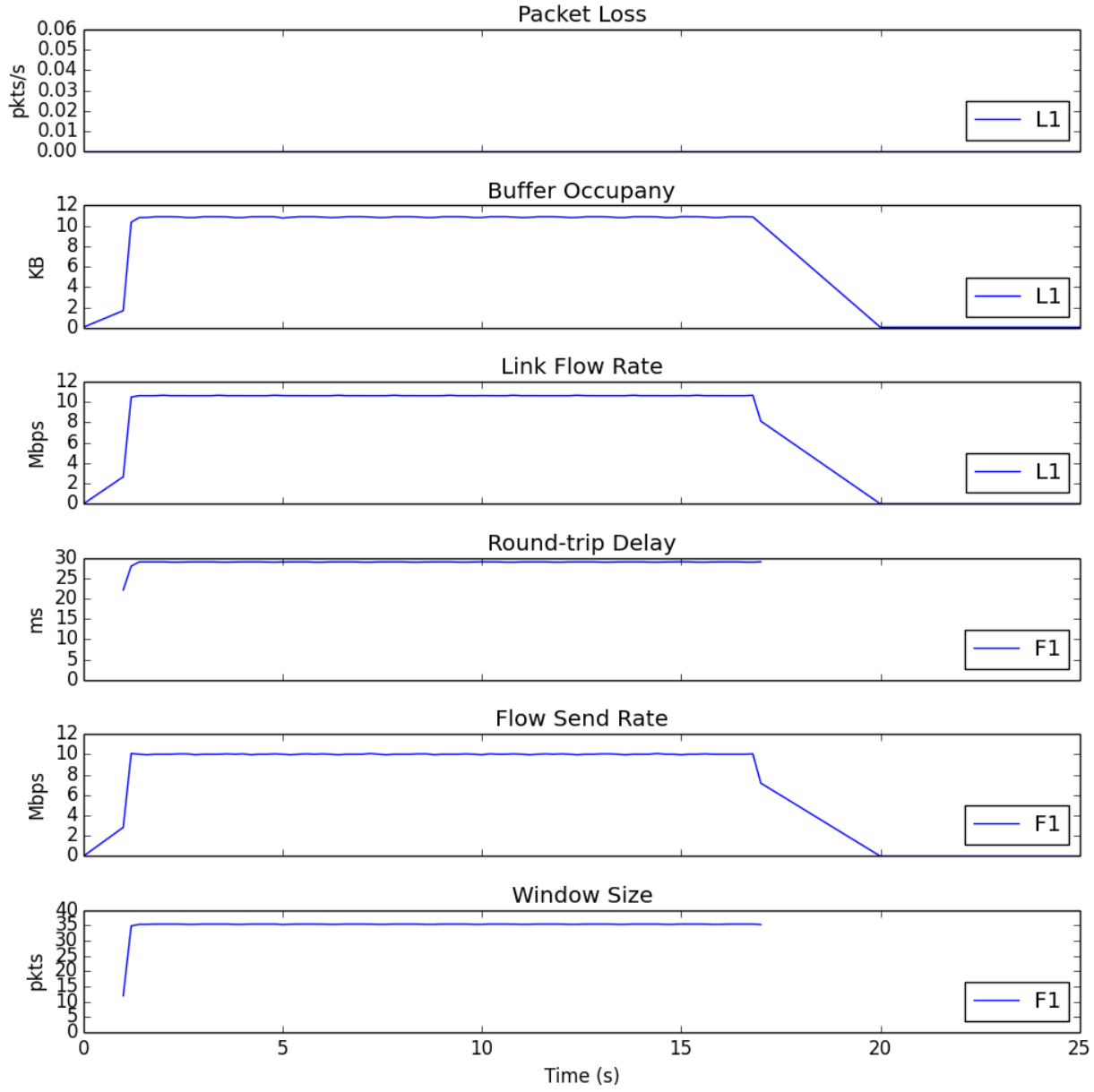
For test case 0, our implementation of TCP-Reno appears to be correct. With no conflicting flows, the window size behaves as we would expect for Reno. Flow rate stays constant for the single flow.

FAST-TCP (α = 3)



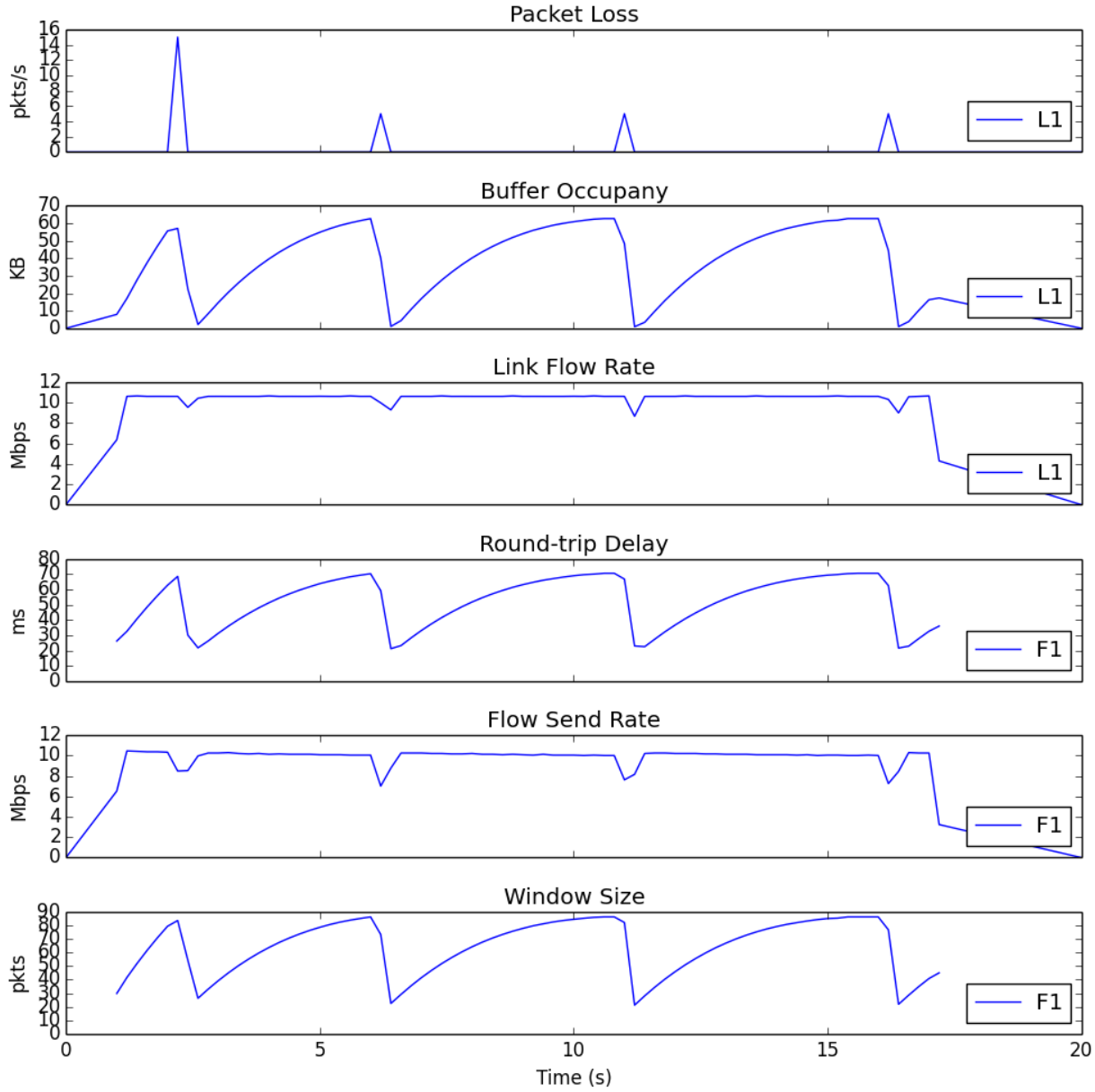
For test case 0, our implementation of FAST-TCP appears to be correct. Since there is no interference from other links or flows, all data statistics stay constant once they reach their maximum capacities.

FAST-TCP (α = 10)



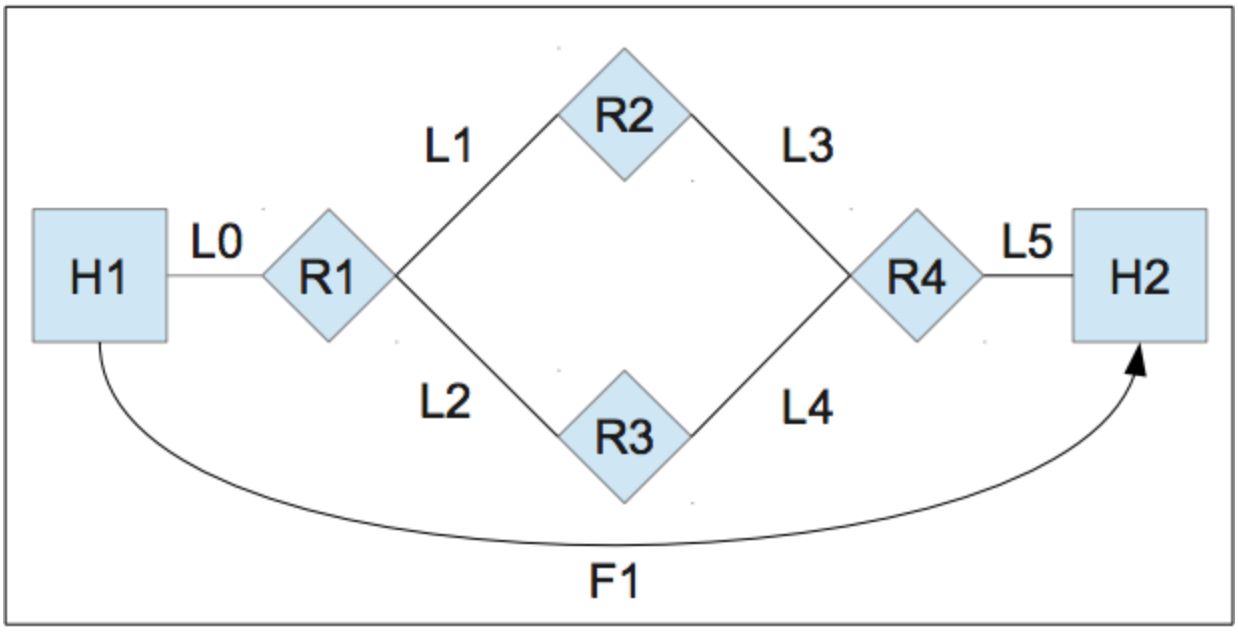
When α = 10, we see the same pattern as before, so we can assume that our FAST-TCP was implemented properly. For this higher α value, buffer occupancy and window size increase, as expected.

CUBIC TCP



For test case 0, our implementation of CUBIC-TCP appears to be correct. The window size follows the curve that we expect to see for CUBIC.

Test case 1

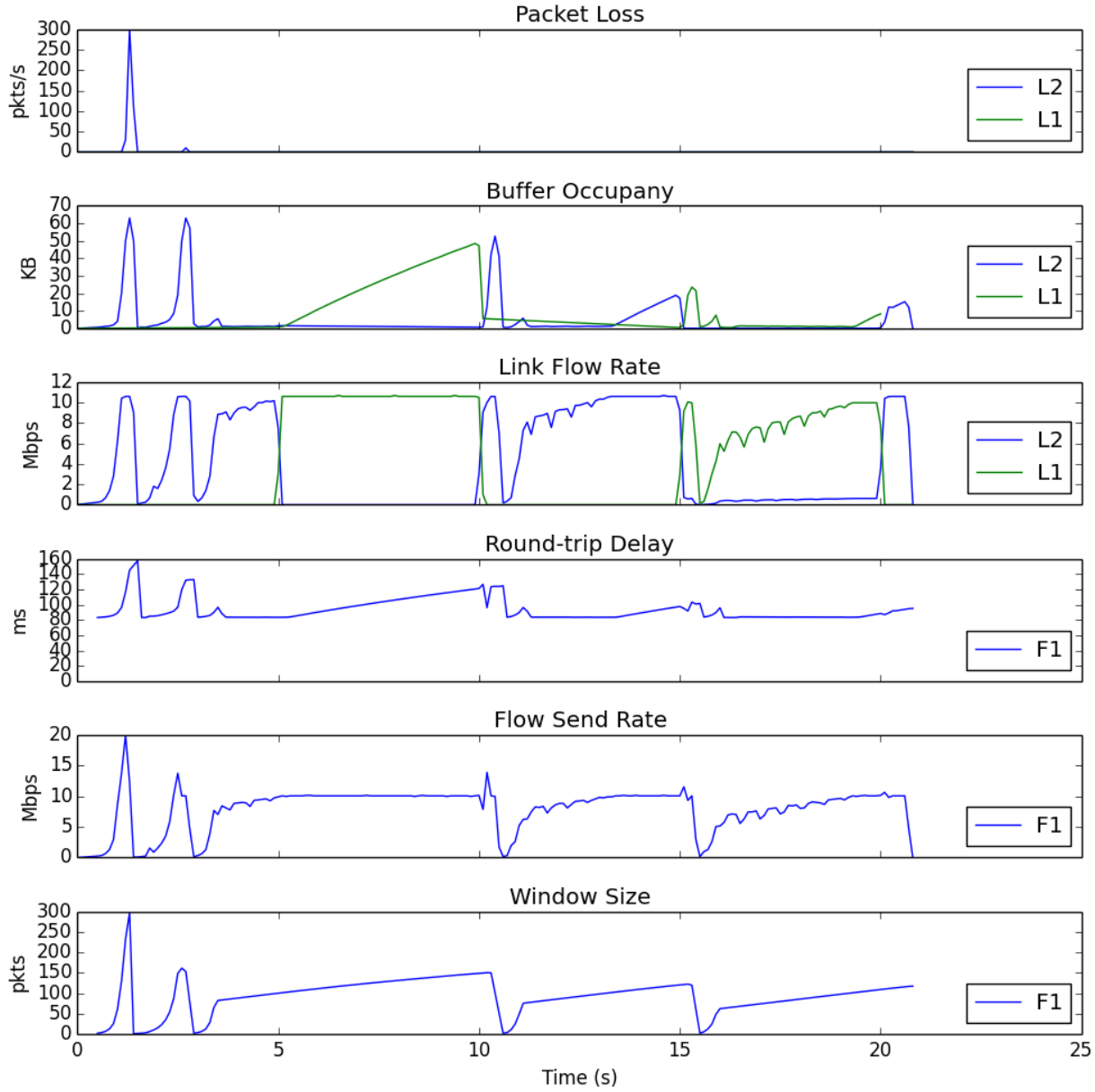


Analysis

For test case 1, we have one flow sending data from one host to another. In this case, there are two paths that data can be sent on: one involving L1 and the other involving L2.

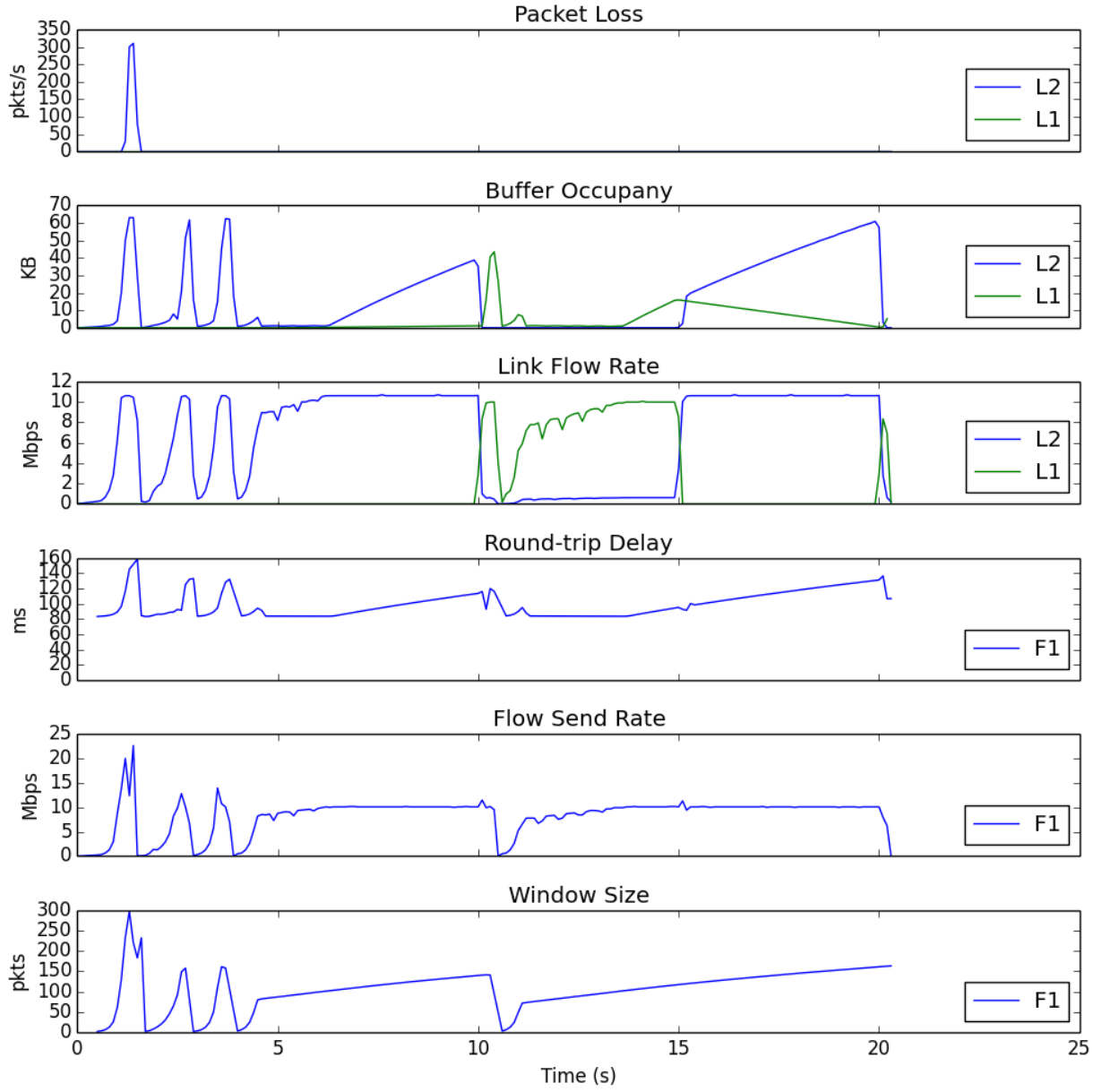
Thus, we expect to see no packet loss and a flow rate switch between L1 and L2.

TCP Tahoe



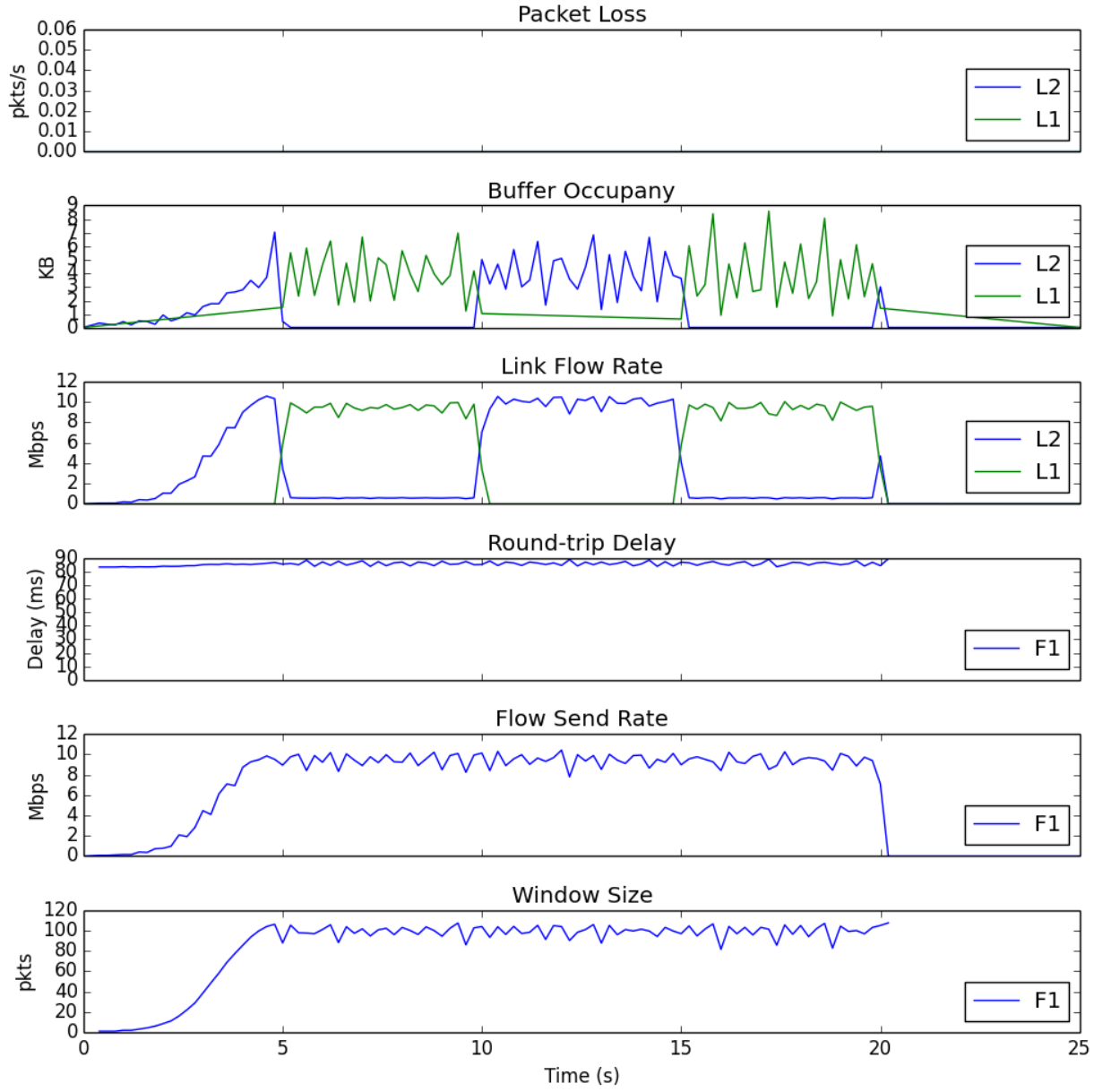
For test case 1, our implementation of TCP-Tahoe appears to be correct. From the buffer occupancy and especially the link flow rate, we see that of the two paths that each use one of the links, the network only uses one at a time to send packets. Consequently, the increase in round-trip delay and decrease in flow send rate make sense. The window size still shows the appropriate behavior for TCP-Tahoe.

TCP Reno



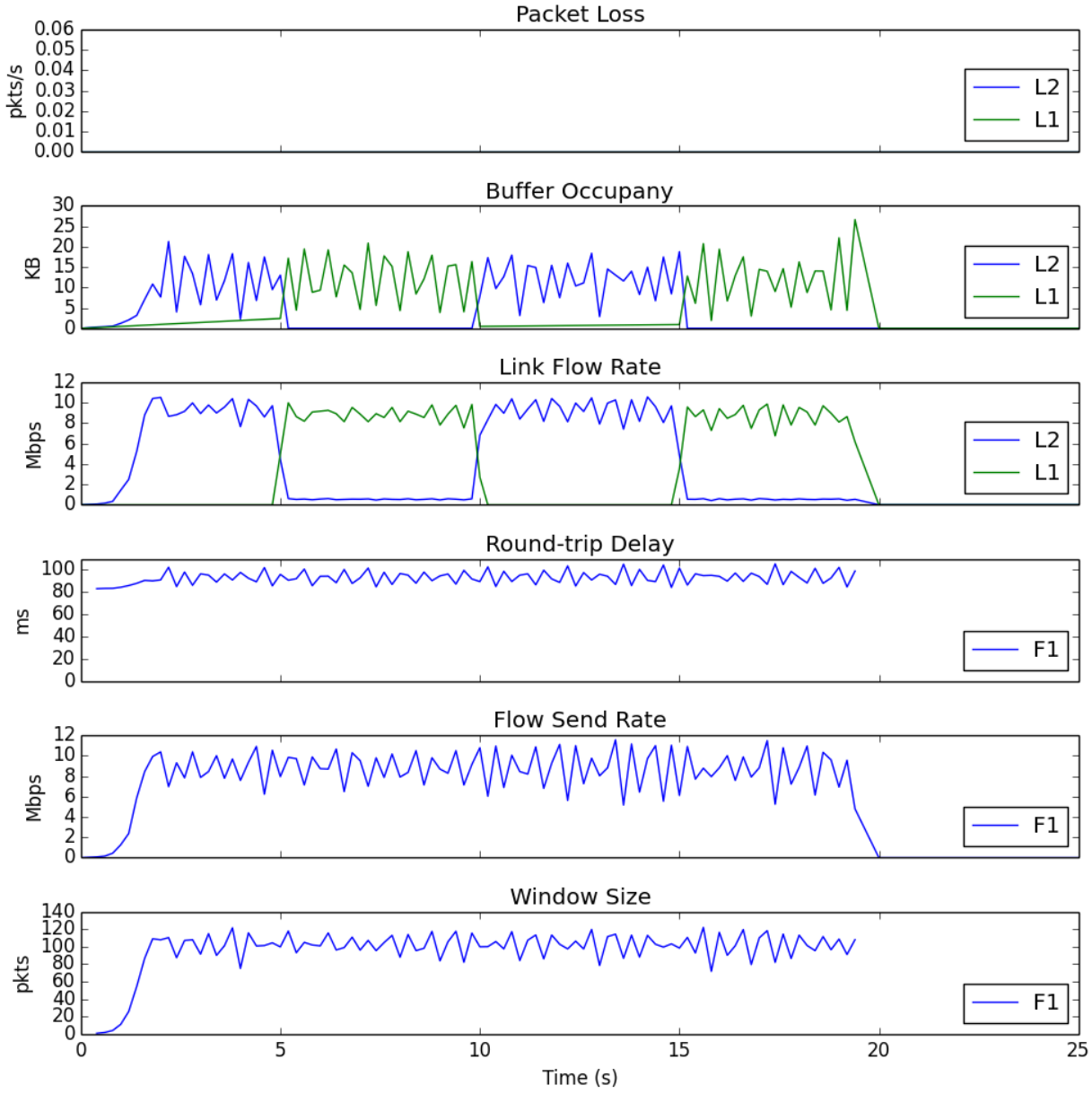
For test case 1, our implementation of TCP-Reno appears to be correct. From the buffer occupancy and especially the link flow rate, we see that of the two paths that each use one of the links, the network only uses one at a time to send packets. Consequently, the increase in round-trip delay and decrease in flow send rate make sense. The window size shows the appropriate behavior for TCP-Reno, and here we can see a difference from TCP-Tahoe due to the difference in update step. Since there is still only one flow, it makes sense that TCP-Tahoe and TCP-Reno behave similarly. We also observe that our results very much resemble the results given in the sample test cases.

FAST-TCP (α = 3)



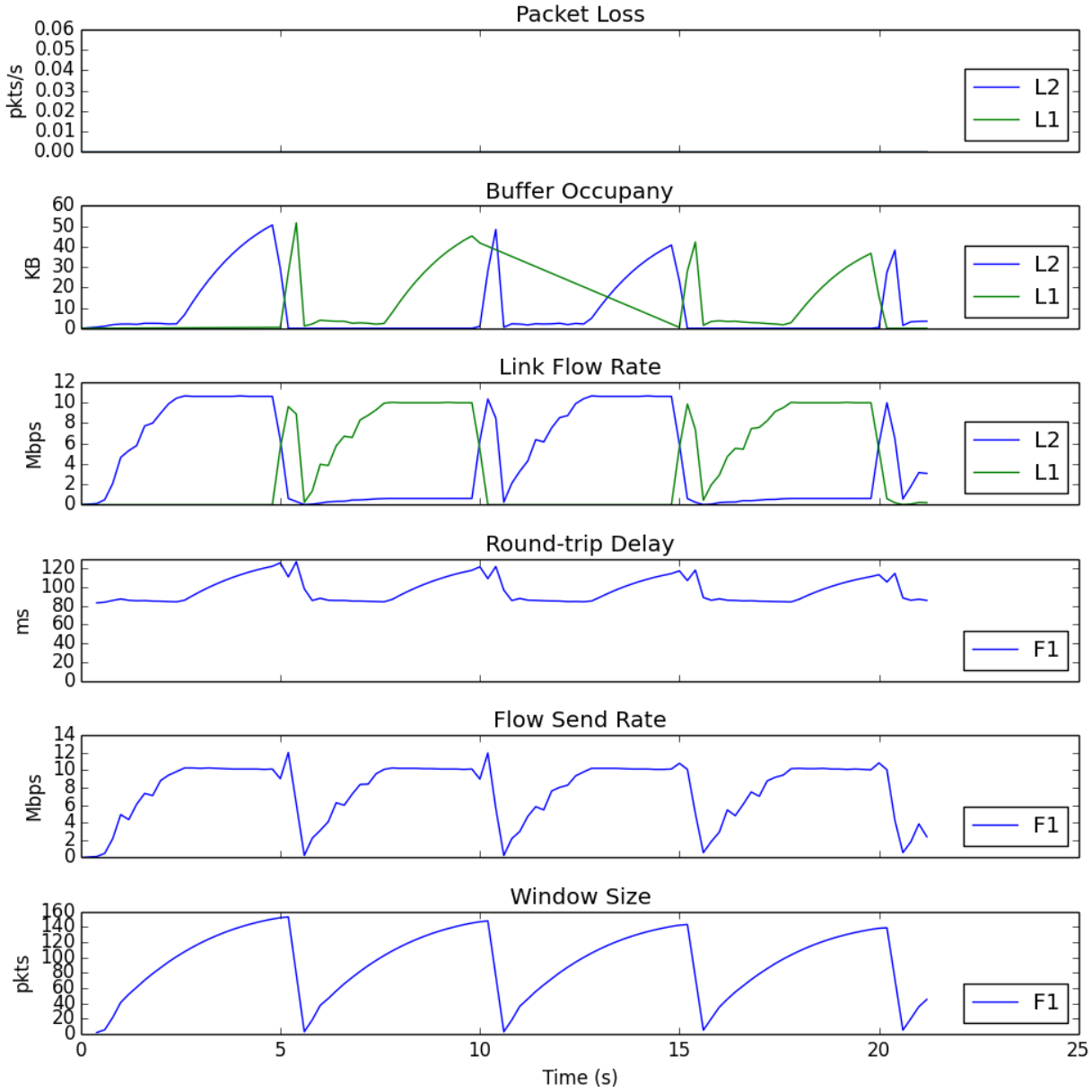
For test case 1, our implementation of FAST-TCP appears to be correct. We see that window size stays approximately constant throughout the entire simulation. Round trip delay and flow send rate stay mostly constant as well. We also observe that our results very much resemble the results given in the sample test cases.

FAST-TCP (α = 10)



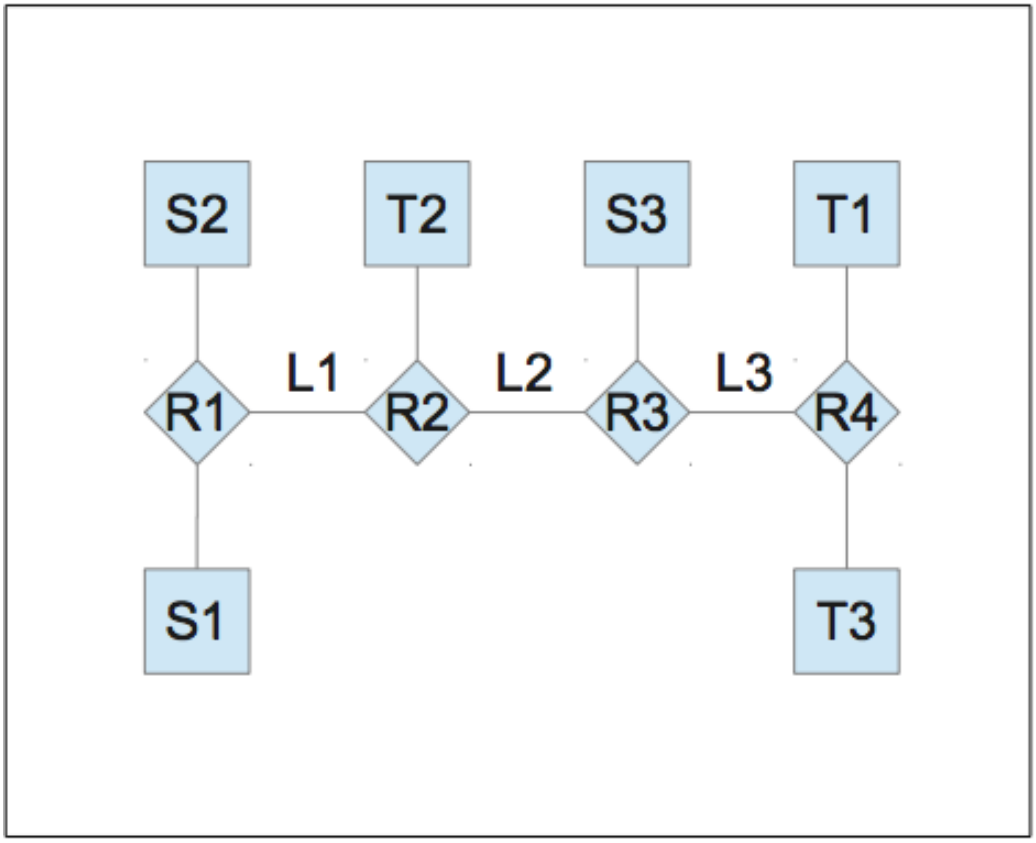
We observe the same patterns in the data as above, only with increased window size and increased buffer occupancy, as expected.

CUBIC TCP



For test case 1, our implementation of CUBIC-TCP appears to be correct. The window size continues to follow the same expected curve for a single flow. For two links, we see the two paths taking turns to send data for the single flow.

Test case 2

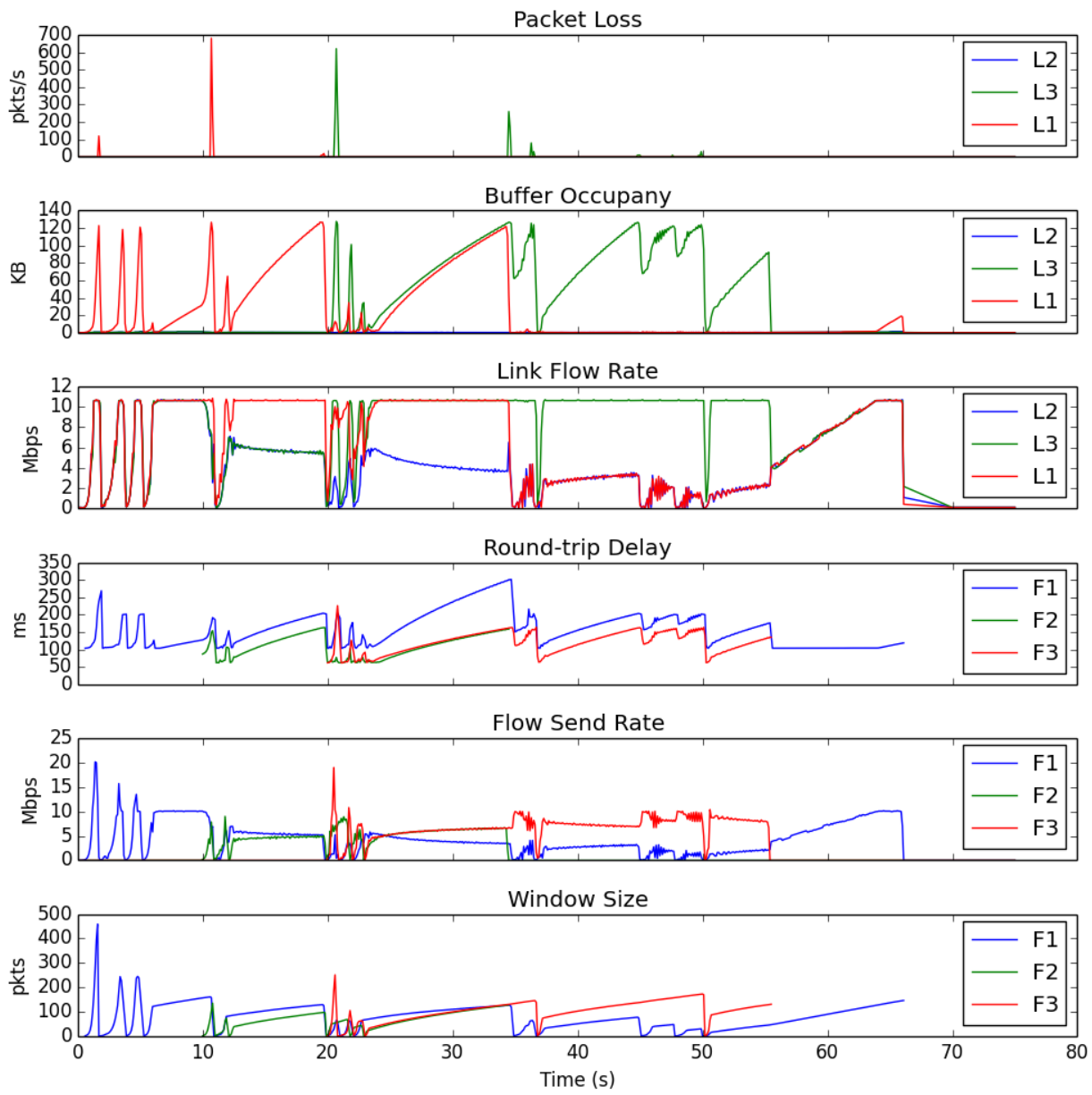


Analysis

In test case 2, we have three flows sending data throughout a more complicated network. Flow 1 sends data from S1 to T1, thus using L1, L2, and L3. Flow 2 sends data from S2 to T2, thus using L1. Flow 3 sends data from S3 to T3, thus using L3.

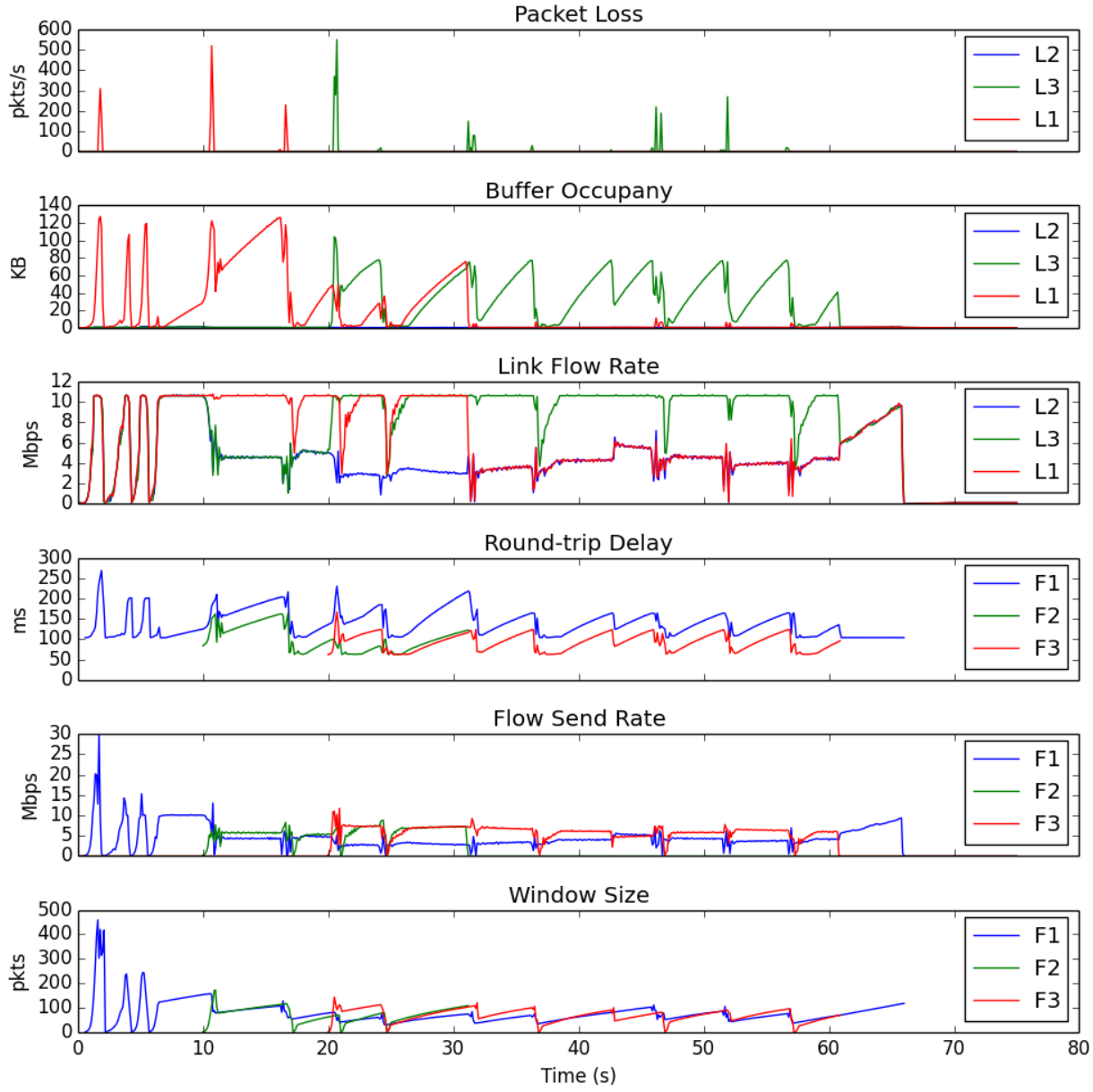
Thus, we expect to see packet loss and buffer occupancy for only L1 and L3.

TCP Tahoe



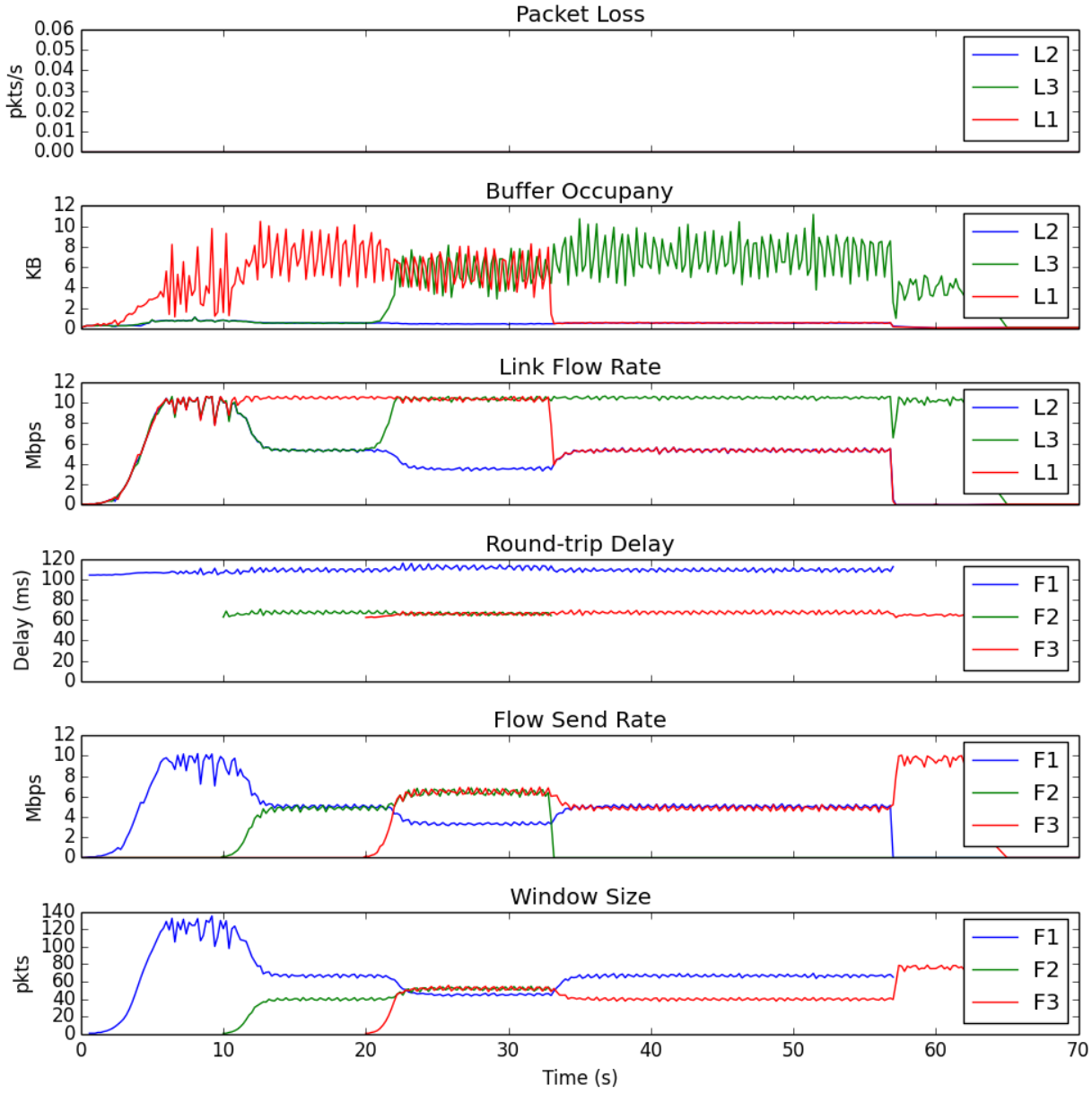
For test case 2, our implementation of TCP-Tahoe appears to be correct. Each time a new flow starts, we see the same fluctuation in link rate that is seen in the sample data. When all three flows are running, we see that at any time, there is fairness for two of the links, while the last link experiences different performance. Packet loss and buffer occupancy are only seen for links 1 and 3 as expected. For round trip delay, flow rate, and window size, all three links behave similarly. Our data is what we expect to see and is similar to the sample data.

TCP Reno



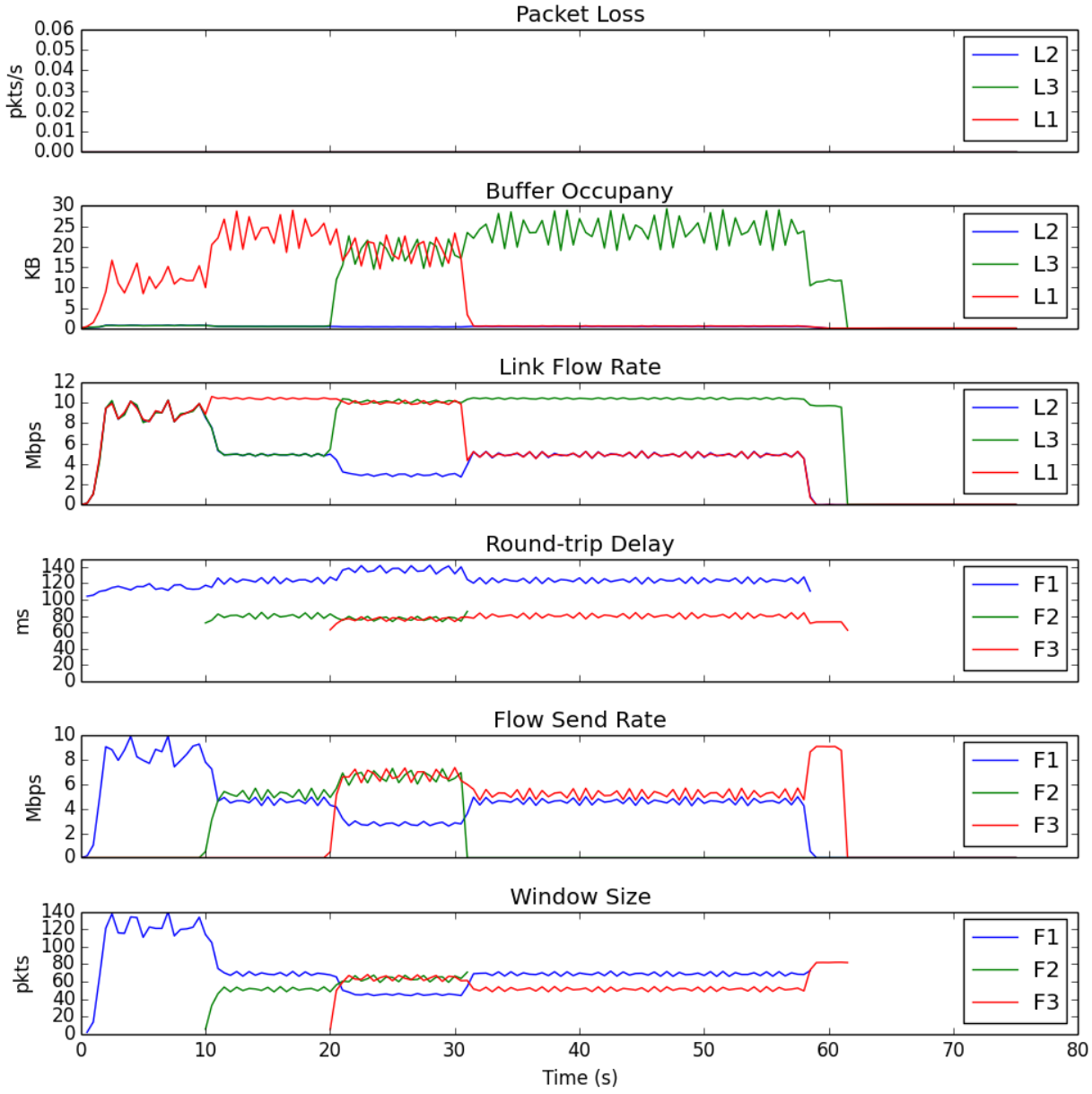
For test case 2, our implementation of TCP-Reno appears to be correct. Each time a new flow starts, we see the same fluctuation in link rate that is seen in the sample data. When all three flows are running, we see that at any time, there is fairness for two of the links, while the last link experiences different performance. Packet loss and buffer occupancy are only seen for links 1 and 3 because only those links experience congestion. For round trip delay, flow rate, and window size, all three links behave similarly. This result is very similar to TCP-Tahoe. However, we see differences in the window size, which we expect because of the difference in the window update step. There is also less round trip delay due to the Fast Recovery step.

FAST-TCP (α = 3)



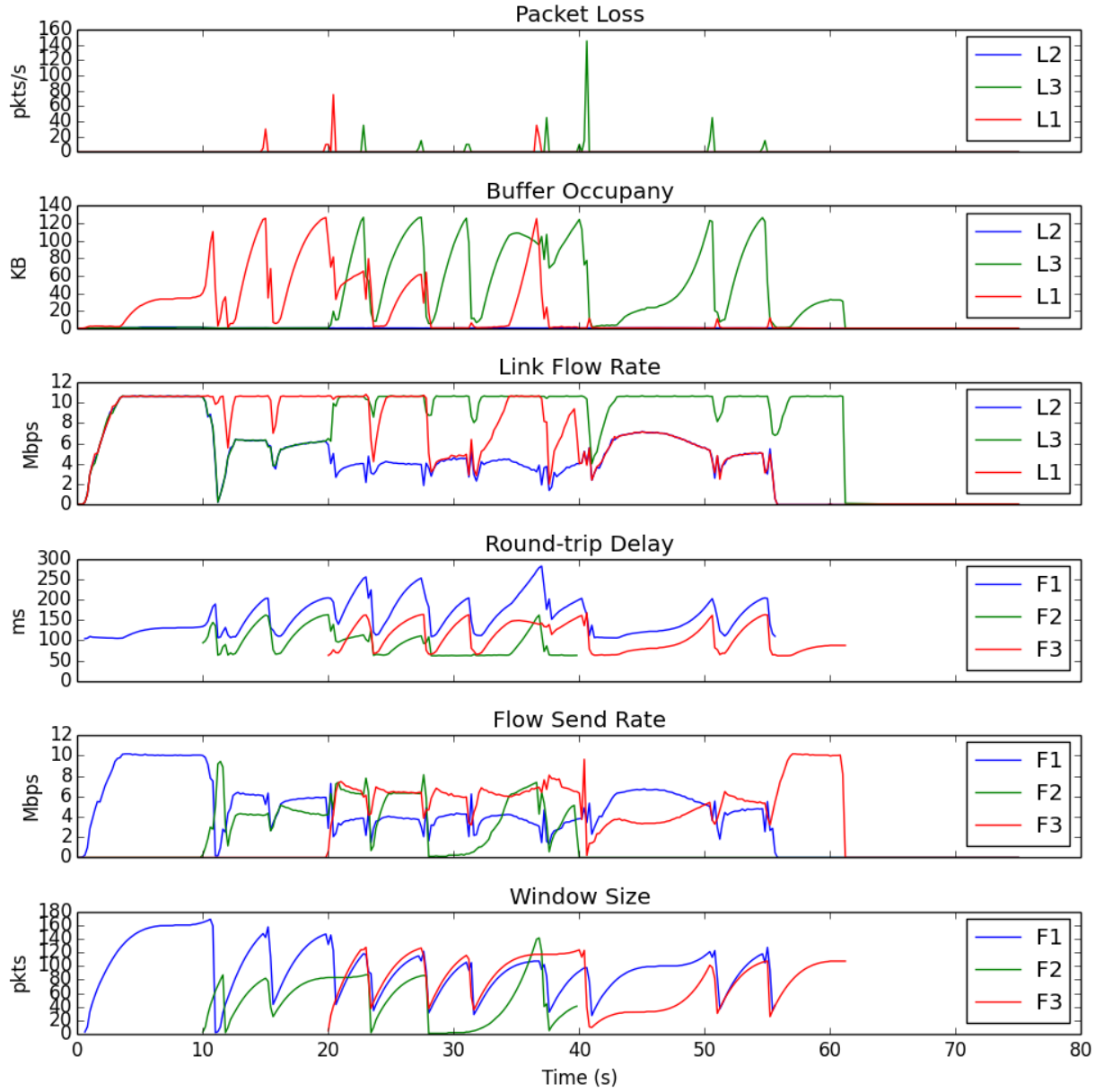
For test case 2, our implementation of FAST-TCP appears to be correct. Because the window size is adjusted according to queueing delay instead of packet loss, we see no packet loss. We see similar patterns for link rate, however they are much more constant because of the window update method.

FAST-TCP (α = 10)



When α is increased, the window size is larger. We see that this causes the data to take less time to complete sending. Additionally, we see a convergence of the link flow rate that remains steady.

CUBIC TCP



For test case 2, our implementation of CUBIC-TCP appears to be correct. In the graph for window size, we see that the plots follow the cubic curve that we expect to see. We had a few issues(i.e. multiple packet loss), but we managed to get good curves.

Distribution of Work

Ying-Yu

- Project Manager
- Architecture
    - Network devices
    - Packets
    - Congestion control
- Link
- Routing
    - Dijkstra's algorithm (SonarPacket and EchoPacket)
- Basic TCP Protocol
- Congestion control algorithms
    - TCP Tahoe
    - TCP Reno

Sharjeel

- Routing
    - Bellman-Ford algorithm (RoutingPacket)
- Helped with graphics and devices

Emily

- Text input design and parsing
- Data processing tool
- Congestion control algorithms
    - FAST-TCP
    - CUBIC TCP

Chris

- Text output design and parsing
- Graphic plotting tool (graphics class)
- Network builder gui tool (guigenerator.py)
- Helped with routing

Project Process

During weeks 1 and 2, we developed an understanding of the project and began to plan out our architecture. We decided what tools we were going to use, i.e. SimPy in Python, Github, Hackpad, etc.

During weeks 3 and 4, we finalized our initial architecture and began implementing some of the basic components of the network. At the same time, group members experimented with SimPy on their own to see how the framework could fit the network components.

By the end of weeks 5 and 6, we had formatted the test cases as input to the network, implemented devices (Host, basic Router, Packet, and Link), and started to work on Flow. While beginning implementation, we realized that our architecture would need adjusting.

By week 7, we had the network working for test case 0. This meant that all of our devices were functioning, and we had a very basic implementation of Flow. We also began working on data output for each simulation.

During weeks 8 and 9, we struggled a lot trying to get the Flow class to work. We ended up having to adjust the architecture once more to accommodate a different, but much improved, implementation of Flow. We also worked on dynamic routing.

During week 10, we perfected our implementation of Flow and our routing algorithms.

The most difficult part was getting the architecture done and other team members understanding it. Implementing and using simpy was also difficult. The hardest part was programming the flow. There is no step-by-step process for programming the flow (like for routing). Hence, it was difficult to program out and visualize the process.

Conclusion

By the end of the ten weeks, we were able to implement a working network simulator. The simulator supports both static and dynamic routing, and congestion control for multiple flows of data through a network of hosts, links, and routers. The data obtained from our simulations for TCP-Tahoe, TCP-Reno, FAST-TCP, and CUBIC-TCP are what we expect to see, both analytically and according to the sample data we were given for test cases 0, 1, and 2.