# Configurable Network Simulation for TCP Reno and FAST Flows

Laksh Bhasin[1], David Luo[1], Yubo Su[1], Sharon Yang[1]
Instructor: Steven Low. December 2015.
[1] *California Institute of Technology, 1200 E California Blvd., Pasadena, CA 91125, USA*

# 1 Overview

The goal of this project was to produce a program that could simulate arbitrary network configurations of flows, routers, and hosts. To this end, we have produced an event-driven simulation in `Python 2.7.10` that is able to simulate flows running TCP Reno and TCP FAST as their congestion control algorithms. We have also applied an object-oriented design for our network components (e.g. `Router`s, `Host`s, `Link`s, etc).

In this section, we present an overview of this system, including our statistics collection. In Section 2, we discuss the network components in detail. Section 3 covers three test cases (Test Cases 0-2). And finally, an analytic check on Test Case 2's FAST performance is included to quantitatively validate our results.

## 1.1 Events

In the context of this simulation, `Event`s are instantaneous occurrences that happen at a specific moment in time (relative to some global clock). Since these occurrences cover a wide range of functionality, from initiating a routing table update to handling a packet receipt, we have defined a general interface for `Event`s with two functions:

- `run()` — Carries out the main actions of an `Event` and updates statistics.
- `schedule_new_events()` — Adds follow-up `Event`s and updates statistics if necessary.

`Event`s themselves are run as part of an event loop. The event loop represents a priority queue of `Event`s, where priority is based on the execution time of the `Event`. The execution of this loop simply involves getting the earliest `Event`, calling `run()` and `schedule_new_events()` in turn, and then repeating this until all `FlowCompleteEvent`s are received. In order to facilitate event scheduling, our `MainEventLoop` class has a `schedule_event_with_delay()` function.

## 1.2 Statistics

Our simulator includes a `Statistics` class, which holds fields that map network components (specifically, `Link`s, `Host`s, and `Flow`s) to various collected statistics about them. Specifically, the following statistics are recorded:

- For `Link`s:
  - A list of (timestamp, buffer occupancy) tuples that marks changes in buffer occupancy over time. Note that only `DataPacket`s are included in the buffer occupancy statistic, in order to make the Test Case 2 analytic comparison easier (see Subsection 3.4).
  - A list of timestamps corresponding to packet losses (due to buffer overflow).
  - A list of (timestamp, packet size) tuples that describe each packet transmitted.
- For `Host`s:
  - A list of (timestamp, packet size) tuples that describe each packet sent.
  - Same as the previous field, but for each packet received at the host.
- For `Flow`s:
  - A list of (timestamp, packet size) tuples that describes each packet sent.
  - Same as the previous field, but for each (ACK) packet received for that flow.
  - A list of (timestamp, round-trip time (RTT)) tuples for various packets received.
  - A list of (timestamp, window size) tuples describing changes in window size at specific times.

`Link` statistics are used to plot time traces of the buffer occupancy, packet losses (using time interval windows), and transmission rate (also using time interval windows). `Host` statistics are used to plot both the send and receive rates of a host as a function of time (using windowing). `Flow` statistics are used to plot time traces of flow send and receive rates (with windowing), data packet RTTs, and flow window sizes. All of this statistical processing is described in our `plot_tool` module, which outputs both time traces (using `matplotlib`) as well as simulation-long averages of the aforementioned statistics.

## 1.3 Network Topology JSON

`NetworkTopology` is a wrapper class that holds a list of `Link`s, `Host`s, `Router`s, and `Flow`s. This class is used to conveniently convert network topologies into JSON, using the `jsonpickle` package. It is important to make sure that a `NetworkTopology` object's data fields do not hold references to other components since `jsonpickle` doesn't properly handle saving references (e.g. `Router`s do not store their connected links as a reference in the original `NetworkTopology` JSON). All of the necessary component reference connection happens in the `NetworkTopology` class's `complete_initialization()` function.

### 1.4 Overall Program Flow

The overall flow of this simulator is described in Figure 1. The initializer script sets up initial Events, including Flow initiation and routing table updates, which are then passed into the main event loop.

This network simulator uses the initializer script as its entry point. One example usage of this script is as follows:

- Example usage (from root directory of project): `python2 initializer.py -v INFO -f log/log.txt data/test_case_0.json -l L0 L1`

In this case, the above script call will take data/test_case_0.json as the input NetworkTopology. The log level will be set to "INFO", with the file log/log.txt used as the log file (although stdout may also be used). Lastly, the final optional arguments ("-l L0 L1") specify desired Links' statistics to be plotted.

## 2 Network Components

Network components are placed in separate Python modules. Each network component's module (e.g. link.py for Links) contains a class definition for that component, as well as subclasses (e.g. FlowFast is a TCP FAST implementation of Flows) and Events related to that component. The functions defined for each network component are supposed to run "instantaneously" in simulation time.

The following network components are included in this simulation. Each of these is described in more detail in the following subsections.

- Device (an abstract class that simply features an address field).
  - Subclasses: Host and Router
- Link, LinkBuffer, LinkBufferElement
- Packet (abstract class)
  - Subclasses: DataPacket, AckPacket, and RouterPacket
- Flow (abstract class)
  - Subclasses: FlowReno and FlowFast

### 2.1 Packet

Packets are units of information in the network. They are generated by a Flow; all Packets in the same Flow go back and forth between the same two Hosts. There are three kinds of Packets: DataPackets, AckPackets, and RouterPackets.

#### 2.1.1 Fields

Packet fields include:

- packet_id — The id of the Packet.
- flow_id — The flow that the Packet is a part of.
- source_id, dest_id — The source and destination of the Packet.
- start_time_sec — The time when the Packet was sent (from the Host it originated at).
- size_bits — The size in bits of the Packet.

#### 2.1.2 Subclasses

Packets have three subclasses:

- DataPacket — An 8192-bit Packet used to transfer data between Hosts. This subclass has no additional fields of interest.
- AckPacket — A 512-bit Packet used to send ACKs between hosts. This subclass has three additional fields of interest:
  - data_packet_start_time_sec — The start time of the original DataPacket that triggered this ACK.
  - flow_packets_received — An ascending sorted list of all of the Packet IDs received by the destination Host for the given Flow. This is used to implement selective repeat on the Flow's side.
  - loss_occurred — A boolean that is (later) set to True if flow_packets_received has a gap in its list of Packet IDs.
- RouterPacket — A 512-bit Packet used to send routing tables between neighboring Routers. This has one field of interest:
  - router_to_host_dists – A map from (Router, Host) pairs to the minimum distance (time-wise) between that Router and Host. This map represents a Router's "current" knowledge on the min cost to all known Hosts (accumulated since its last update began).

### 2.2 Host

Hosts are endpoints of the network. They have network addresses that are distinct from other Device addresses. Each Host keeps track of the Flows that originate from it and the Packets that those Flows receive. A HostReceivedPacketEvent is triggered when a Packet is received. If the Packet is a DataPacket, the list of Packets received by that Host for the given Flow is updated. If the Packet is an AckPacket, a FlowReceivedAckEvent is generated, and the Flow itself handles the receipt. RouterPackets are never received by Hosts.

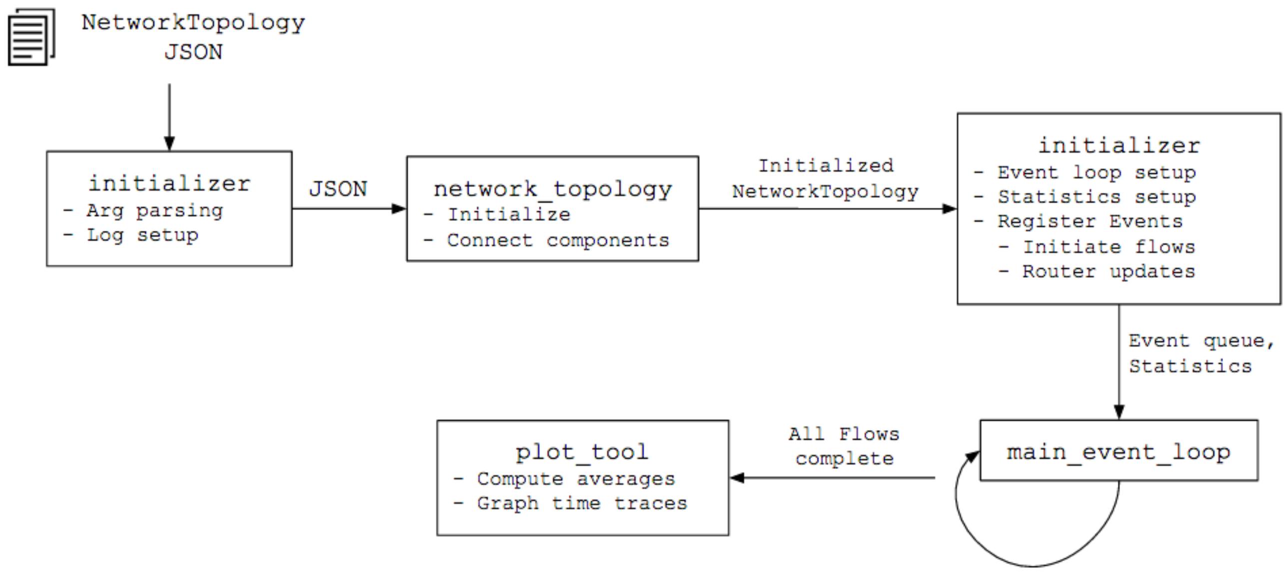


**Figure 1:** *A flow chart describing the flow of control in this network simulator. Titles for each box represent file names, while bullet points describe functionality. The entry point for this simulator is the* `initializer` *script, which takes a* `NetworkTopology` *JSON file (and optional arguments). After the topology is set up and connected together, initial* `Events` *are set up and passed into the* `MainEventLoop`*. Once all* `Flows` *finish running, the loop exits and statistics are outputted.*

#### 2.2.1 Fields

Some of the fields of a `Host` are:

- `flows` — The `Flows` originating from this `Host`.
- `flow_packets_received` — A map from the `flow_id` to a sorted list of `packet_ids` that have been received for that `Flow`.
- `link` — The `Link` connected to the `host`.

#### 2.2.2 Events

Host has the following `Events`:

- `HostReceivedPacketEvent` — Represents a `Host` receiving a `Packet`. The behavior of this `Event` depends on the kind of `Packet` received:
  - If the `Packet` received is a `DataPacket`, then the `run()` function puts the `Packet`'s ID into flow_packets_received via binary search (to maintain sorted order) and updates statistics. The `schedule_new_events()` function then generates an `AckPacket` to send back to the `DataPacket`'s original `Host`.
  - If the `Packet` received is an `AckPacket`, then the `run()` function just updates statistics, and the `schedule_new_events()` function simply generates a `FlowReceivedAckEvent`.

### 2.3 Link

`Links` connect `Devices` together and carry `Packets` along this connection. In this simulation, every `Link` is half-duplex (only one direction may be traveled at a time) and has a specified capacity. As such, there is only one buffer per `Link`, and `Packets` in this buffer may be going towards either end of the `Link`.

#### 2.3.1 Fields

`Link` fields include but are not limited to:

- `end_1_device, end_2_device` — These record the endpoints of the `Link`. A `get_other_end(Device)` function is provided so one `Device` can ask a connected `Link` what `Device` is on the other end.
- `static_delay_sec` — The propagation delay (in seconds). This represents one component of the queuing-independent link delay (the other being the transmission delay).
- `capacity_bps` — The `Link`'s capacity in bits per second. This allows us to compute the transmission delay for a given `Packet` size.
- `LinkBuffer` — A class containing the following fields:
  - `max_buffer_size_bits` — The maximum buffer size in bits.
  - `queue` – A FIFO queue of `LinkBufferElements`, which represents a `Link`'s buffer. These elements, in turn, contain the following fields:

    * packet — The Packet being sent.
    * dest_dev — The end of the Link that the Packet is going towards. Must be one of {end_1_device, end_2_device}.
    * entry_time — The time at which the Packet entered the buffer.
  - queuing_delays — A deque of up to 30 queuing delays of packets that exited less than 1 second ago, which is used to compute an average queuing delay.

#### 2.3.2 Functions

Some of the functions in the Link class are:

- get_link_cost() — Computes the Link's cost (in seconds) as a sum of propagation delay (static_time_delay) and average queuing delay (computed using the aforementioned deque of queuing delays). The transmission delay is ignored.

- push() — Pushes a Packet onto the LinkBuffer. This Packet will later be sent to one end of the Link via LinkSendEvents. RouterPackets are always pushed onto the buffer even if it forces the LinkBuffer's maximum size to be exceeded. In other words, for the purposes of this simulation, the possibility of dropped RouterPackets was not considered, as this would greatly disrupt Bellman-Ford. On the other hand, DataPackets and AckPackets are dropped if the LinkBuffer's maximum size is exceeded.

The LinkBuffer class also has a pop() function that pops off the top LinkBufferElement and updates metadata (including the queuing_delays deque).

#### 2.3.3 Events

Links have the following Events governing their logic:

- DeviceToLinkEvent — Represents a Packet arriving at this Link. When this Event runs, it pushes the Packet and, if the Link is not busy, it immediately schedules a LinkSendEvent. Otherwise, it lets the preexisting LinkCheckStatusEvent (which is already in the event loop's priority queue) automatically schedule the next LinkSendEvent.

- LinkSendEvent — Represents a Packet leaving the LinkBuffer and heading to the next Device on its route. This Event schedules a HostReceivedPacketEvent or a RouterReceivedPacketEvent for a time get_link_cost() seconds in the future. It also schedules a CheckLinkStatusEvent a single transmission delay in the future (which represents the time needed before the next Packet can be transmitted). Finally, this Event sets the Link's status to busy.

- CheckLinkStatusEvent — Checks whether the Link's buffer is empty and, if so, sets the Link's status as non-busy and does nothing. Otherwise, a LinkSendEvent is immediately scheduled. Note that this LinkSendEvent is delayed from the previous LinkSendEvent by a transmission time, as expected.

### 2.4 Router

Routers route packets through the network and sit between Devices. They have distinct network addresses (from other Devices), can have an arbitrary number of links connected, and are assumed to route Packets instantaneously.

In our simulation, Routers periodically initiate routing table updates, and use the Bellman-Ford algorithm to compute the next-hop Link to get to a given Host. These updates begin with the Router updating its costs to direct neighbors, and broadcasting those costs to all of its neighboring Routers via RouterPackets. As each neighboring Router receives RouterPackets, they account for their neighbors' data and recompute their own routing tables via Bellman-Ford. If their own cost estimates (i.e. time to go from that Router to any Host) changed, those Routers simply rebroadcast their routing tables, with the new information included. For the topologies tested, this procedure eventually leads to convergence.

#### 2.4.1 Fields

Some noteworthy fields for Routers are:

- links — List of Links connected to Router
- neighbors — List of directly-connected Devices.
- stable_routing_table — A stable version of the routing table, which maps (destination) Host addresses to a next-hop Link.
- new_routing_table — An (initially) unstable version of the routing table, which is used to build up a new routing table after an update is initiated. This table can only be used if 0.15s have passed since the routing table update initiation, or if the stable routing table doesn't contain a given Host.
- last_table_update_timestamp — The global time (in seconds) when the last routing table update began.
- self_to_neighb_dists — The known (time) costs to go from this Router to a neighbor. This includes queuing delays (from when they were last measured, at the beginning of an update).
- neighb_to_host_dists — The minimum cost (in seconds) to go from a neighboring Router to a given Host. This is what gets communicated via RouterPackets.
- self_to_host_dists — The minimum cost (in seconds) to go from this Router to a given Host, as estimated via Bellman-Ford and message-passing.

#### 2.4.2 Functions

A few significant functions in the Router class are:

- get_link_to_host_id() — Finds the next-hop Link to get to a given Host address, based on the routing table. This uses the new routing table if 0.15s passed since the last update, or if the Host of interest is not in the stable table.

- update_routing_table() — Uses received information on a neighboring Router's estimated costs to various Hosts, in order to update this Router's self → Host costs (via Bellman-Ford) and routing table. This function is called when a RouterPacket is received, or when a routing table update is initiated; in the latter case, the Router's self → neighbor costs are updated to account for the latest queuing delays.

- broadcast_router_packets() — Sends RouterPackets to all neighboring Routers.

#### 2.4.3 Events

Some Router-related Events (which are defined in router.py) are:

- InitiateRoutingTableUpdateEvent — Initiates a routing table update, and then schedules another similar Event 5s later. These are one of the first Events to be scheduled in the initializer script.

- RouterReceivedPacketEvent — Represents a Router receiving a Packet. The behavior of this Event depends on the kind of Packet received:
  - If the Packet received is a DataPacket or an AckPacket, then this Event looks up the next-hop Link (from the routing table) and routes the Packet onto that Link (via a DeviceToLinkEvent).
  - If the Packet received is a RouterPacket, then this Event updates this Router's routing table based on the information in the RouterPacket. Additional RouterPackets are rebroadcasted if the (receiving) Router's self → Host minimum (time) cost estimates changed.

### 2.5 Flow

Flows represent an active connection between a source and destination Host in the network. Their main purpose generate Packets at a rate controlled by the TCP congestion control algorithm for that Flow. They send Packets until they've filled a certain window size (e.g. 8 packets), and adjust this window size in response to losses, successes, or other metrics. If the window size has already been exceeded (which can happen if the window size suddenly dropped in response to a loss), then Flows cannot transmit any *new* Packets.

Flows are also responsible for retransmitting lost Packets based on (repeated) ACK gaps (i.e. Packet ID gaps in AckPacket data) or timeout. Retransmissions do not increase the number of Packets in transit, and are always immediately allowed even if the window size has (temporarily) been exceeded.

In this simulation, Flow is just an abstract class; there are specific subclasses for TCP Reno (FlowReno) and TCP FAST (FlowFast) implementations.

#### 2.5.1 Common Fields and Functions

The following are some of the fields common to all Flows:

- flow_id — The unique ID representing this Flow.

- source, dest — The source and destination Hosts for this Flow.

- data_size_bits — The size (in bits) of the data being sent.

- start_time_sec — The start time of the flow (in seconds), relative to the main event loop's global clock.

- window_size_packets — The current window size (in Packets).

- packets_in_transit — A set of Packet IDs that are either currently in transit, or scheduled to (soon) be in transit.

- gap_retrans_packets — A set of DataPacket IDs that have been retransmitted due to (possibly multiple) gaps in an AckPacket's selective repeat data. This is used to make sure retransmissions due to selective repeat gaps are not repeated (but retransmissions due to timeouts can be repeated).

- num_timeouts_pending — A map from Packet ID to the number of timeouts pending for that Packet (can be $> 1$ due to retransmission). This is used to make sure only the latest timeout scheduled for a given DataPacket is handled.

And these are the abstract functions common to all Flows:

- handle_packet_success() — Responds to a successful round trip (i.e. the receipt of an AckPacket). This function also updates metadata (e.g. RTTs), statistics, and window sizes as necessary.

- handle_packet_loss() — Responds to a timeout or a single AckPacket indicating a gap. This function updates metadata, statistics, and window sizes as necessary. DataPackets are also retransmitted if needed. Note that losses are handled *after* successes when an AckPacket is received, so the loss handler has the final say on window sizes, etc.

### 2.5.2 FlowReno

This class is an implementation of TCP Reno, based largely on [1]. There are three states that a `FlowReno` can be in at a given time: slow start (SS; usually only in the beginning or after timeouts), congestion avoidance, and fast retransmit / fast recovery (FR/FR). The criteria and behavior for entering or leaving FR/FR are described in [2]; the only difference is that this implementation looks for three `AckPacket` gaps for the same `Packet` ID (selective repeat), not three "duplicate ACKs."

The ACK success handler simply updates the window size and state based on the current state (see [1] for more details). The loss handler handles two types of losses separately:

- Three separate `AckPackets` indicating a gap (i.e. a break in the list of `Packet` IDs received by a destination `Host`) for the same `Packet` ID. In this case, the `DataPacket` is retransmitted and the `Flow` enters FR/FR.
- Timeouts (0.5s after transmission). In this case, the `DataPacket` is retransmitted, and the `Flow` enters SS and updates its SS threshold.

### 2.5.3 FlowFast

This class is an implementation of TCP FAST, based on both [1] and [3]. Unlike Reno, FAST only makes updates to window size periodically – with a 0.05s period in this simulation. These updates compare the recent RTTs to the smallest-ever RTT (`base_rtt`), and lower the window size proportionally when the current RTT is too large. Because the FAST formula [3] makes these proportionate window size changes, it is able to achieve stable window sizes more easily than Reno, without overshooting link capacities. Reno, on the other hand, exhibits sawtooth patterns in window size time traces, constantly going back and forth between CA and FR/FR or SS as the network capacity gets exceeded.

In computing the "recent" RTT for FAST, we average the RTTs of the last 40 `AckPackets` that were received by the `Flow`. In addition, for our window size formula, we use parameters of $\gamma = 0.5$, and either $\alpha = 50$ (for Test Cases 0 and 2) or $\alpha = 30$ (for Test Case 1). The window size updates happen in the `handle_periodic_interrupt()` function.

The success and loss handlers for FAST are relatively simple. On success, the `base_rtt` field is updated, in addition to basic metadata and stats. After a loss, which can either be a timeout or a single ACK gap (not 3 like Reno), the `DataPacket` is retransmitted. The only caveat is that `Packets` are not retransmitted after a single ACK gap if they were already retransmitted for that reason (to avoid multiple retransmissions).

### 2.5.4 Events

The following `Flow Events` are included in this simulation:

- `InitiateFlowEvent` — Kicks off a `Flow` by sending the first full window of `Packets`. These `Events` are scheduled in the `initializer` script.
- `FlowReceivedAckEvent` — Called when a `Flow` receives an `AckPacket`. This `Event` calls the success/loss handlers appropriately (e.g. a loss occurs when the `AckPacket` indicates a gap), and then sends `Packets` to make sure the new window size is filled.
- `PeriodicFlowInterrupt` — Periodically (with a period of 0.05s) calls `handle_periodic_interrupt()` on a `FlowFast`, fills up a window of `Packets`, and then schedules another periodic interrupt.
- `FlowSendPacketsEvent` — Sends multiple `Packets` to the `Host`'s connected `Link`, and schedules timeouts for them (in 0.5s).
- `FlowTimeoutPacketEvent` — Responds to timeouts by calling the loss handler, unless the `Packet` was already received or there is a later timeout pending (due to the `Packet` being retransmitted).
- `FlowCompleteEvent` — Called when a `Flow` finishes sending data. This event does nothing, but notifies the main event loop about the completion.

# 3 Test Cases

The following three test cases demonstrate the performance of TCP FAST and TCP Reno under various topologies and `Flow` specifications. Note that only time traces are included, although the simulator is capable of outputting averages as well. In addition, only `Link` and `Flow`-specific data is plotted; `Host`-specific plots have been excluded for space reasons, as have plots of `Flow` receive rates (which are usually time-delayed versions of the send rate plots, decreased by a factor of 16 due to `AckPacket` and `DataPacket` size differences).

## 3.1 Test Case 0

Test case 0 is a very simple setup. It comprises a single `Flow` going over a single `Link` between two `Hosts`. The setup is provided in Figure 2, and the specs are in Figure 3.

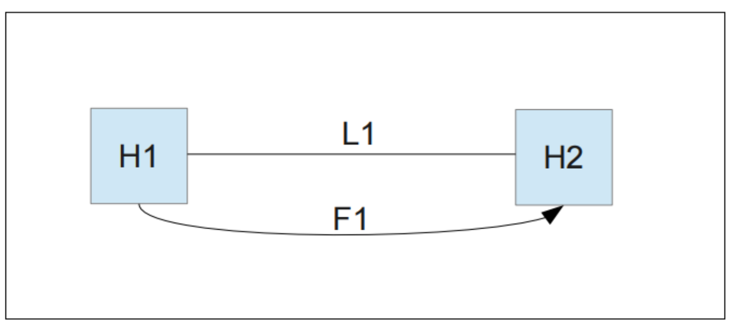


**Figure 2:** *Setup for Test Case 0.*

***Link Specifications***

| **Link ID** | **Link Rate (Mbps)** | **Link Delay (ms)** | **Link Buffer (KB)** |
|---|---|---|---|
| L1 | 10 | 10 | 64 |

***Flow Specifications***

| **Flow ID** | **Flow Src** | **Flow Dest** | **Data Amt (MB)** | **Flow Start (s)** |
|---|---|---|---|---|
| F1 | H1 | H2 | 20 | 1.0 |

**Figure 3:** *Specs for Test Case 0.*

The results of running this test case with FAST and Reno are provided in Figure 4 and Figure 5 respectively. In the FAST output, the time traces rise steeply upward a couple of seconds after the run has initiated. In particular, the flow window size soon attains a peak value and stabilizes. The send and receive rates oscillate between a small range of values (some of which might be due to the chosen window size) but, overall, they remain at a similar rate of almost 10 Mbps. The Reno output, on the other hand, does not exhibit the same stabilization as FAST. Most plots, including `Link` buffer occupancy, `Flow` send and receive rates, and window size, zigzag in a sawtooth fashion. This is expected because, when the `FlowReno` encounters multiple `AckPacket` gaps or timeouts (due to `Packet`s being dropped), it cuts down on its window sizes. This in turn reduces congestion and rates. Finally, it is worth noting that the `FlowReno` case takes a few seconds longer to terminate than the `FlowFast` case, and also includes more packet losses (due to buffer overflow, which occurs fairly regularly after the first losses at $\sim$ 1s).

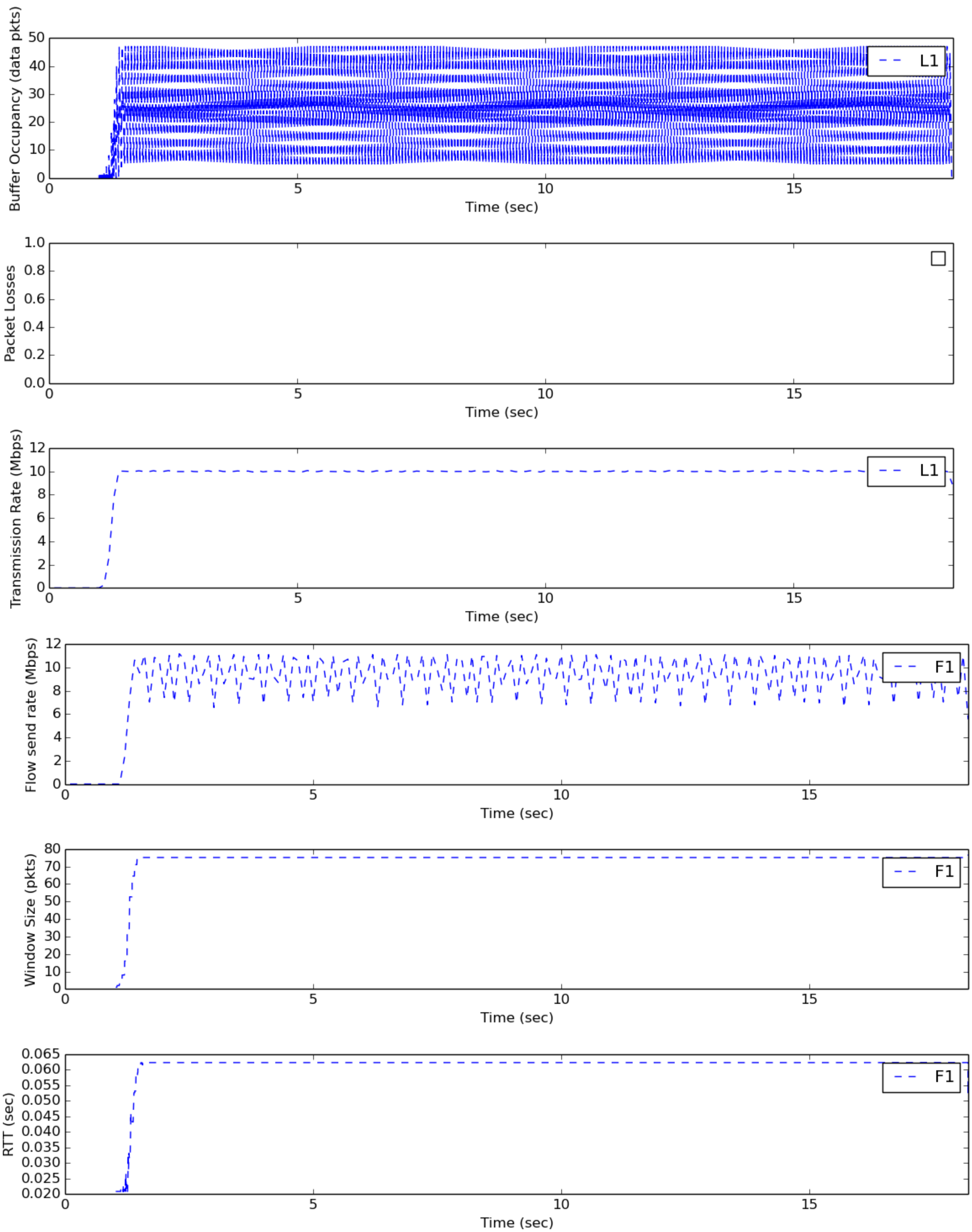


**Figure 4:** *Statistics of Test Case 0 with a TCP FAST* `Flow`*.*

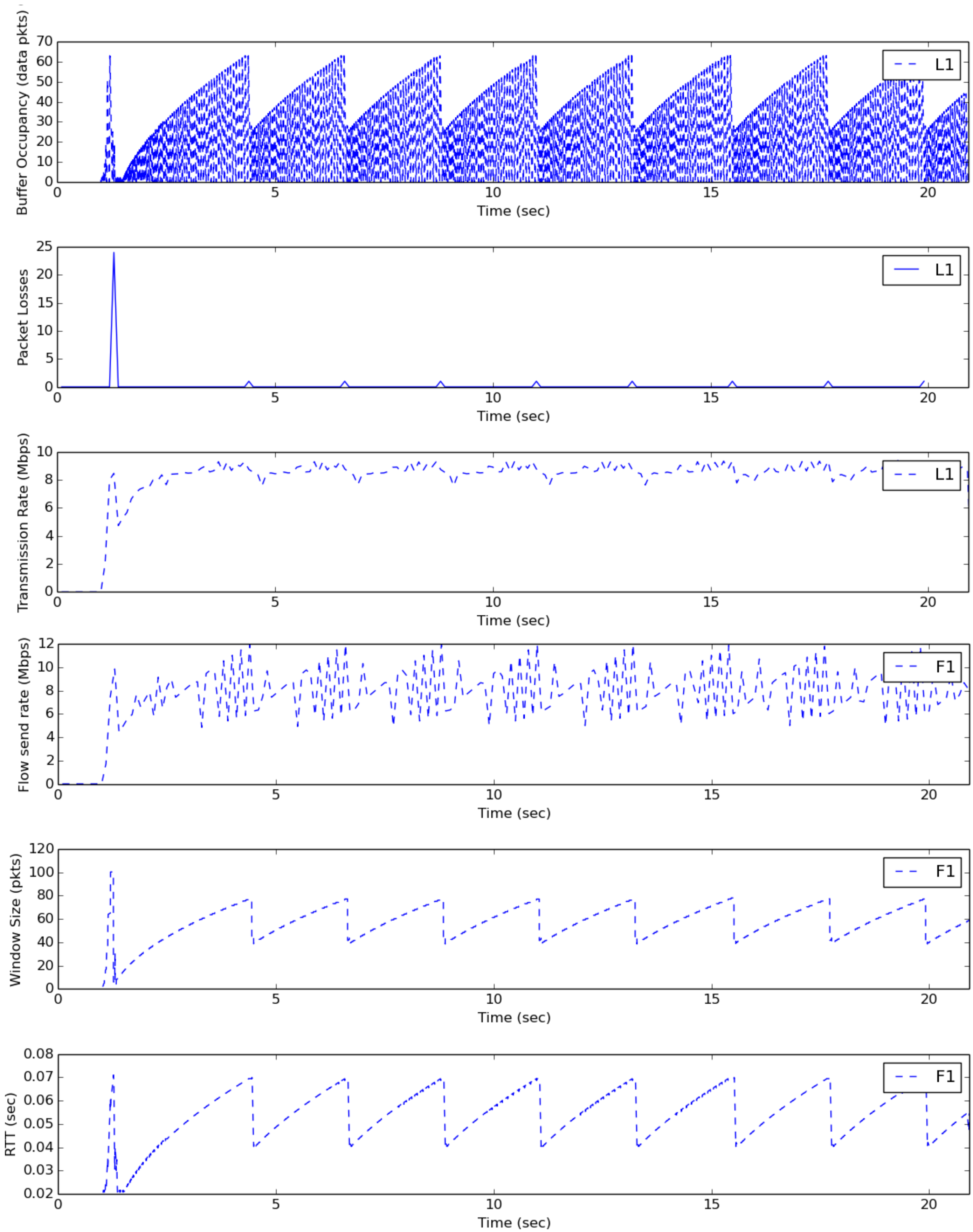


**Figure 5:** *Statistics of Test Case 0 with a TCP Reno `Flow`.*

## 3.2 Test Case 1

Test case 1 looks at whether `Routers` can detect and adjust for queuing delays, in recomputing their paths. This case's setup is depicted in Figure 6, and its specs are in Figure 7.

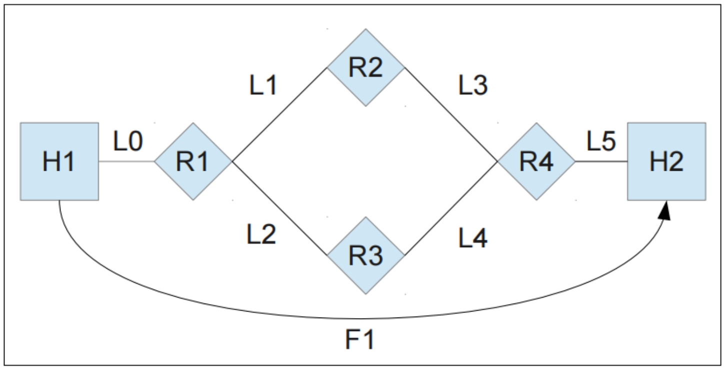


**Figure 6:** *Setup for Test Case 1.*

***Link Specifications***

| Link ID | Link Rate (Mbps) | Link Delay (ms) | Link Buffer (KB) |
|---|---|---|---|
| L0 | 12.5 | 10 | 64 |
| **L1** | 10 | 10 | 64 |
| **L2** | 10 | 10 | 64 |
| L3 | 10 | 10 | 64 |
| L4 | 10 | 10 | 64 |
| L5 | 12.5 | 10 | 64 |

***Flow Specifications***

| Flow ID | Flow Src | Flow Dest | Data Amt (MB) | Flow Start (s) |
|---|---|---|---|---|
| F1 | H1 | H2 | 20 | 0.5 |

**Figure 7:** *Specs for Test Case 1.*

There is one source `Host` (H1) that can choose between two distinct paths to its endpoint `Host` (H2). The expected behavior is that, when a routing table update is performed, the `Routers` account for the queuing delay through the current path, and therefore reroute future `Packets` through the other path. Thus, packets are expected to alternate between the two paths after each routing update (which occurs every 5s).

The results of running the flow using FAST and Reno congestion control algorithms are provided in Figure 8 and Figure 9 respectively. Indeed, as expected, the flow of `Packets` does swap between the two links L1 and L2 every 5s (with a routing table stabilization delay of 0.15s as explained in Subsection 2.4). Furthermore, as in Subsection 3.1, FAST plots show stabilization and Reno plots exhibit a sawtooth pattern. This increased stability at a high send rate allows FAST to terminate a few seconds quicker, as before.

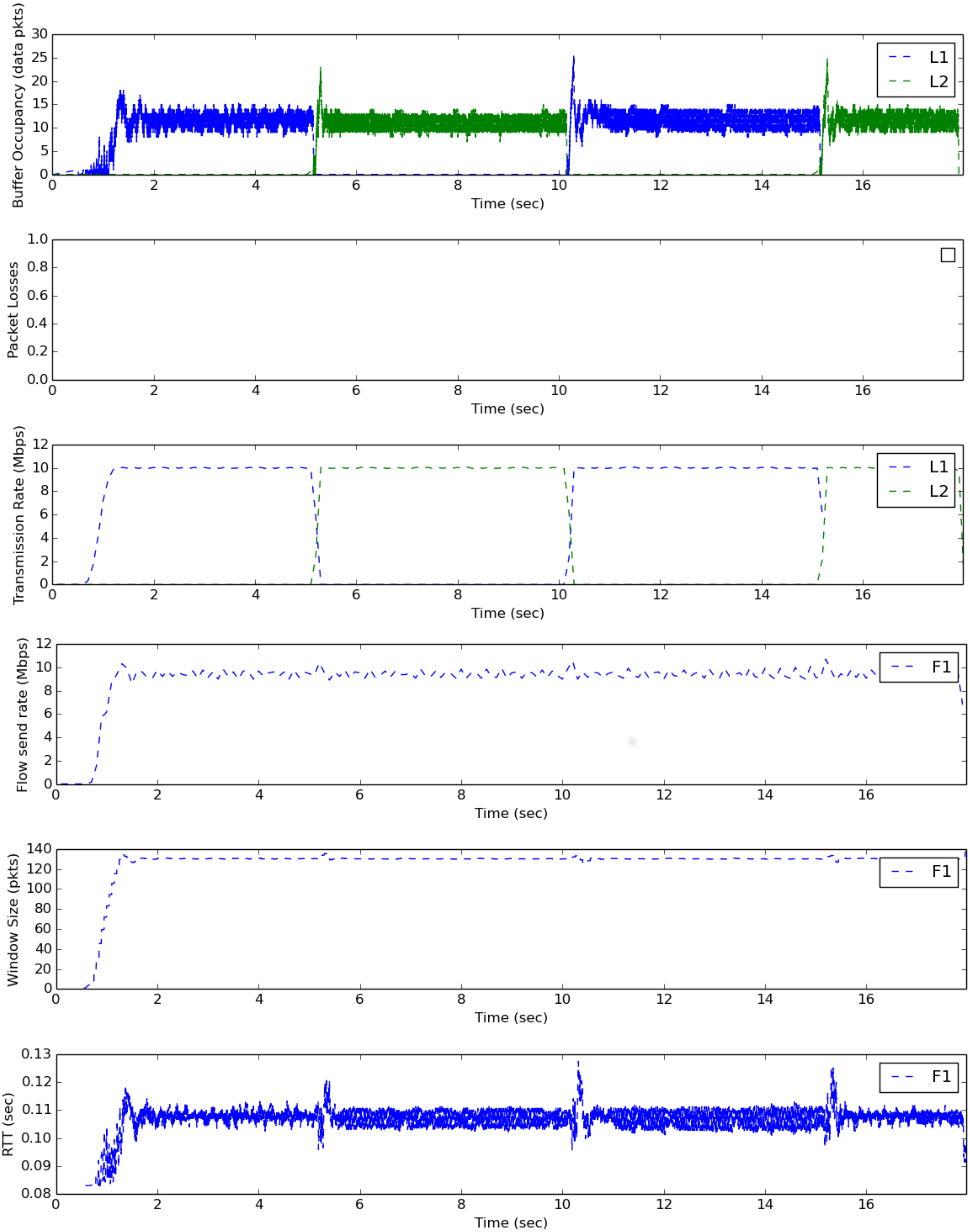


**Figure 8:** *Statistics of Test Case 1 with a TCP FAST* `Flow`*. Note that only L1 and L2's statistics are plotted in the* `Link`*-related graphs.*

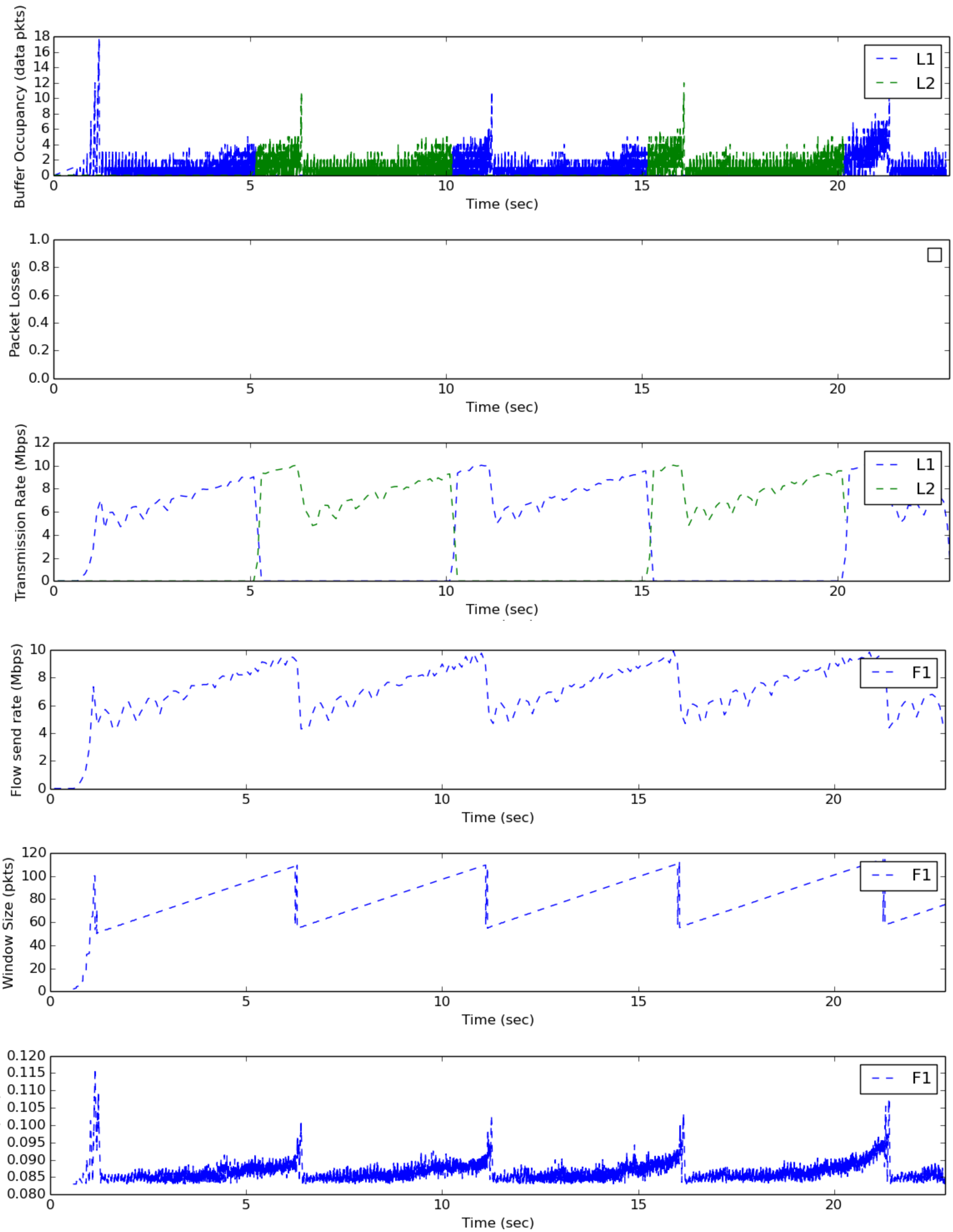


**Figure 9:** *Statistics of Test Case 1 with a TCP Reno* **`Flow`**. *Note that only L1 and L2's statistics are plotted in the* **`Link`***-related graphs.*

### 3.3 Test Case 2

Test case 2 tests interactions between multiple `Flows` within the same network. Three `Flows` start up at different times, between three pairs of distinct sources and destinations, and compete for the same `Links`' resources. The setup is provided in Figure 10, and the specs are in Figure 11.

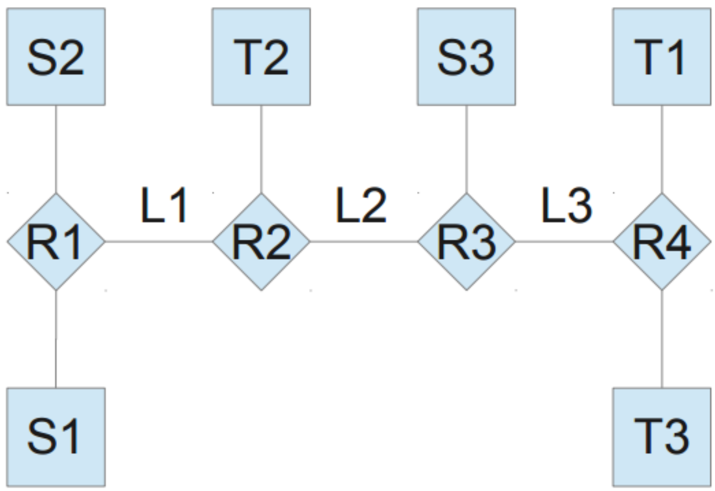


**Figure 10:** *Setup for Test Case 2.*

***Link Specifications***

| **Link ID** | **Link Rate (Mbps)** | **Link Delay (ms)** | **Link Buffer (KB)** |
|---|---|---|---|
| **L1, L2, L3** | 10 | 10 | 128 |
| All Other Links | 12.5 | 10 | 128 |

***Flow Specifications***

| **Flow ID** | **Flow Src** | **Flow Dest** | **Data Amt (MB)** | **Flow Start (s)** |
|---|---|---|---|---|
| F1 | S1 | T1 | 35 | 0.5 |
| F2 | S2 | T2 | 15 | 10 |
| F3 | S3 | T3 | 30 | 20 |

**Figure 11:** *Specs for Test Case 2.*

The results of running the flow with FAST and Reno are provided in Figure 12 and Figure 13 respectively. Overall, the plots show the same general characteristics of FAST and Reno as described in Subsection 3.1. There are more components in this test case than the previous ones (so the plots are more cluttered), but the FAST plots still present smoother, and more stabilized, time traces. On the other hand, the Reno plots exhibit the same sawtooth nature as before. In addition, FAST terminates around 8 seconds faster than Reno and has fewer `Packet` losses.

### 3.4 Test Case 2 Analytic Solution

In this subsection, we analytically predict the equilibrium queue sizes and `Flow` send rates after each `Flow` is introduced or terminated. These predictions (calculated in Subsection 3.4.1) are compared to the actual values (see Figure 12) in Subsection 3.4.2.

#### 3.4.1 Calculating Predicted Values

In terms of notation, we will say that $C = 1\text{KB} = 1024\text{B} = 8192\text{b}$ is the constant size of each `DataPacket`.

All three `Flows` in this analysis use TCP FAST, with window size $W_i = \frac{RTT_{i,min}}{RTT_i} W_i + \alpha$ (where $\alpha = 50$ is constant but $W_i$ and $RTT_i$ depend on the `Flow` $i$). The throughput of each `Flow` (in bps) will then be given by

$$x_i = \frac{\alpha C}{RTT_i - RTT_{i,min}}$$

from HW3 Problem 6.8.

We will denote $p_l$ as the queuing delay over `Link` $l$, and $q_i$ as the round-trip queuing delay for `Flow` $i$. There are then three distinct periods of steady state behavior to examine:

- 0.5s–10s

  During this time, all packets are buffered at $L1$ (the `Link` between $R1$ and $S1$ will not buffer because it has a larger capacity) and there is no queuing delay felt by $F1$ as no other flows are in the network. Thus, $x_1$ simply equals 10Mbps. Since there are no other flows in the network yet, $x_1 = \frac{\alpha(1\text{KB})}{q_1}$ (since $RTT_1 - RTT_{1,\text{min}} = p_1 = q_1$ in this case), which solves to $q_1 = \frac{4\alpha}{5120}\text{s}$. For $\alpha = 50$ (as is the case throughout Test Case 2), this gives $p_1 = q_1 = \frac{200}{5120}\text{s} = 39.063\text{ms}$. This in turn corresponds to an $L1$ buffer occupancy (from Little's Law) of $10\text{Mbps} \cdot 39.063\text{ms}/2 = 200.00\text{Kb} = 25.00$ data packets, as each `DataPacket` is 8Kb. The division by 2 comes from the fact that each packet is only delayed by half the round-trip time at each link.

- 10s–20s

  It is obvious that $L1$ will be bottlenecked with the introduction of $F2$. If we exclusively examine $L1$ the throughputs must sum to 10 Mbps.

  $$x_1 + x_2 = \frac{\alpha C}{p_1} + \frac{\alpha C}{p_1 - 39.063\text{ms}} = 10\text{Mbps}$$

  Where 39.063ms is the previous round-trip delay on $L1$ (as mentioned before), and the reasoning behind the subtraction (i.e. $p_1 - 39.063\text{ms}$ in the denominator) comes from the fact that $RTT_{2,\text{min}}$ includes 39.063ms as part of its min RTT estimate (see HW3 Problem 6.8). For $\alpha = 50$, we find that $p_1 = 102.27\text{ms}$ and the resulting throughputs are $x_1 = 3.911\text{Mbps}$ and $x_2 = 6.089\text{Mbps}$. This corresponds to a buffer size of 65.46 data packets at $L1$.

- 20s to F2's termination

  At this point, F3 has been initiated. If we then compute throughputs through $L1$ and $L3$ respectively, we arrive

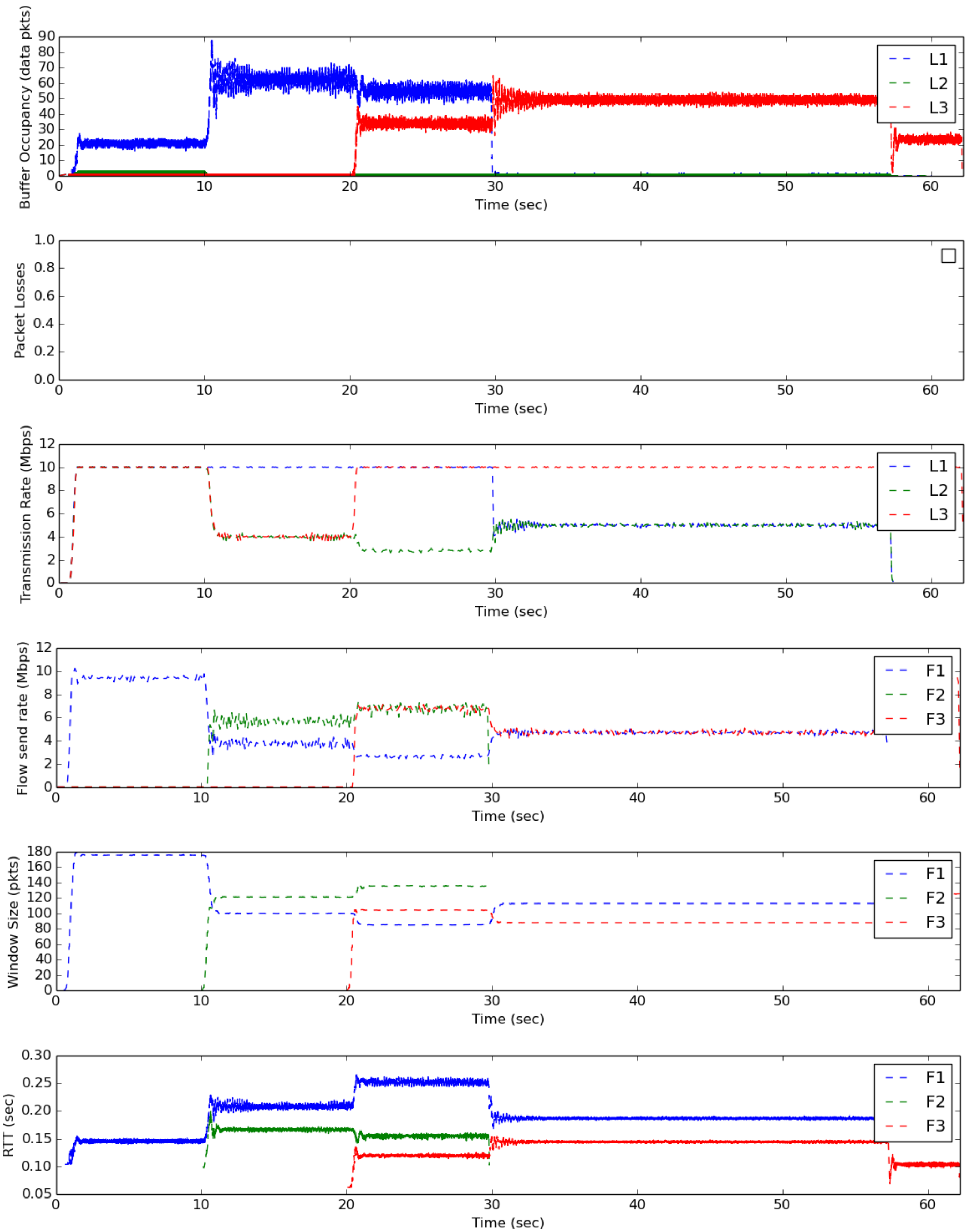


**Figure 12:** *Statistics of Test Case 2, with all* `Flows` *running TCP FAST. Note that only L1, L2, and L3's statistics are plotted in the* `Link`*-related graphs.*

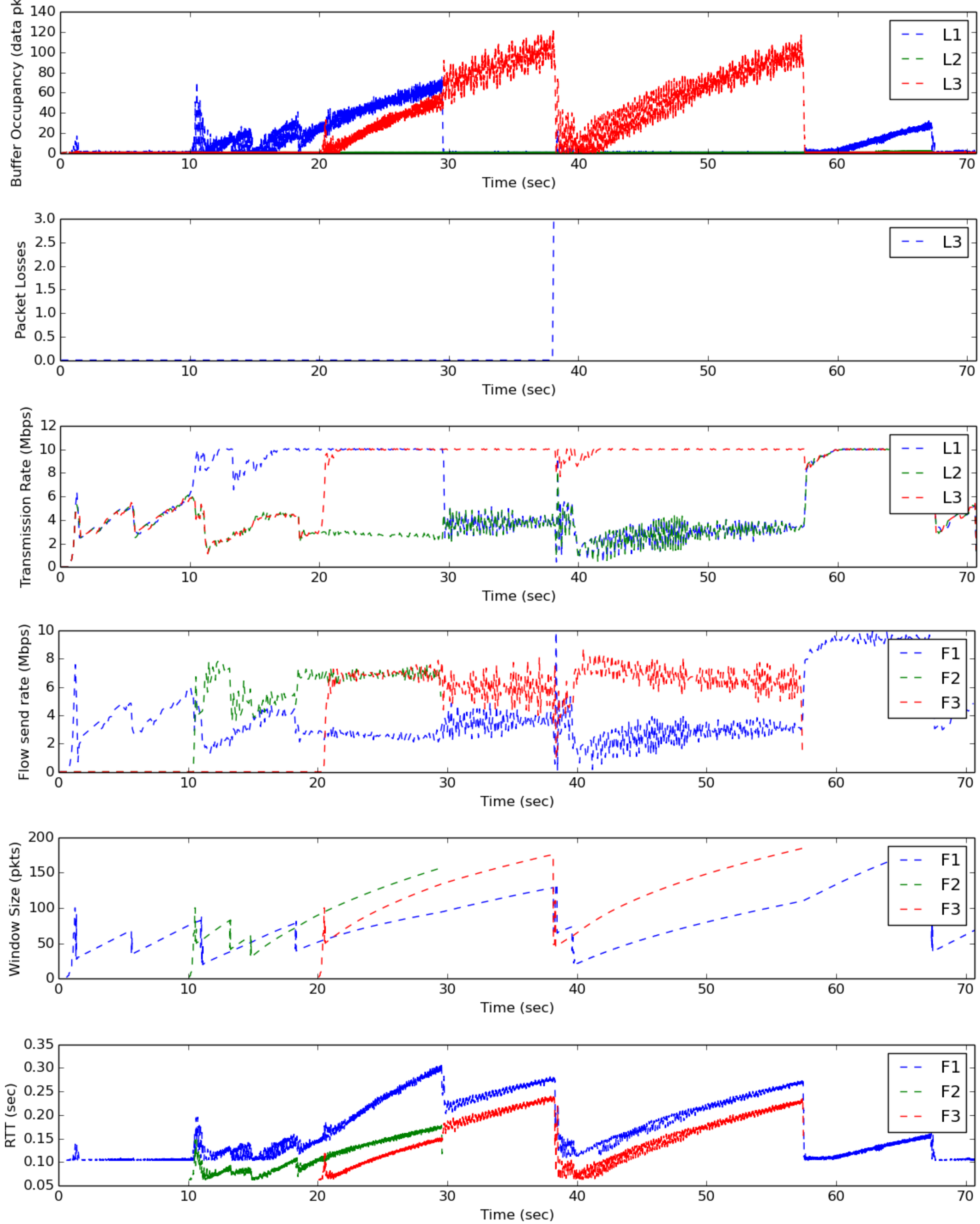


**Figure 13:** *Statistics of Test Case 2, with all* `Flows` *running TCP Reno. Note that only L1, L2, and L3's statistics are plotted in the* `Link`*-related graphs.*

at the following equations

$$x_1 + x_2 = \frac{\alpha C}{p_1 + p_3} + \frac{\alpha C}{p_1 - 39.063\text{ms}} = 10\text{Mbps}$$
$$x_1 + x_3 = \frac{\alpha C}{p_1 + p_3} + \frac{\alpha C}{p_3} = 10\text{Mbps}$$

where 39.063ms is the (round-trip) delay over `Link` $L1$ due to the preexisting `Flow` $F1$. With $\alpha = 50$, we find `Link` (round-trip) delays of $p_1 = 92.42$ms and $p_3 = 53.36$ms, and so the throughputs are $x_1 = 2.679$Mbps and $x_2 = x_3 = 7.321$Mbps. These correspond to buffer sizes of 59.15 `DataPackets` on $L1$ and 34.15 `DataPackets` on $L3$.

After the 20s mark, the three `Flows` send `Packets` at approximately the aforementioned steady rates, until a flow terminates. The work shown to compute how much each `Flow` has sent by each time is given in Table 1. To compute these, we just used the steady-state throughputs (in Mbps) for various time intervals (including some of the throughputs mentioned later on) and converted the total data each flow had to send into Megabits (280Mb for $F1$, 120Mb for $F2$, 240Mb for $F3$). Note that analytic predictions using equilibrium rates underpredict termination time because of the non-equilibrium transmission periods where the window is still adjusting.

| $t$ (s) | F1(Mb) | F2(Mb) | F3(Mb) |
|---|---|---|---|
| 0.5 | 0 | 0 | 0 |
| 10 | 95 | 0 | 0 |
| 20 | 134.11 | 60.89 | 0 |
| 28.07 | 155.74 | 120 | 59.11 |
| 52.92 | 280 | 120 | 183.37 |
| 58.58 | 280 | 120 | 240 |

**Table 1:** *Approximate amounts of data sent by each* `Flow` *by key times. The simulated termination times were* $\{52.92, 28.07, 58.58\}$ s *for the three flows* $\{F1, F2, F3\}$ *(respectively).*

After each `Flow` exits, we have the following rate and buffer capacity predictions:

- $t = 28.07$s — `Flow` 2 terminates (predicted).

  The network now bottlenecks at $L3$ with no queuing delay. Thus, the throughputs through $L3$ satisfy

  $$x_1 + x_3 = \frac{\alpha C}{p_3} + \frac{\alpha C}{p_3} = 10\text{Mbps}$$

  which gives (by symmetry) $x_1 = x_3 = 5$Mbps and so $p_3 = \frac{\alpha C}{5\text{Mbps}}$. With $\alpha = 50$, this yields $p_3 = 78.12$ms and an $L3$ buffer size of 50 `DataPackets` (simply twice the single-`Flow` case).

- $t = 52.95$s — `Flow` 1 terminates (predicted).

  All that remains at this point is just `Flow` $F3$ by itself, so $x_3 = 10$Mbps. This is effectively the same configuration as the 0.5s–10s case from before, and so we find $p_1 = \frac{4\alpha}{5120}$s $= 39.063$ms and a buffer size of 25 `DataPackets`.

- $t = 58.58$s — `Flow` 3 terminates (predicted).

### 3.4.2 Comparing Predicted and Actual Values

We summarize all of the earlier predicted results and compare them to the actual simulation results below:

- 0.5s - 10s
  - Expected F1 throughput: 10 Mbps. Actual: $\sim 9.6$ Mbps.
  - Expected L1 buffer occupancy: 25 pkts. Actual: $\sim 20$ pkts.
- 10s - 20s
  - Expected F1 throughput: 3.91 Mbps. Actual: $\sim 3.9$ Mbps.
  - Expected F2 throughput: 6.09 Mbps. Actual: $\sim 6.0$ Mbps.
  - Expected L1 buffer occupancy: 65 pkts. Actual: $\sim 60$ pkts.
- 20s - 30s
  - Expected F1 throughput: 2.68 Mbps. Actual: $\sim 2.7$ Mbps.
  - Expect F2 throughput = F3 throughput: 7.32 Mbps. Actual: $\sim 7.1$ Mbps.
  - Expected L1 buffer occupancy: 59 pkts. Actual: $\sim 55$ pkts.
  - Expected L3 buffer occupancy: 34 pkts. Actual: $\sim 33$ pkts.
- After `Flow` 2 drops out ($\sim 30$s to $\sim 57$ s)
  - Expected F1 throughput: 5 Mbps. Actual: $\sim 4.9$ Mbps.
  - Expected F3 throughput: 5 Mbps. Actual: $\sim 4.9$ Mbps.
  - Expected L1 buffer occupancy: 0 pkts. Actual: 0 pkts.
  - Expected L3 buffer occupancy: 50 pkts. Actual: $\sim 50$ pkts.
- After `Flow` 1 drops out ($\sim 57$s to $\sim 62$s)
  - Expected F3 throughput: 10 Mbps. Actual: $\sim 9.6$ Mbps.
  - Expected L3 buffer occupancy: 25 pkts. Actual: $\sim 23$ pkts.

- Predicted Termination Times
  - F2: 28.07 s (Actual: 29.72 s)
  - F1: 52.92 s (Actual: 57.43 s)
  - F3: 58.58 s (Actual: 62.23 s)

# 4 Summary

This network simulator successfully incorporates a variety of features – including dynamic routing, TCP Reno, and TCP FAST – into an event-driven model with statistics tracking. The three test cases examined have the expected qualitative behavior in terms of their time traces. An additional analytic check on Test Case 2 further confirms the accuracy of the simulator.

# 5 GitHub Repository

A link to our GitHub repository is included in [4].

# 6 Acknowledgments

The authors would like to acknowledge Professor Steven Low and TA Cody Han for their instruction and help throughout this project. This paper was completed as part of CS/EE 143, a class on communication networks at the California Institute of Technology.

# References

[1] Low S. "Congestion Control & Optimization." Web. 2011. URL.

[2] "Javis in Action: Fast Recovery Algorithm." Web. Accessed Nov 2015. URL.

[3] Low S. "FAST TCP: Motivation, Architecture, Algorithms, Performance." IEEE/ACM Transactions on Networking pp. 1246-1259. Dec 2006. URL.

[4] Bhasin L., Luo D., Su Y., and Yang S. "lakshbhasin/network_simulator." Web. Dec 2 2015. URL.