Network Equilibria: An Applied Approach

Moya Chen Louis O'Bryan Victor Duan

December 10, 2013

CS 143 Communication Networks
Final Project, Fall Term 2013

Contents

1 Introduction

Networking theory is an important part to the function of the Internet. Congestion control algorithms can heavily influence the speed of a connection.

In this paper, we describe the implementation of a network simulator including its architecture and design choices. We also include graphs of this simulator as run on a series of test cases. Lastly, we give a mathematical analysis for one of the congestion algorithms, as applied to one of the cases.

2 Simulator Description

Our network simulator is written in C++.

The network consists of several different types of network classes as well as a Controller to monitor the simulation.

We use event-based simulation, with a global event queue, to advance actions in our program.

For input, we take an XML file as input and use the pugixml C++ XML parser to read the file and create necessary network objects. The program uses these objects to simulate the network until all the flows finish.

The congestion control algorithms we implemented were TCP Reno and TCP Vegas.

As the simulation runs, we collect various statistics. At the end of the simulation, we use gnuplot to create figures based off of collected data.

2.1 Design Decisions

We chose to write the simulator in C++ for several reasons. C++ is a powerful and fast object oriented programming language which suited our architecture plan well. We wrote classes for different network objects (Link, Flow, etc.) as well as simulation objects (Controller, CongestionAlgorithm, etc.). By using C++s inheritance, we were able to group objects with similar behavior (ex. Host and Router) and create abstraction hierarchies (ex. Node, Packet). C++ allowed for writing these classes with encapsulation.

Because the network that we are simulating is based on discrete events, we chose to use event-based simulation. Continuous simulation did not make sense as it would be very CPU heavy and take a longer time to run on our computers. Because we knew that we would be spending a fair amount of time running code, we figured that event-based simulation would be the most efficient.

We chose the pugixml parser because it is lightweight and has a simple, open-source library. The lightweightness was particularly attractive because we did not want the simulator to be spending most of its runtime parsing input files. Furthermore, the pugixml library was easy to use. XML was the chosen input format because as a universally recognized format it was easy to make it fit our desired system parameters. Thus, we ended up using an XML DOM type input, as it could easily describe the objects we wanted for the simulation.

We chose two TCP congestion control algorithms: TCP Reno and TCP Vegas. We wanted to choose one loss-based congestion control algorithm (Reno) and one delay-based congestion control algorithm (Vegas). We chose Reno because among the loss-based control algorithms (Reno and Tahoe) because it included more advanced congestion control through the use of fast_recover. Further equations for the analysis are in the notes.

We chose gnuplot to generate our plots because like the pugixml parser, it was easy to use and provided us with the flexibility that we wanted to make the images that we desired.

3 Architecture

The network design was split into the groups of classes as follows: Simulation, Network, and Input. The Simulation group contains the classes Event, Scheduler, and Controller. These classes are used to hold the network objects as well as manage the event-based simulation. The Network group contains the classes Node, Host, Router, Link, Buffer, Router, RoutingTable, Packet, RoutingPacket, RouterRoutingPacket, HostRoutingPacket, DataPacket, AckPacket, Flow, CongestionAlgorithm, SLOW_START, TCP_TAHOE, TCP_RENO, Vegas, and Cubic. These classes are used to simulate network objects and their behavior. Their exact behavior and hierarchy are described in more detail below. Finally, the Input group of classes contains InputParser, SystemInfo, HostInfo, RouterInfo, LinkInfo, and FlowInfo. The InputParser is used to read in information from the input XML file and that information is used in Controller to create the objects.

3.1 Simulation Classes

Classes in this category: Controller, Scheduler, Event

The Simulation classes function to keep track of high-level system parameters as the simulation runs. The Controller class has a few main tasks. One instance of this class is created upon program start. (For simplicity, we will refer to this one instance of the Controller class as the controller.) It is the Controller to which the InputParser adds system objects. (See Input/Output below.) The controller is what keeps track of system time. It also contains an instance of Scheduler (hence forth called the scheduler). It is also the object that interfaces with network objects when the network objects want to make events. The controller also initializes the first routing update and system print events. After initialization, these Events will schedule the next instance of themsleves.

In essence, the controller runs the entire simulation.

The scheduler contains a priority queue, ordered by event time, which contains all of the systems events. In this scheduler, there is a doNext() function which is called by the controller until the system has finished running of all of its flows. This function enacts the next event in the queue.

The Event class contains 3 things: a timestamp for when the event should be executed, a function pointer to the function scheduled, and a pointer to arguments that the function may need for execution. When an Event is executed, the function is called with the arguments.

To do event-based simulation, each element in the system that wants to call some function in the future will create a new Event with the time of execution, the function, and appropriate arguments. It will then call the controller's add(Event *e) function in order to put this new Event into the scheduler's queue. Until the simulation finishes, the controller pops off the top Event off of the scheduler queue, executing it. It then checks if there are any flows left. If there still exist flows, it pops the top Even off of the scheduler queue again, and so on and so forth.

In order to make sure that periodic events like router updates and system snapshot happen, the controller is initialized with the first Event of these. For system snapshot, the controller iterates through all links, routers, hosts, and flows, writing appropriate data to output files and resetting necessary statistics. Router updates trigger the creation of routing packets at each router, which are put into the different links. When these packets are recieved by routers and hosts, they create packets as appropriate, sending them back to the routers which are used for updates.

3.2 Network Classes

The Network classes are used to hold information of the various network objects. They are split into a few general subgroups: Node, Link, Packet. Node is further split into Host and Routing. Because Host and Routing is fairly involved, we will explain these in detail as well.

Node The Node subgroup consists of Node, Host, and Router. Node is an abstract class that contains the id of the node as well as the list of links to which is connected. The Node class guarantees a function handlePacket(Packet *p). We choose to implement this abstract class because class Router and Host both have this as part of their interface.

Router The Routing subsubgroup consists of Router and RoutingTable. Each router contains its own RoutingTable pointer which describes the behavior it will follow when routing packets. The RoutingTable itself is a map from Node pointers to pairs of weight and Link pointers. The key is the destination Node of the packet to route. It is a Node (as opposed to Host) because while all DataPackets must be sent to Hosts, RoutingPackets can also be sent to Routers. The weights are then the weight of that path and the Link is the next link to follow on that path. In addition to the routing table, the primary function of the Router class is to handle packets that it receives. Data and ack packets are simply forwarded to the next link in the

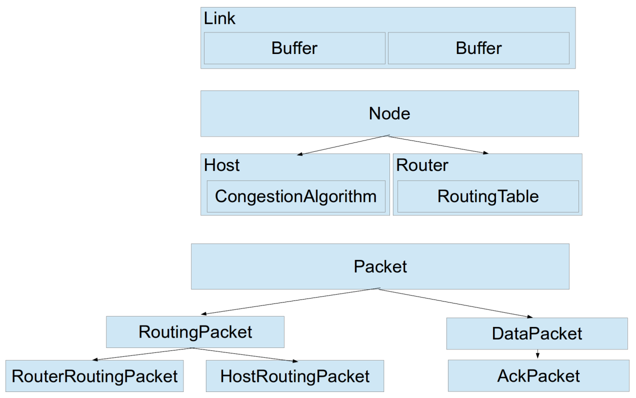

Figure 1: An image of hierarchy of network object classes in our system. Arrows denote inheritance. Boxes inside of boxes denote a single instance of an object of a class being initiated as part of every instance of another class. (Ex. a link always contains 2 buffers)

path. Routing packets, on the other hand, are used to update the routers own routing table. This is done in the routing table update steps every routingTableUpdateTime.

As far as the actual routing updates are done, we implement the Bellman-Ford algorithm. We chose to use this algorithm because it utilizes local knowledge of the graph as opposed to Dijkstras algorithm, which depends on global knowledge of the network. As such, it seems more natural to use Bellman-Ford for the routing table updates. Dynamic routing is calculated through a combination of buffer delay and propagation delay. Buffer weights are only calculated once per routing update period so that the routing table weights – stored as doubles – stabilize. This behavior mimics real routing table updates because otherwise the routing weights constantly change by slight amounts.

Host The Host subsubgroup consists of the Host class, the Flow class, as well as all of the congestion control algorithm classes. These classes together are responsible for the creation and tracking of data and acknowledgement packets sent throughout the system.

The Host class is responsible for data collection and passing packets from Links to Flows.

The Flow class holds onto the information needed to handle the flow, including size, progress, outstanding packets, acks received, and relevant stat fields. Flows also include an instance of CongestionAlgorithm. The Flows create DataPackets upon request from the CongestionAlgorithm, which it sends to its source Host for transfer. When a DataPacket is received by the recieve Host, the Flow keeps track of which id it should use to create an AckPacket for in response to the reception of that DataPacket. When a Flow is added to the Controller, an initial event is scheduled to cause the first data packet transmission.

CongestionAlgorithm is the abstract base class used to simulate the congestion control algorithms. It holds onto fields necessary for congestion control such as window size and ssthresh. We then have several different congestion control algorithms that implement various types of TCP congestion control.

The simplest of these is SLOW_START, which inherits from CongestionAlgorithm. SLOW_START implements slow start for that part of TCP, and then uses very basic congestion avoidance. While not in slow start, it increments cwnd by 1/cwnd every ack. Otherwise, it increments cwnd by 1. When packets time out, it sets ssthresh to half cwnd and sets cwnd to 1. With SLOW_START, it is possible to see network behavior while running just only multiplictive increase and congestion avoidance.

Next, we have TCP_TAHOE, which inherits from SLOW_START. TCP_TAHOE implements fast_retransmit. In fast_retransmit, when 3 duplicate ACKs are recieved, it is treated the same as a timeout. Rather than having to wait a full round trip time in order to send a new packet, TCP_TAHOE detects packets sooner than SLOW_START. Thus, we can implement Additive Increase, Multiplicative Decrease efficiently.

After this, we have TCP_RENO, which then inherits from TCP_TAHOE. This adds fast_recovery. Rather than sending a single packet upon detecting packet loss from triple acknowledgement, TCP_RENO includes a modified algorithm which temporarily inflates the window size to compensate for packets as they go missing. Thus, rather than collapsing the window size to 1 as soon as packet loss is detected, TCP_RENO can adjust to a more reasonable size and adjust from there.

We also have Vegas, which inherits from TCP_RENO. The Vegas algorithm is very similar to TCP_RENO except that it has a different window update rule in Congestion Avoidance. Rather than using packet loss as its main metric, it uses packet delay. This allows more fine-grained tuning of network status as packets are being sent and recieved. The idea of Vegas is that there is some ideal range for the value of the current window size divided by what the round trip time would be of the system if there were no queuing delays compared to if there were. By keeping this difference within a constant set of values, we can ensure that window size is not too big or not too small.

Finally, we have Cubic, which inherits from SLOW_START. It is a TCP algorithm based on a window update algorithm using a Cubic function. While this class currently does not currently run successfully on simulation, we have included it nonetheless for possible future work.

Link The Link subgroup consists of Link and Buffer. The Buffer class is used to simulate the buffers on the links. Each link has two buffers, one in each direction and each have the capacity described in the input file. It also consists of pointers to the Nodes to which an instance of Link is attached.

The buffers are implemented as FIFO queues of Packets. The Link class uses the buffers and its own fields to manage the links activities.

The Link also includes various functions to handle packet transfer. We describe them in the section Packet Transfer below.

Packet The Packet subgroup consists of the large variety of Packet classes. All of these stem from the abstract Packet class. Packet contains core information of the packet such as id, size, source, and destination, and packet type.

There are two subgroups of Packets: RoutingPacket and Datapacket.

Inheriting from Packet, we have RoutingPacket, which are the packets used to update the Router's RoutingTables. In class RoutingPacket, we add a field to describe the link that a RoutingPacket packet came from. Inheriting from RoutingPacket, we have RouterRoutingPacket and HostRoutingPacket. HostRoutingPackets tell Routers that the hosts still exist at that link. RouterRoutingPackets contain routing tables to tell the routers whether or not they have to update their routing tables.

In class DataPacket, we inherit from Packet. Data packets use extra information for the flow it is part of and the start time, which is used for calculating stats such as RTT and packet delay. Finally, we have AckPacket, which inherits from DataPacket. We inherit from AckPacket as many of the properties (such as flow) are the same between the two. This also aids in debugging, as we can print which DataPacket triggered the creation of which AckPacket. DataPacket and AckPacket are both created by the Flow.

3.2.1 Packet Transfer

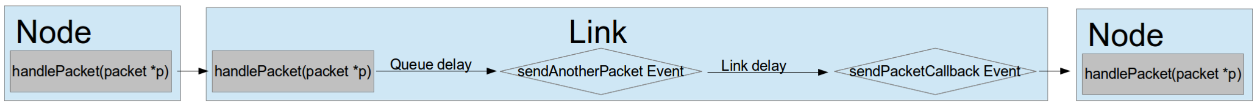

Figure 2: Packet transfer process between nodes (hosts, routers) and links. Arrows here represent packet transfer in the system. Arrowhead direction denotes passing of packets between functions. Arrow label denotes time offset added, as perceived by scheduler. (Time offset is zero when not labeled.)

Because the packet transfer structure may be a bit confusing without explaination, we provide one here.

When a node puts a packet into the link, it will call that link's handlePacket(Packet *p) function. Because there may be other packets traveling the link at the same time, we first schedule an event which calls sendAnotherPacket on the link. This event is scheduled at a time so that delays that may be incurred at the head of a link (queue delay, processing delay) are included.

When sendAnotherPacket is called, we schedule an event that calls sendPacketCallback. This simulates the delay time that a link would have in transmitting the packet over its length.

Finally, when sendAnotherPacket is called again, we know the packet has finished transmitting the length of the link. Thus, it can be sent on to the next node in the system.

3.3 Input/Output

The InputParser class uses the pugixml C++ XML parsing ability to parse input files and store the information in lists of the various Info objects. One thing to note is that pugixml does not provide any method of checking the XML file against an XSD schema. As such, the expected input and some behavior is described in much greater detail in the networkSchema.txt file in the InputParser directory.

The SystemInfo class is used as the parent class for the other Info classes. All it holds is the printing flag for the object. The other Info classes each contain values specific to the type of object it is designed to hold information of. For example, LinkInfo will hold members such as linkRate. These are explained in more detail in the networkSchema.txt file.

If the input file is formatted correctly, the InputParser can call its run function, which takes in references to the values it will set, to parse the file, set global parameter fields, and create network objects. The global parameter fields set are snapshotTime, routingUpdateTime, hosts, routers, links, flows, and plotOptions. snapshotTime is the time interval between stat-collecting events. The smaller this is, the finer the resolution. routingUpdateTime is the time interval between updating the routing tables. hosts, routers, links, and flows are lists of their corresponding Info objects that will be used to create the objects later. plotOptions sets various plot settings for gnuplot for the output graphs.

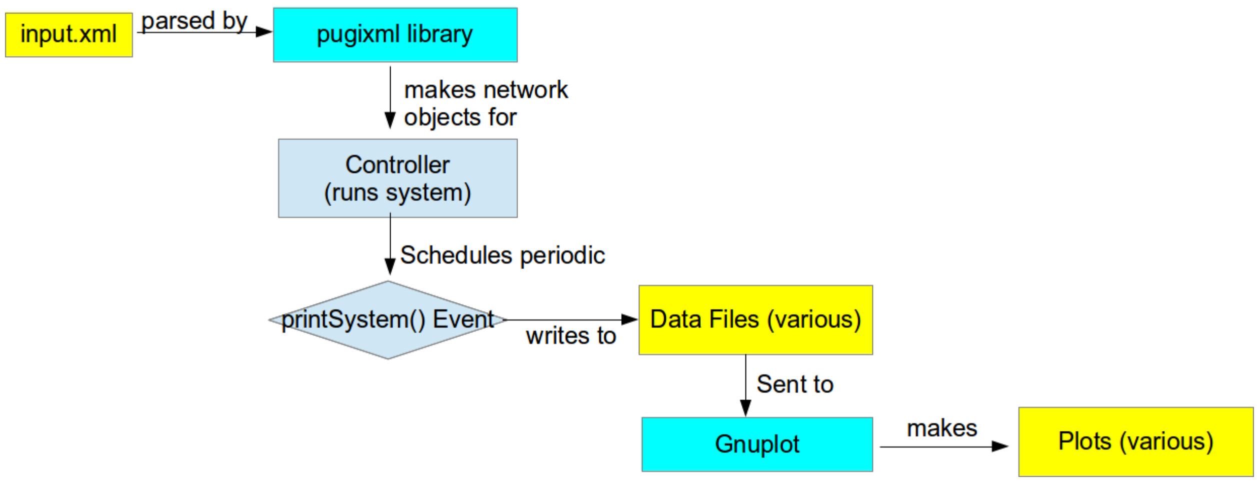

Figure 3: Input/output structure of our system. Yellow denotes input/output files. Cyan denotes helper libraries. Blue rectangle denotes simulator objects. Blue diamond denotes simulator events.

System output is done via periodic printing to data files. At the start of the simulation, the controller schedules an event which calls printSystem(). This function iterates through all links, flows, and hosts, requesting and then resetting data as appropriate. This data is written to a series of output files. At the end of the call to printSystem(), the printSystem() function will schedule an event to call itself again at a time snapshotTime later. Once the controller has finished running the simulation, it will make a function call to appropriate library functions in gnuplot. This will take the data from the files we have written to above and graph it.

4 Usage guide

The makefile is in the top level directory. You do not need to specify a target; the "all" target is the one you want anyway.

It may be necessary to create a "bin" folder in the top level directory if one does not already exist.

Once the code has compiled, run "bin/test input.xml" or whatever input file you have as the first command line argument after the program name. An example of sample input and documentation of the schema is in InputParser/networkSchema.txt. Once the simulation is done, graphs will be made in the Output directory.

For convenience, we have included testcase0.xml, testcase1.xml, and testcase2.xml. The only lines that need to be changed are the ones relating to the congestion algorithm type.

5 Mathematical Analysis

* Test Case 2 can be solved analytically. Set up the TCP equations that TCP Vegas / FAST-TCP are engineered to solve, and then find (and justify!) the equilibrium point for each phase of Test Case 2's operation. Compare your calculated results to your experimental results; if they are wildly different, discuss why this might be so.

In this section, we provide a mathematical analysis of TCP Vegas as applied to testcase 2 of our system.

By example 9 of chapter 3 (3.9), the equilibrium solution to Vegas is the same as the optimal minimization of $\sum_{i=0}^{n} \alpha_i \log(x_i)$ subject to the link capacity constraints $Rx <= c$. In the first time interval, there is only one flow, so this problem becomes

$$\frac{\alpha_1}{x_1} = \mu_1 + \mu_2 + \mu_3$$
$$\mu_1(x_1 - 10) = 0$$
$$\mu_2(x_1 - 10) = 0$$
$$\mu_3(x_1 - 10) = 0$$

Solving these equations gives the solutions $x_1 = 10$Mbps, and $\mu_1 = 0.055, \mu_2 = 0, \mu_3 = 0$ (although these may be chosen arbitrarily to satisfy nonnegativity and the top equation). This rate matches the realized rate of flow 1 during simulation after the initial slow start.

The propagation delay d_1 is 100ms because each of the 5 links along the path from S1 to T1 has the same delay of 10ms. The base RTT is approximately 110ms because of the queueing delay. By Little's law, the equilibrium queueing delay is α/x_1, where in our simulation, $\alpha = 0.55 packets/ms$. This is equal to $\frac{0.55packets/ms}{10Mbps} * \frac{1024B}{packet} * \frac{Mb}{10^6 b} * \frac{8b}{B} * \frac{10^3 ms}{s} * (110ms) = 50ms$. So the equilibrium RTT is 150 ms. This calculation also agrees with the round trip time in our simulation. The corresponding window size is $150ms * 10Mbps = 183 packets$.

When flow 2 enters, its base RTT includes the queueing delay at link 1. The queueing delay is approximately 50 ms by the above, assuming that the queueing delay is the same as the previous equilibrium queueing delay. In addition, the propagation delay is approximately 66 ms. The KKT conditions to the minimization become

$$\frac{\alpha_1}{x_1} = \mu_1 + \mu_2 + \mu_3$$
$$\frac{\alpha_2}{x_2} = \mu_1$$
$$\mu_1(x_1 + x_2 - 10) = 0$$
$$\mu_2(x_1 - 10) = 0$$
$$\mu_3(x_1 - 10) = 0$$

Solving these equations gives the solutions $x_1 = 5$Mbps, $x_2 = 5$ Mbps, and $\mu_1 = 0.110, \mu_2 = 0, \mu_3 = 0$. The simulation's flow rates of flow 1 and flow 2 converge toward this common value in the period before flow 3 is added, although they do not quite reach it before that time.

With the same base RTT for flow 1, the new equilibrium queueing delay is $\frac{0.55packets/ms}{5Mbps} * \frac{1024B}{packet} * \frac{Mb}{10^6 b} * \frac{8b}{B} * \frac{10^3 ms}{s} * (110ms) = 100ms$. The new RTT of flow 1 is 200 ms, which agrees with the simulation's RTT. The equilibrium queueing delay for flow 2 is $\frac{0.55packets/ms}{5Mbps} * \frac{1024B}{packet} * \frac{Mb}{10^6 b} * \frac{8b}{B} * \frac{10^3 ms}{s} * (66ms + 50ms) = 105ms$

since the base RTT is $66ms + 50ms$ (the propagation delay plus the equilibrium queueing delay from the first interval). So the equilibrium RTT for flow 2 is $105ms + 66ms = 171ms$. This RTT does not match the simulation exactly, which produces round trip times of about 150 ms, but it is within a reasonable margin of error. The corresponding window sizes for flows 1 and 2 are 122 packets and 104 packets. In turn, these measurements slightly differ from the simulation's values. A possible reason for shorter round trip times is that the links in our simulation allowed for higher throughputs than advertised on the simulation specifications because of the buffers on both ends, as we explained previously.

When flow 3 enters, its base RTT only depends on the propagation delay to T3 since the buffer at L3 is empty. So its base RTT is 66 ms, and its propagation delay is 60 ms. The KKT conditions become

$$\frac{\alpha_1}{x_1} = \mu_1 + \mu_2 + \mu_3$$

$$\frac{\alpha_2}{x_2} = \mu_1$$

$$\frac{\alpha_3}{x_3} = \mu_3$$

$$\mu_1(x_1 + x_2 - 10) = 0$$

$$\mu_2(x_1 - 10) = 0$$

$$\mu_3(x_1 + x_3 - 10) = 0$$

The solution to these equations is $x_1 = 3.33$ Mbps, $x_2 = 6.67$ Mbps, $x_3 = 6.67$ Mbps, and $\mu_1 = .0825, \mu_2 = 0, \mu_3 = .0825$. These rates approximately agree with the simulation.

The queueing delays are

$$q_1 = \frac{0.55 packets/ms}{3.33Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (110ms) = 150ms$$

$$q_2 = \frac{0.55 packets/ms}{6.67Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms + 50ms) = 78ms$$

$$q_3 = \frac{0.55 packets/ms}{6.67Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms) = 45ms$$

and the corresponding window sizes are

$$w_1 = 102 packets$$

$$w_2 = 117 packets$$

$$w_3 = 90 packets$$

These correspond to RTT's of $T_1 = 250$ ms, $T_2 = 144$ ms, and $T_3 = 111$ ms. These results agree approximately with the simulation's results, however the simulation shows round trip times slightly lower. The windows sizes also approximately agree, however, they seem to converge to slightly lower results as well. Again, this could be due to the altered link buffer behavior.

When flow 2 finishes, the new equilibrium equations are analogous to those given above for the case when only flow 1 and flow 2 are in the system; $x_1 = 5$ Mbps and $x_3 = 5$ Mbps, and $\mu_1 = 0, \mu_2 = 0, \mu_3 = .110$. The new queueing delays are

$$q_1 = \frac{0.55 packets/ms}{5Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (110ms) = 100ms$$

$$q_1 = \frac{0.55 packets/ms}{5Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms) = 60ms$$

and the window sizes are

$$w_1 = 122packets$$

$$w_2 = 77packets$$

The new RTT's are $T_1 = 200$ ms and $T_2 = 126$ ms. Again, these do not exactly match the simulation's results, however they are not too far away from the results, which were approximately 225 ms and 140 ms. The simulation window sizes were about 135 packets for flow 1 and 60 packets for flow 2. These results may be lower again due to the link buffers.

Finally, when flow 3 is the only flow left, the KKT equations become

$$\frac{\alpha_3}{x_3} = \mu_3$$

$$\mu_3(x_3 - 10) = 0$$

The solution is $x_3 = 10$ Mbps and $\mu_1 = 0, \mu_2 = 0, \mu_3 = .055$. The new queueing delay is

$$q_3 = \frac{0.55packets/ms}{10Mbps} \frac{1024B}{packet} \frac{Mb}{10^6b} \frac{8b}{B} \frac{10^3ms}{s}(66ms) = 30ms$$

and the new window size is

$$w_3 = 117packets$$

6 Workflow and Project Thoughts

For this project, we often met together and debugged in person. More people thinking about the same thing helped things to get done more quickly. On the other hand, there were times that we focused too intensely on one idea when it would have been more useful to work in parallel on different items. For the most part however, the group worked together efficiently.

Because we were such a small team, we had a few luxuries. Since we mostly worked on code in person, it was largely unnecessary to have a ticking system. Additionally, although there was some large scale division of labor (for example, Victor worked mostly on input, Louis on routing, and Moya on the system's control mechanism) there was a lot of cross-over, which was nice for when schedules became less flexible.

At various points in the project, we had some issues with version control. While it was useful for times when we had messed up code very badly and needed to revert, there were quite a few issues with people working on the same files. Because we were all fairly inexperienced with GIT, we figured it would be unnecessary to work on different branches. There were also multiple issues with people staging and committing more files than necessary, causing headaches with conflicts. Had members on the team been more experienced with GIT, the process would have been more streamlined.

Overall, while the team was decent with staying on deadlines, especially towards the second half of the term, there were a few weeks where progress was slow because of other committments. However, given how we had operational code for the project, we were still fairly good at staying on task.

7 Test Case Simulation Plots and Commentary

In this section, we include the plots resulting from our simulation running on various testcases.

7.1 TCP Vegas

For TCP Vegas, we see, more or less, the expected behavior on each of the test cases.

7.1.1 Test Case 0

We only have one link to analyze, and we see that the rate across this link goes through a brief period of slow start at the beginning, fluctuates for a little bit as it converges, and then stays relatively constant at 10 Mbps. This accurately reflects Vegas behavior, which goes through slow start and then should stabilize as the diff parameter (cwnd * (1/rttmin - 1/rttcurrent)) gets between alpha and beta. As for the actual value, we see that it stabilizes to something like 10.6 Mbps. This makes sense because of the way our network is set up. Our links are implemented such that it has a buffer on each side (for each direction). Each buffer has the capacity that the link should have. So, the theoretical maximum of our link rates is actually 2 times the specified rate. The 10.6 Mbps rate on link one makes sense because the algorithm will stabilize naturally to send data across the link at the capacity. So, it will send data packets from host 1 to host 2 at 10 Mbps. The other 0.6 Mbps is a result of the ack packets being returned in the other direction. Because ack packets are about 6% the size of data packets (64 vs 1024 bytes) this result makes sense.

The buffer occupancy graph oscillates between two values. This happens even with a constant window size because the host will send out its window size (which corresponds to the higher buffer occupancy), and then wait until it receives acknowledgements. In that time, the buffer occupancy is slightly lower.

The packet loss of this test case is 0 because no packets are ever lost, fast-retransmit occurs first. There are no other flows, so there is not enough congestion to actually lose packets.

The flow send rate, window size, and packet delay all behave as expected, with a slow start rise, followed by minor fluctuations during the convergence, and finishing with a stable value. The flow rate of 10 Mbps is as expected, because that is the bottleneck link's rate.

7.1.2 Test Case 1

The link rates behave pretty similarly to test cae 0, except with an oscillating behavior. During the first 5 seconds, the routing table shows the same weights for both paths through the network because there is no congestion. So, the flow goes through the same slow start an convergence pattern, and then stabilizes to about 10 Mbps from host 1 to host 2 and 0.6 Mbps in the opposite direction. Our network is implemented that it finds any path with the min weight and forwards packets along that. It happens that on the way from host 1 to host 2, the data packets are forwarded along link 2 at 10 Mbps, which is the expected rate. On the return path, the ack packets are forwarded along link 1 (the choice is arbitrary). So, there is about 0.6 Mbps activity on link 1, similar to test case 0. After the next routing table update, the links behave in an oscillating pattern. After 5 seconds, the lower path is more congested, so the tables change to reflect this and forward all packets along the upper path, resulting in 10.6 Mbps activity along link 1. After this, the upper path is more congested and the routing tables change to forward packets along the lower path, and so on until completion.

The buffer occupancy graphs show that the link with the congestion has a growing buffer occupancy from the beginning of the period (separated by routing table updates) and then stabilizes for the rest of the period. The growing behavior is due to the routing table updates. The paths the packets follow is changed, so there is some time as the packets go through the link and gradually fill the buffer until they reach the stablized value.

Again, packet loss is 0 throughout the duration of the simulation because no packets are lost, retransmits only happen as result of duplicate acknowledgements.

Flow rate, window size, and packet delay all follow the behavior seen in link rate and buffer occupancy. This, of course, makes sense as they are all related. The flow rate of 10 Mbps also reflects the expected behavior of Vegas.

7.1.3 Test Case 2

The analysis of test case 2 with Vegas is done in the mathematical analysis section.

7.2 TCP Reno

For TCP Reno, we see, more or less, the expected behavior on each of the test cases.

7.2.1 Test Case 0

The link rate is again at the expected rate of 10.6 Mbps. It goes through the same slow start rise at the beginning as Vegas, after which the behavior is the same.

The buffer occupancy plots reflect the behavior of TCP Reno. They rise until 3 duplicate acknowledgements are received, at which point the window size is set to 1, and so the buffer occupancy is similarly reduced. If a packet is lost, the ssthresh is also changed, reflected in the decreasing bottoms of the rising arcs.

The packet loss is 0 most of the time, but hits a few packet losses in the slow start phase of Reno, as expected. After this, because it is a loss-based congestion control, there are small blips of packet loss, which correspond to the packet drops after the window gets sufficiently large.

The flow send rate, window size, and packet delay all behave as expected, reflecting the behavior of the link rate and buffer occupancy.

7.2.2 Test Case 1

The link rate in test case 1 shows Reno's behavior in the oscillating network. The link rates grow quickly in the beginning during slow start, then it reaches the same state as Vegas, where the packets are first forwarded along the bottom path (link 2) until the first routing table update, at which point they switch to the other path, and back and forth. The rising behavior of the rate is also expected, as it grows until it reaches the max of about 10.6 Mbps (as before) before dropping, which is Reno's design.

The buffer occupancy behaves somewhat as expected. After slow start, it stays quite low for a while, which is expected, and then starts growing on one of the fork links between 10 and 15 seconds in. Similar behavior occurs before 20 seconds, and before 25 seconds. This reflects the behavior of links where the rates grow until they reach a max and get reduced.

The packet loss has some loss in the initial slow start phase, but no losses afterward. This is expected because the routing table updates change the paths of the packets, so none are actally dropped, but congestion control may happen due to duplicate acks.

The flow send rate, window size, and packet delay all behave as expected, reflecting the behavior of the link rate and buffer occupancy.

7.2.3 Test Case 2

The link rate in test case 2 shows Reno's behavior when there are mutiple flows competing for resources. The link rates grow quickly in the beginning during slow start, and then the links all achieve rates of about 10.6 Mbps, because there is only one flow going across all 3 links, so this is the expected behavior. Once the second flow is added, the behavior changes. Because both flow 1 and 2 use link 1, that is still achieving its 10.6 Mbps rate. However, flow 1 continues on to use links 2 and 3. Because the bottleneck link 1 is congested, the rates on those links are lower, as expected. With the addition of the third flow, link 3 has the same behavior as link 1 because it is essentially the same case. However, link 2 has the same behavior as before, because only flow 1 is crossing it. As flow 2 ends, we see behavior similar to when it was only flow 1 and 2, which makes sense because the simulation is symmetric. When flow 2 ends, all the link flow rates increase back up to the initial state, which is all links with rates of 10 Mbps.

The buffer occupancy behaves somewhat as expected. After slow start, link 1 starts to fill because flow 1 sends it into the network and bottlenecks at link 1, and the other 2 are unaffected. When the second flow joins, it is the same kind of behavior, as we expect, because it is still just the one bottleneck link. With the addition of the third flow, we expect occupancy in buffers of links 1 and 3. We see that they overlap for a few seconds (25-30), where both the buffers are growing in size, as expected. As the second flow ends, the buffer on link 1 drops in size, as expected. Because flow 1 is forced to send at a lower rate due to the congestion at link 3, there is also no buffer waiting at link 1, which is the behavior we see. After flow 3 finishes, we see that it goes back to the initial behavior of link 1 growing in size.

The packet loss is as we expect. We see some losses during slow start, and again when flow 3 joins, because there are too many packets entering the system and some are dropped.

The packet delay shows the behavior we expect. We see that it starts with rising delay for flow 1. As flow 2 is added in, they both grow, and flow 1 stays above it because while they have the same congestion on the outer links and the shared link 1, flow 1 also has to go through links 2 and 3, which is added delay. Similar behavior is seen with all 3 flows, though flow 3 has lower delay than flow 2. This is because the queueing delay for flow 1 is all in link 1, and there are fewer waiting packets in link 3, which allows for flow 3 to achieve a lower delay. When flow 2 ends, we see similar behavior as to when it was only flows 1 and 2, with flow 1 having longer delays due to longer path.

The flow send rate and window size pretty much behave as expected, reflecting the results seen in other stats.

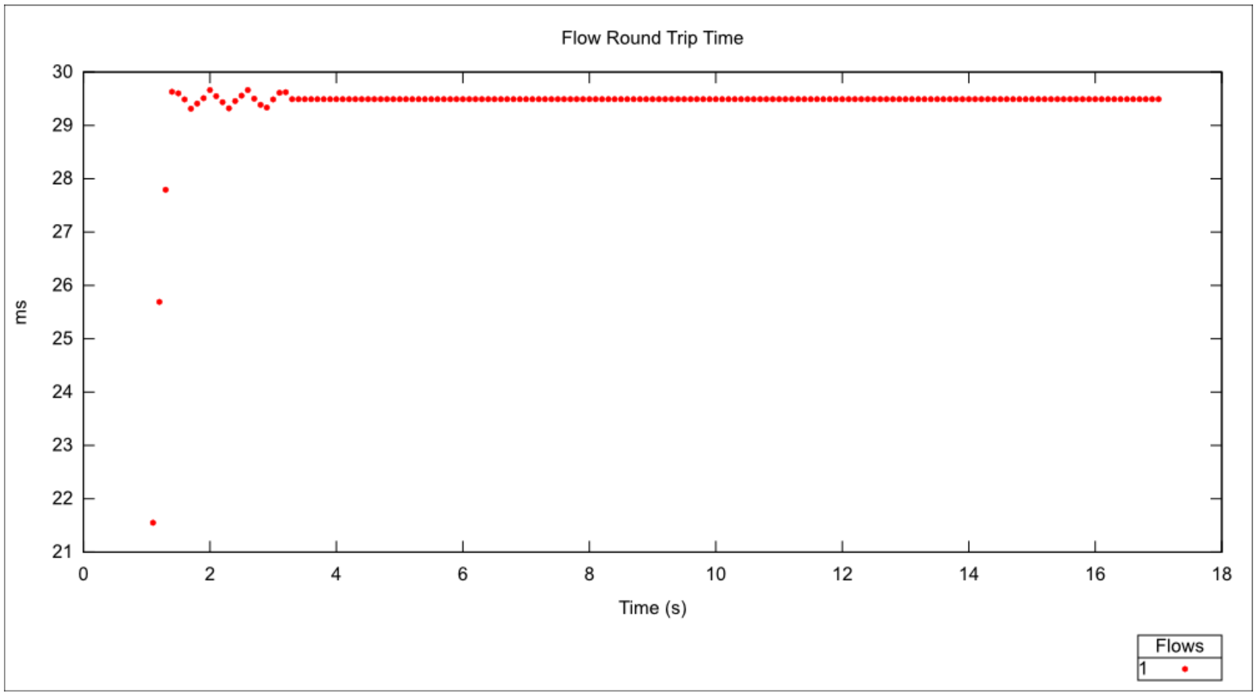

Figure 4: TCP VEGAS, Case 0: Flow Round Trip Time

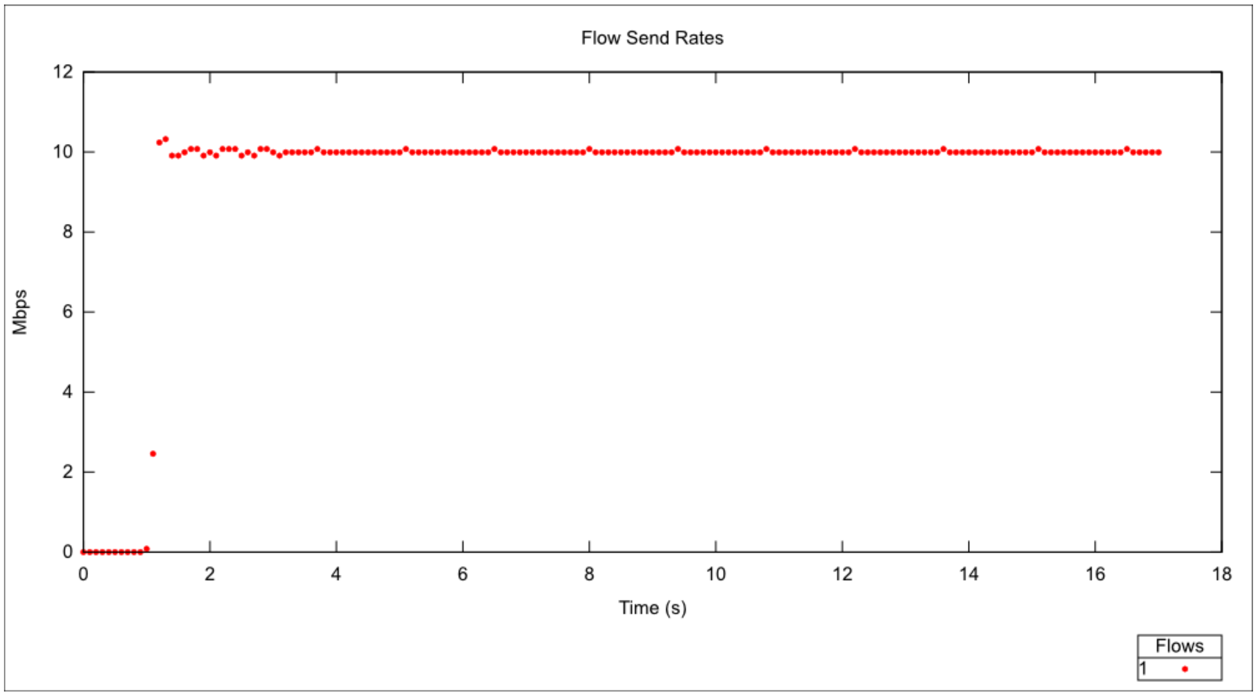

Figure 5: TCP VEGAS, Case 0: Flow Send Rates

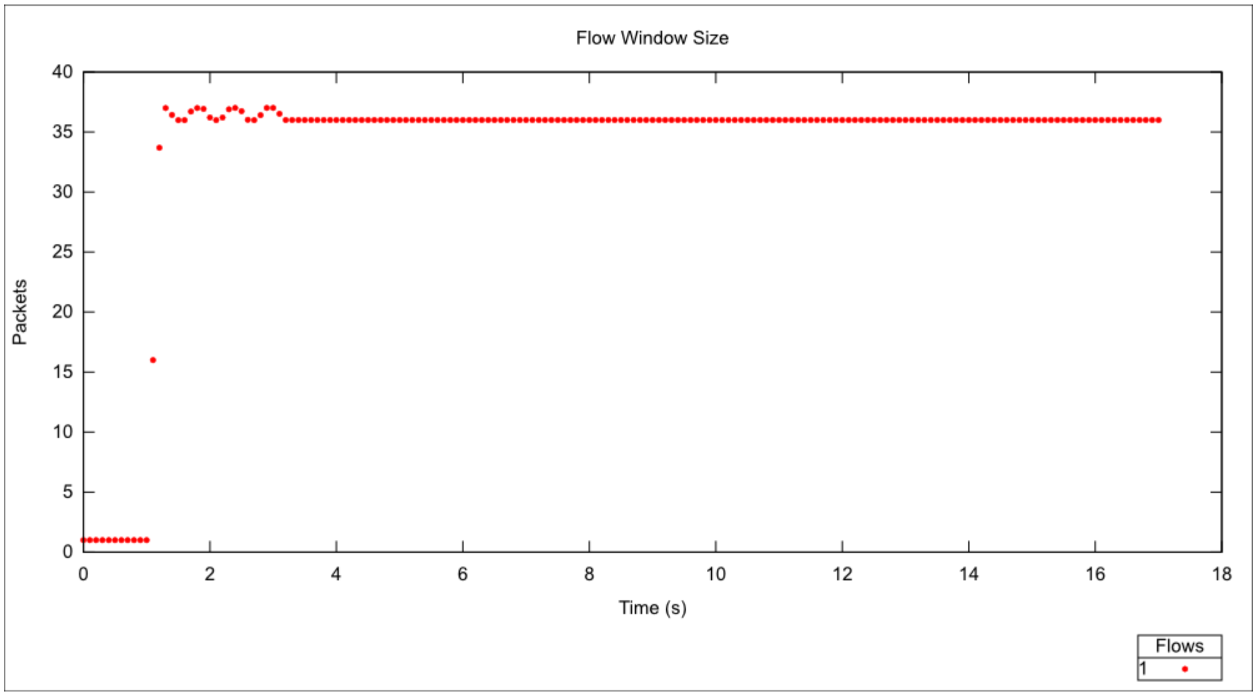

Figure 6: TCP VEGAS, Case 0: Flow Window

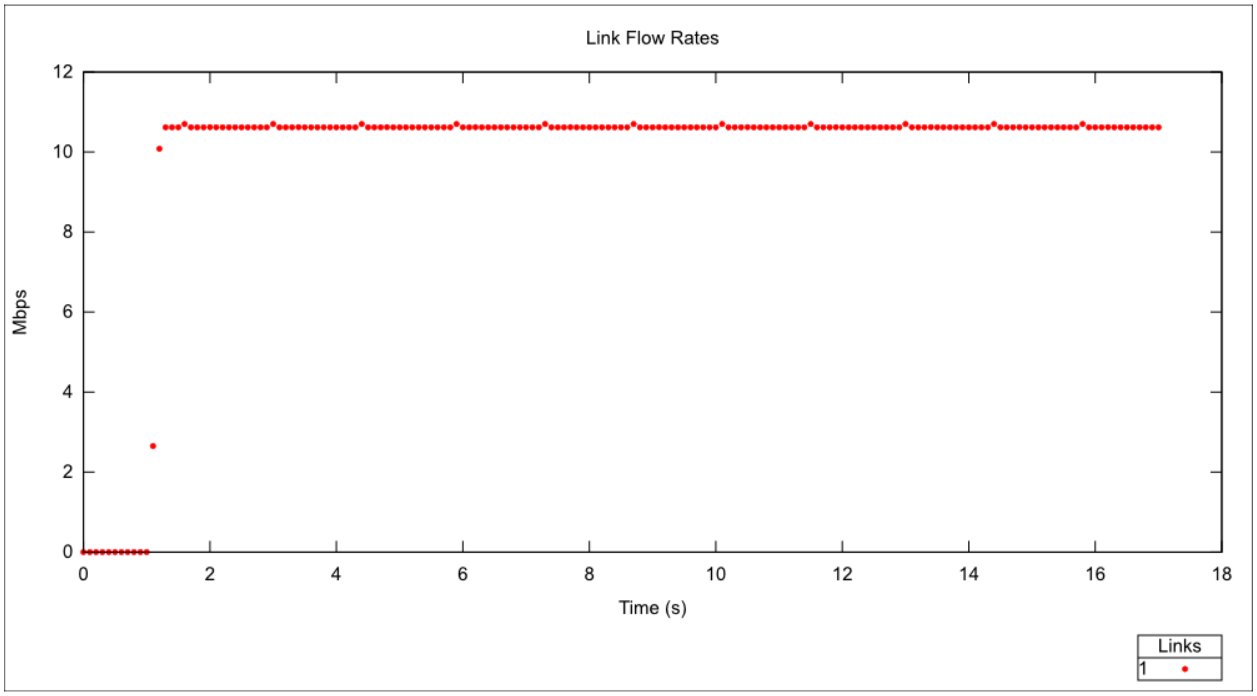

Figure 7: TCP VEGAS, Case 0: Host Send Rate

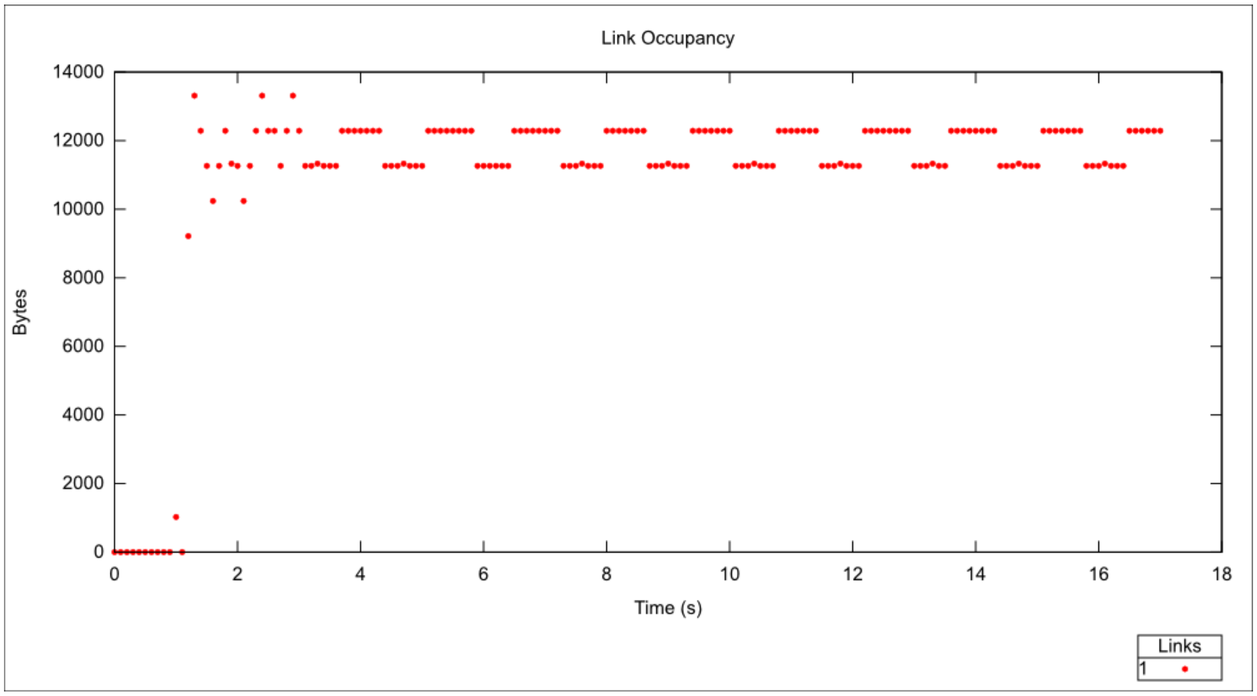

Figure 8: TCP VEGAS, Case 0: Host Send Rate

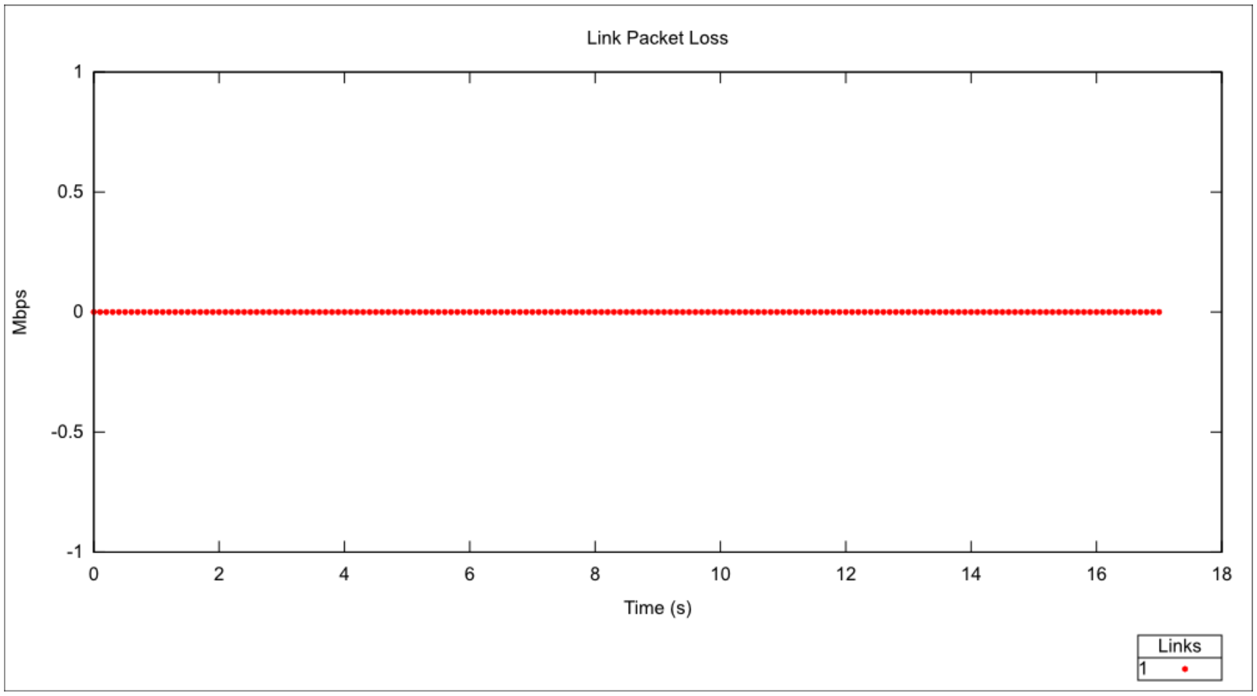

Figure 9: TCP VEGAS, Case 0: Host Send Rate

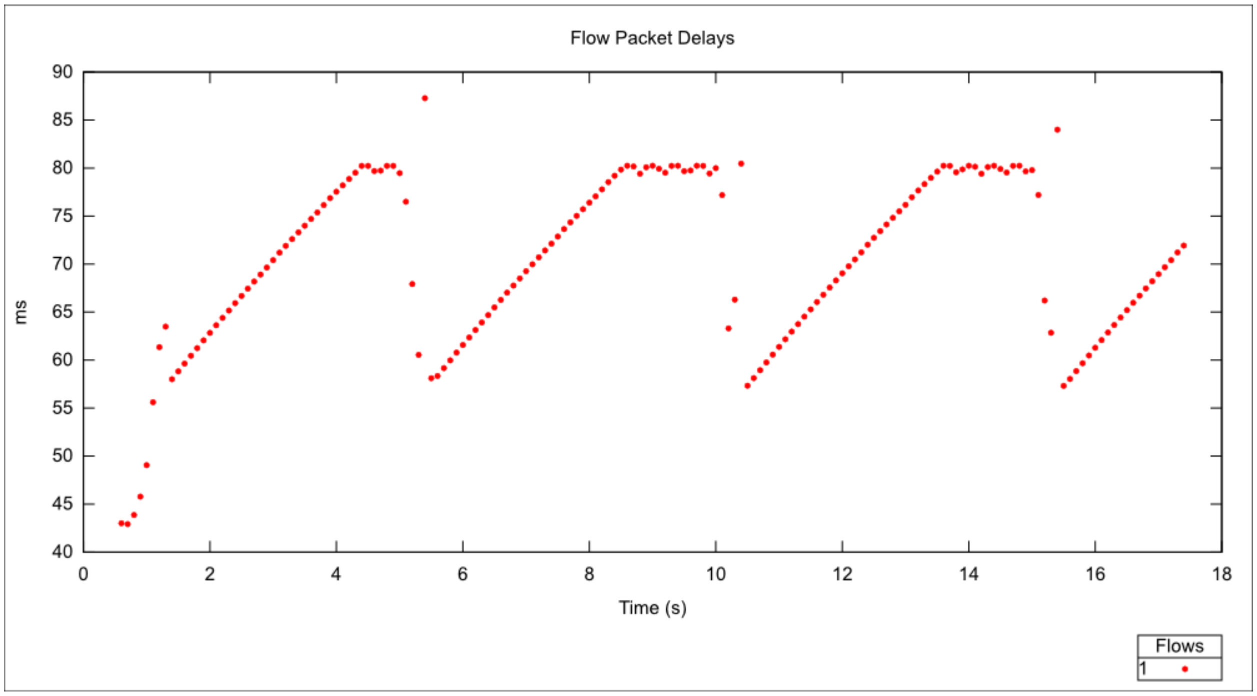

Figure 10: TCP VEGAS, Case 1: Flow Round Trip Time

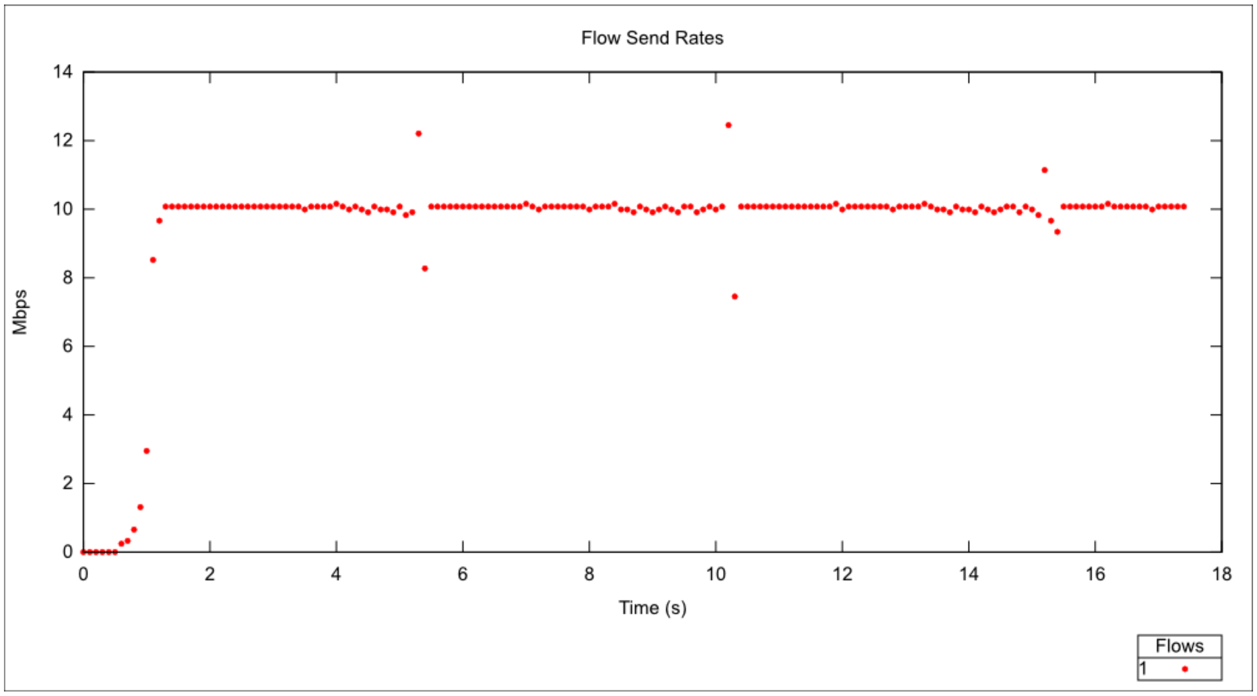

Figure 11: TCP VEGAS, Case 1: Flow Send Rates

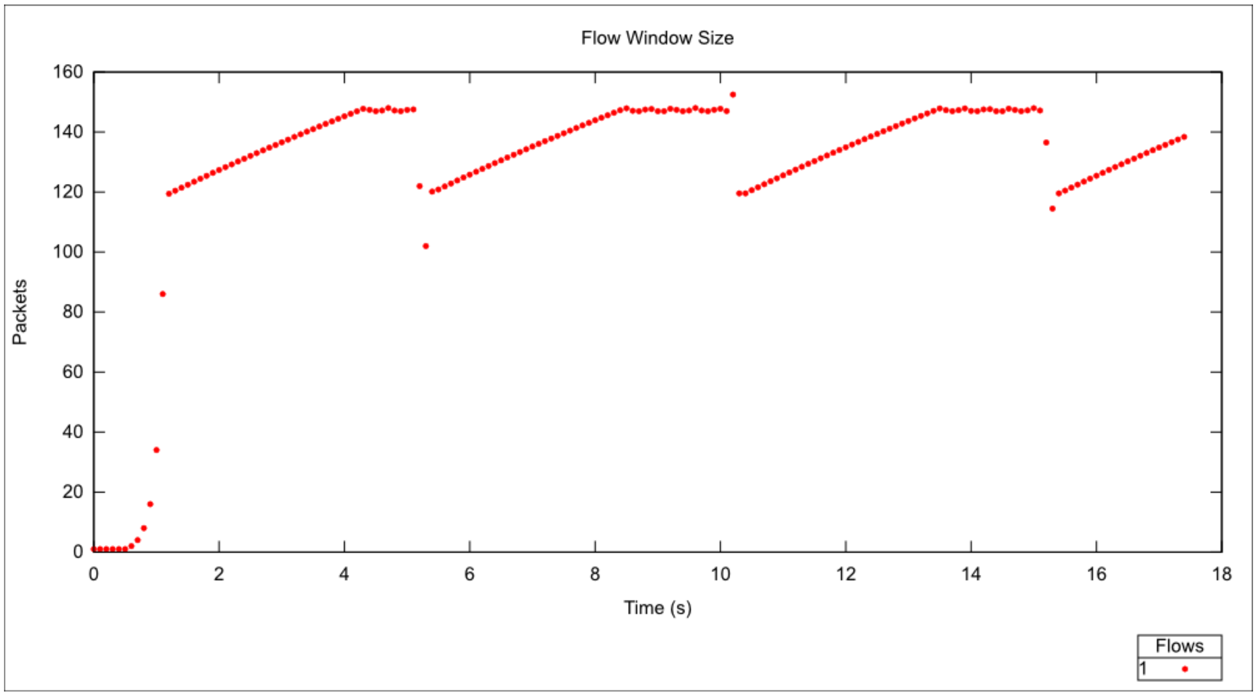

Figure 12: TCP VEGAS, Case 1: Flow Window

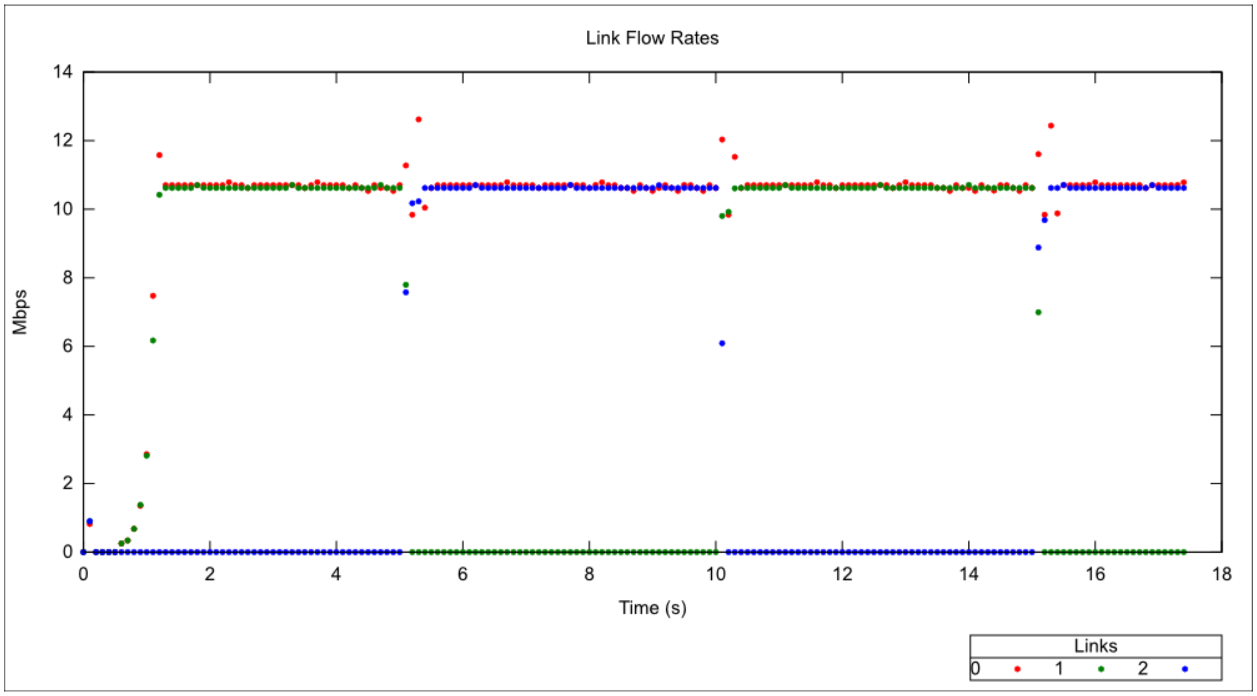

Figure 13: TCP VEGAS, Case 1: Host Send Rate

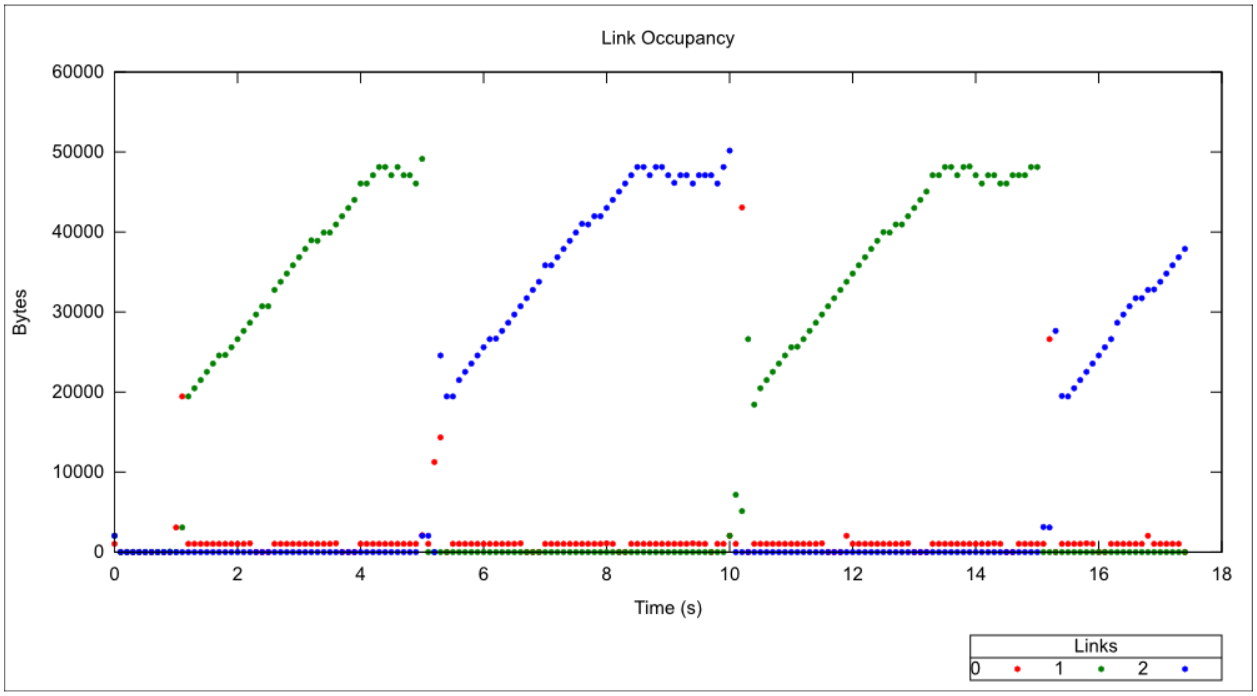

Figure 14: TCP VEGAS, Case 1: Host Send Rate

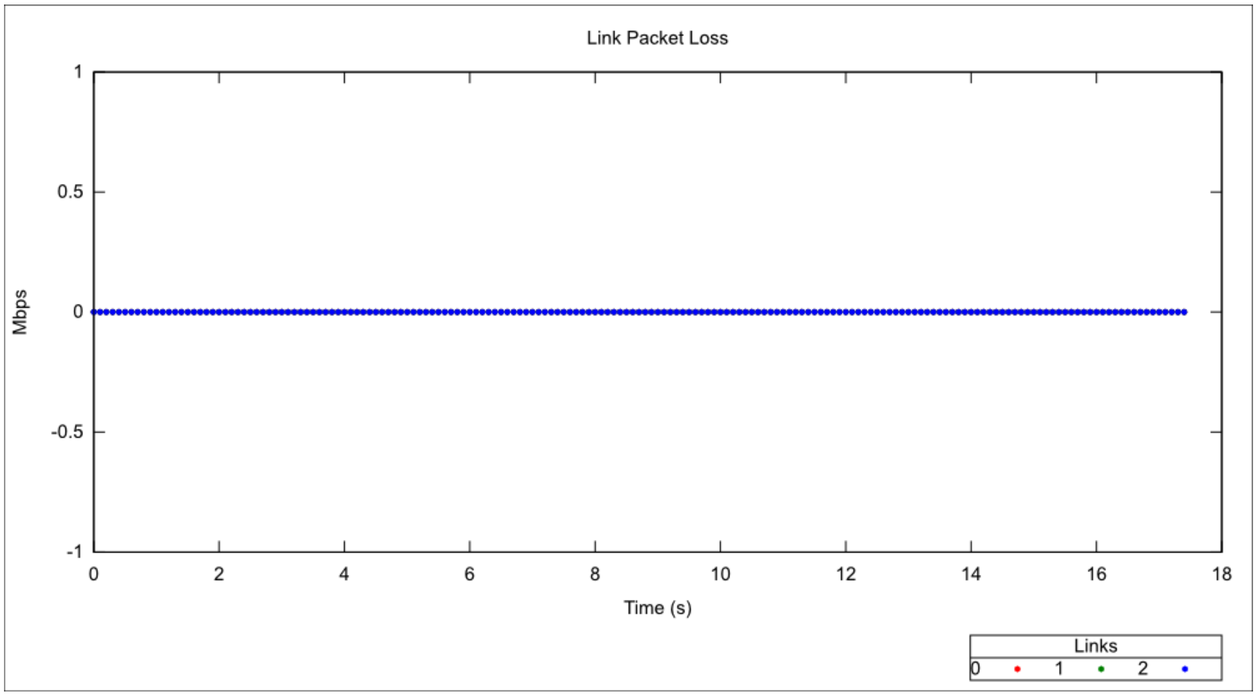

Figure 15: TCP VEGAS, Case 1: Host Send Rate

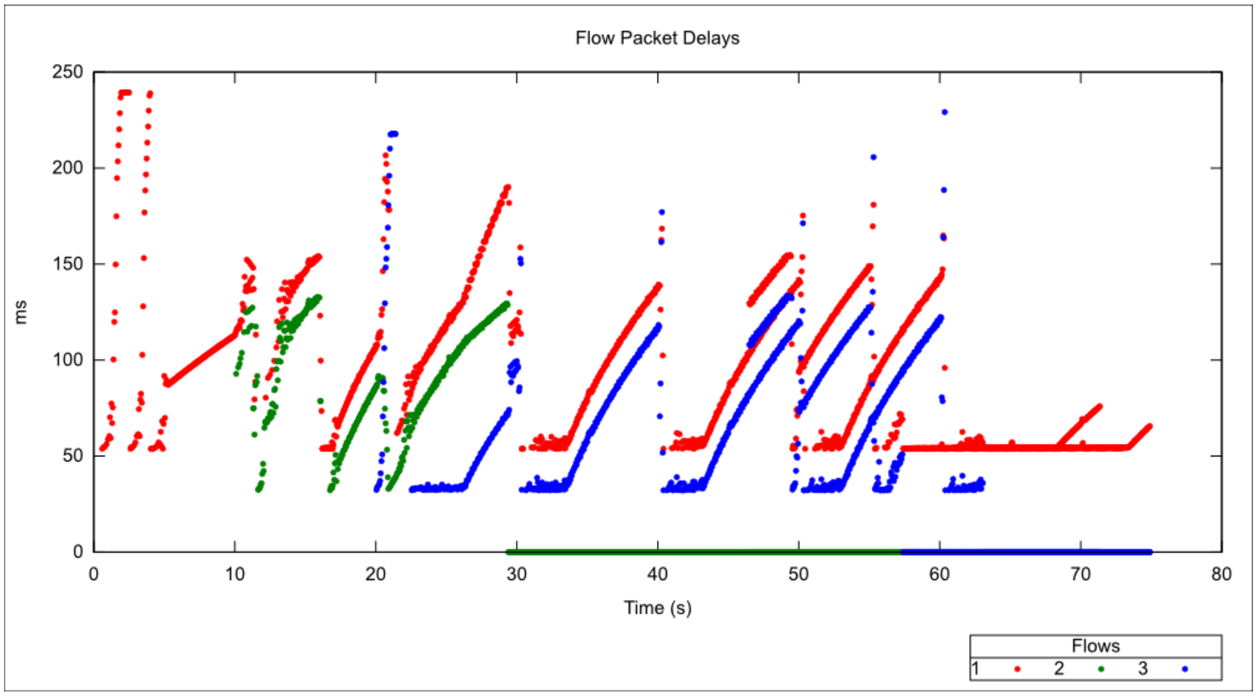

Figure 16: TCP VEGAS, Case 2: Flow Round Trip Time

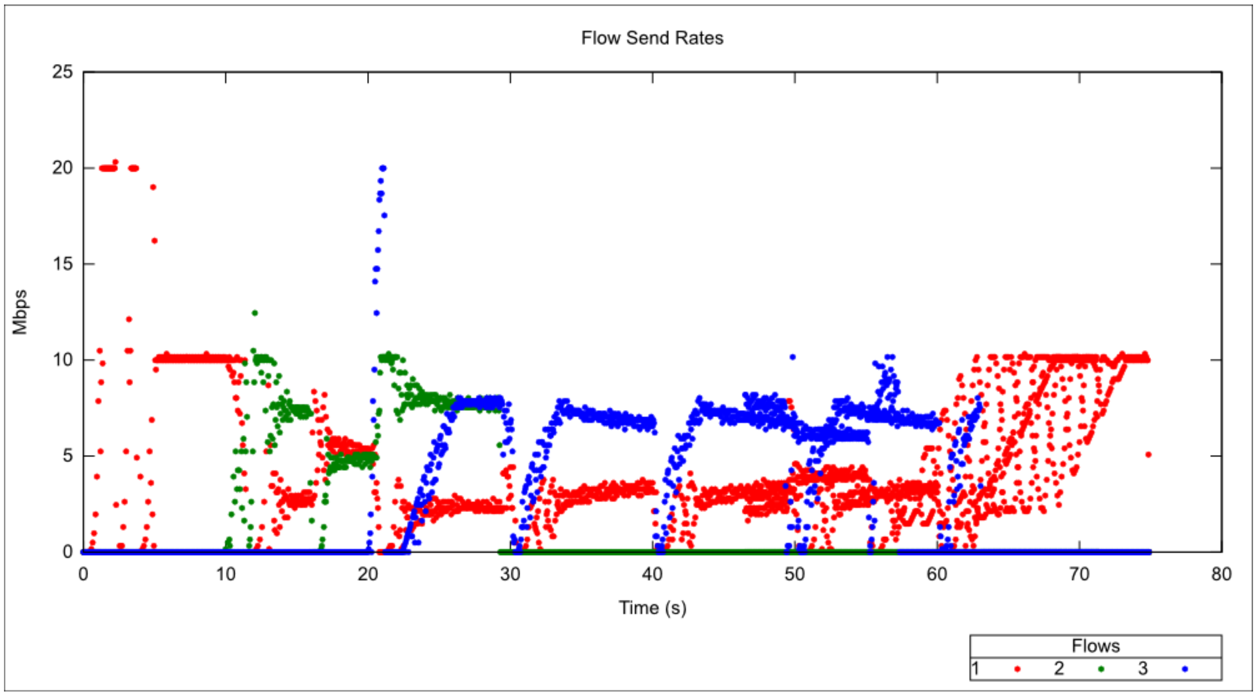

Figure 17: TCP VEGAS, Case 2: Flow Send Rates

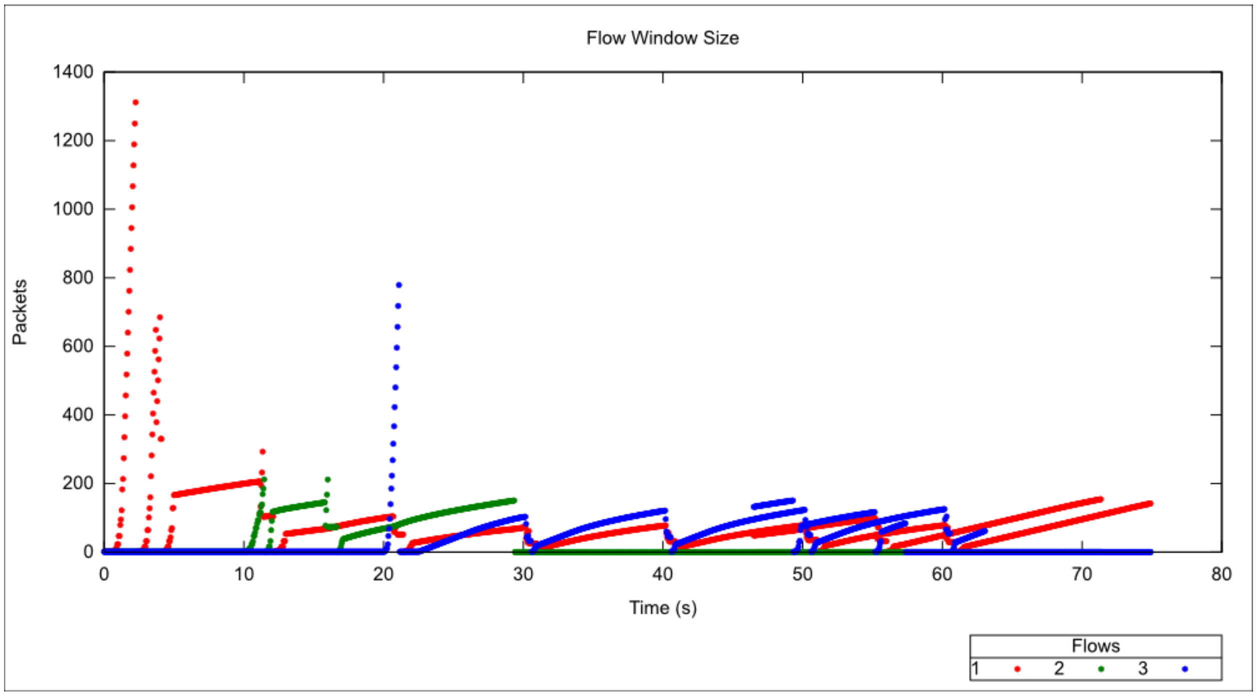

Figure 18: TCP VEGAS, Case 2: Flow Window

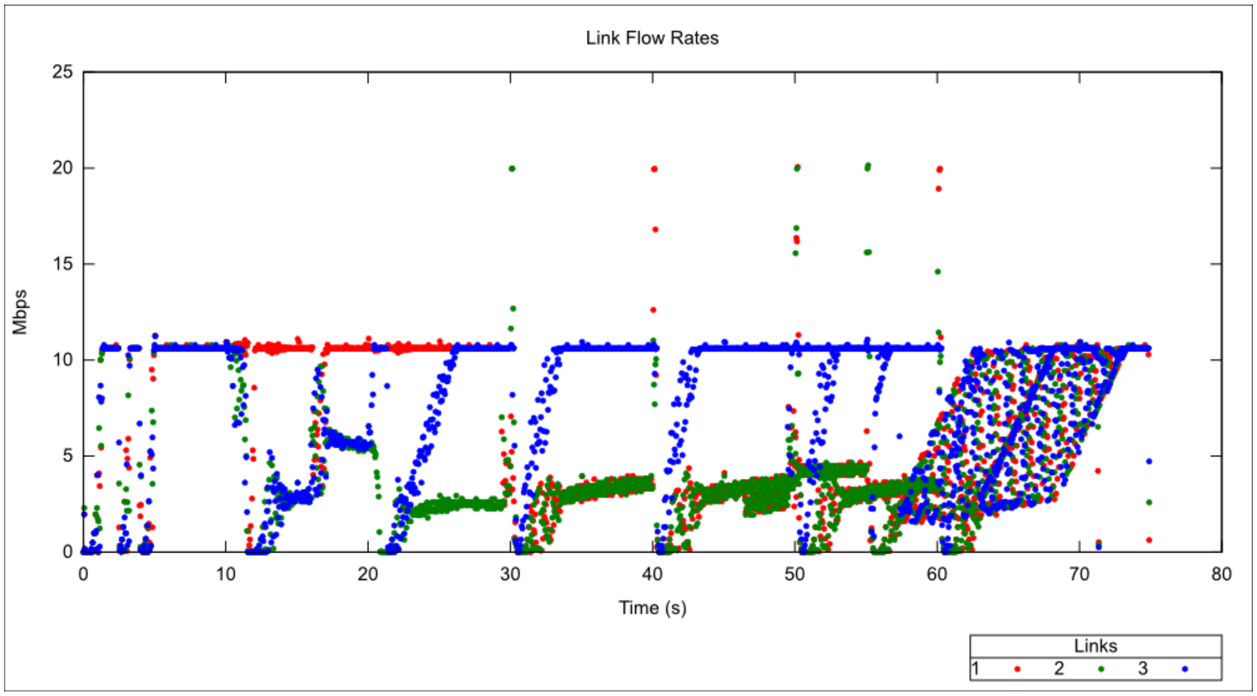

Figure 19: TCP VEGAS, Case 2: Host Send Rate

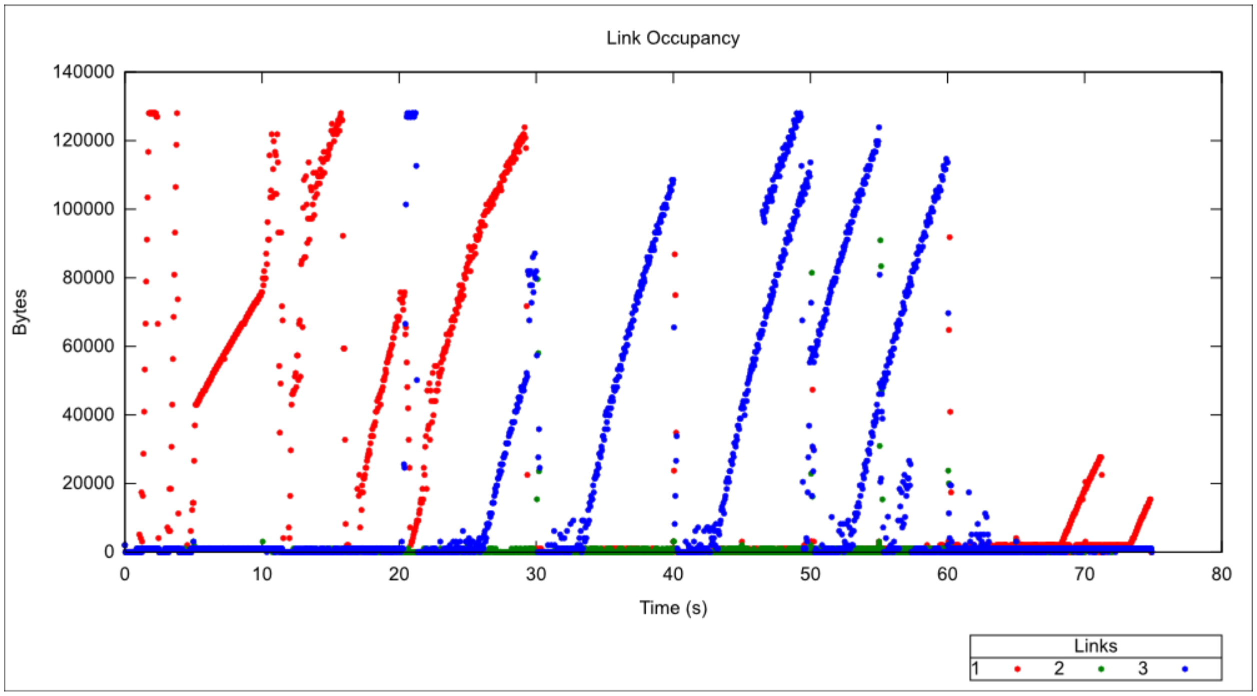

Figure 20: TCP VEGAS, Case 2: Host Send Rate

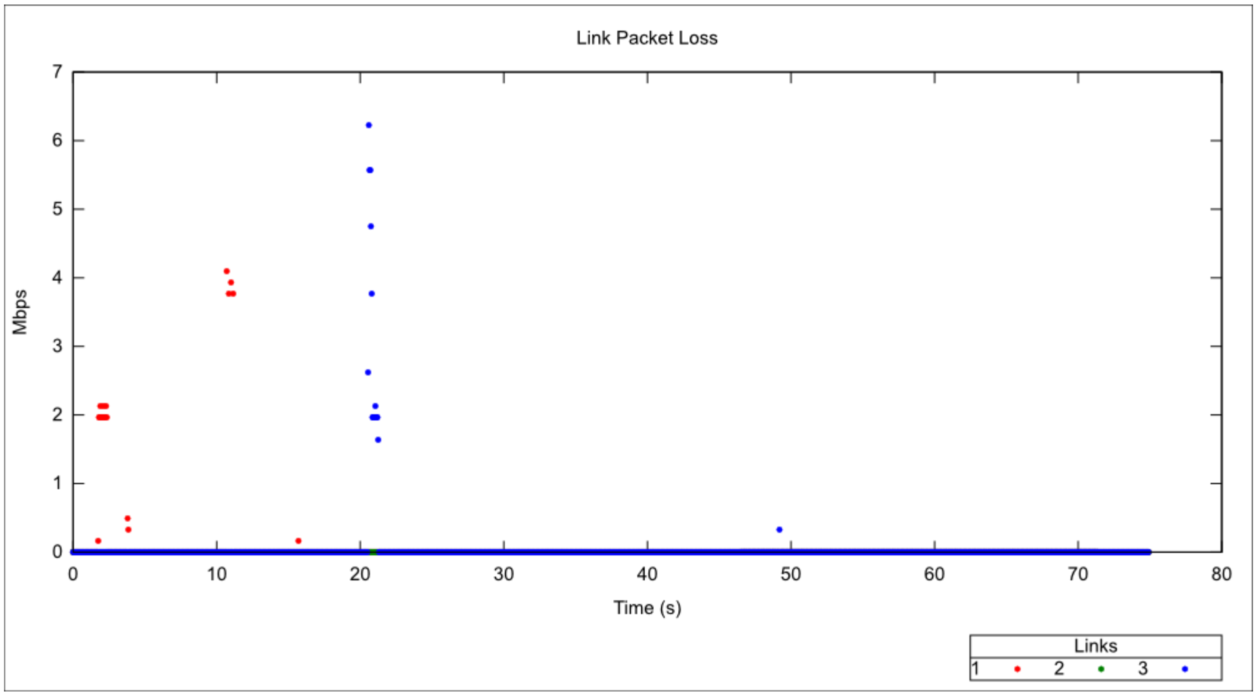

Figure 21: TCP VEGAS, Case 2: Host Send Rate

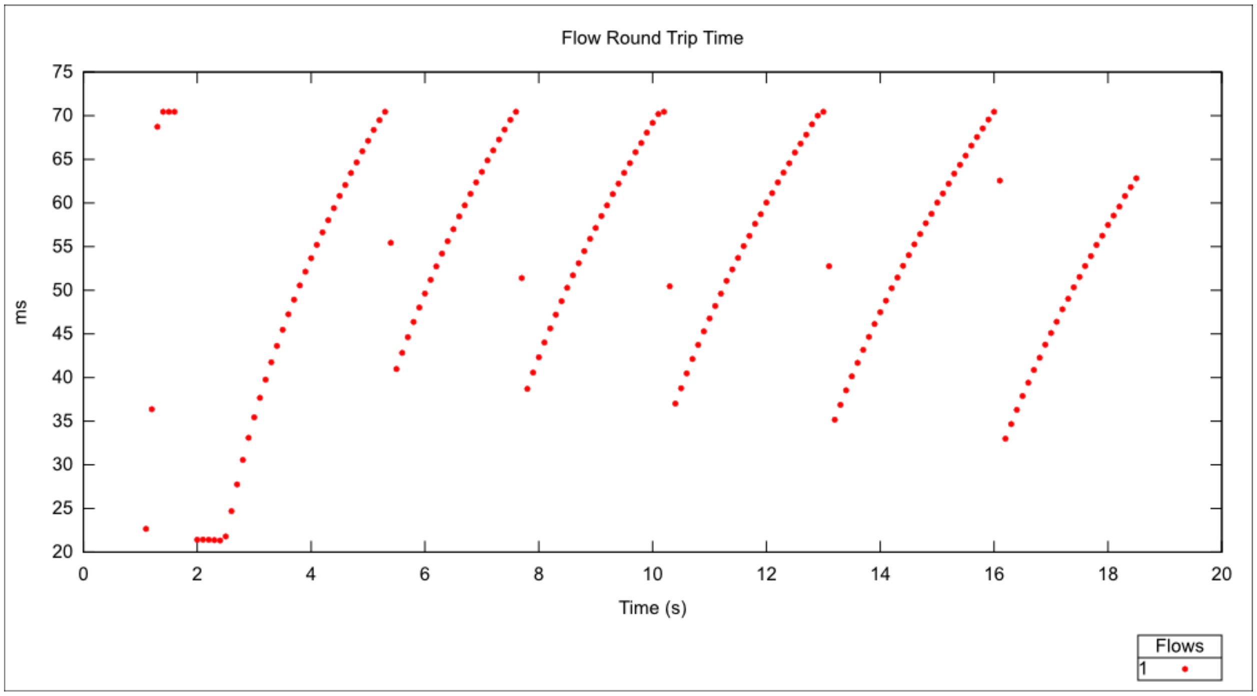

Figure 22: TCP RENO, Case 0: Flow Round Trip Time

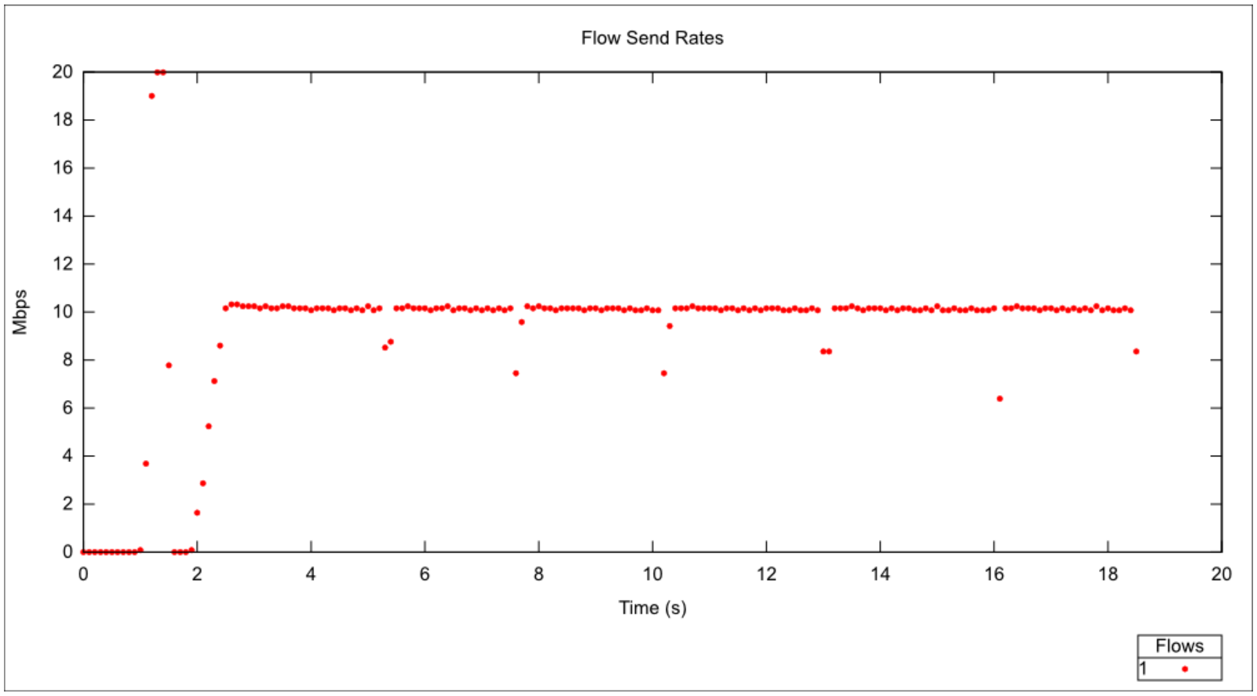

Figure 23: TCP RENO, Case 0: Flow Send Rates

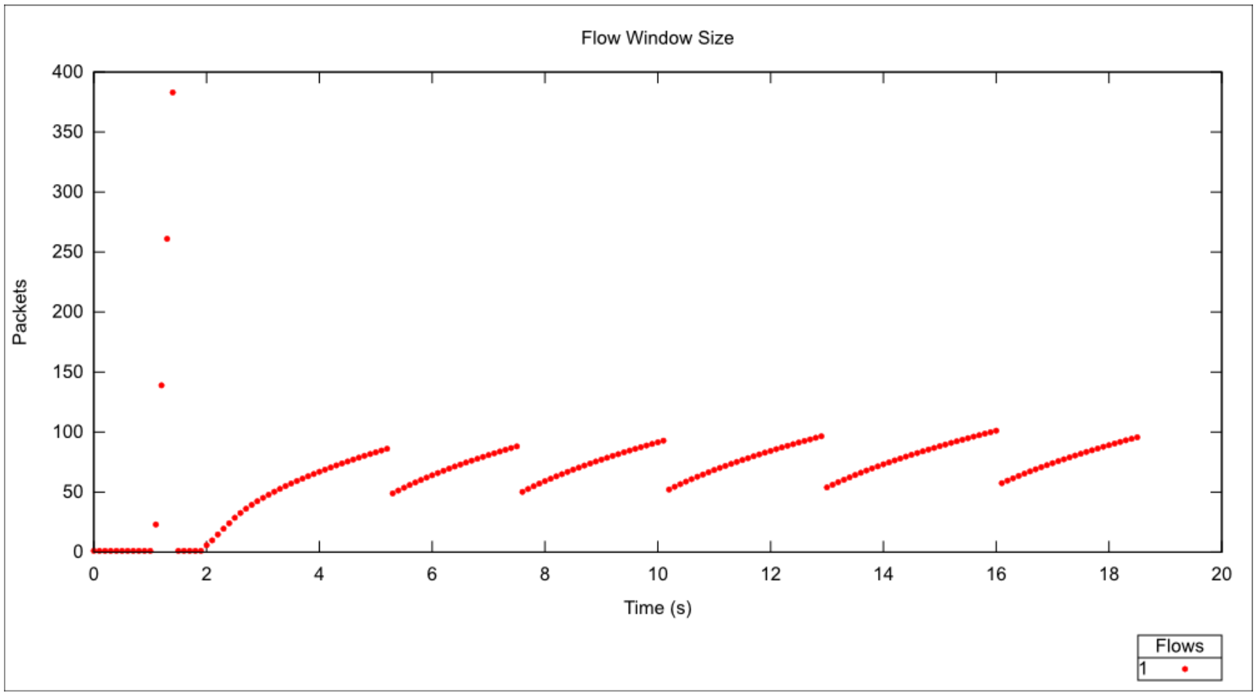

Figure 24: TCP RENO, Case 0: Flow Window

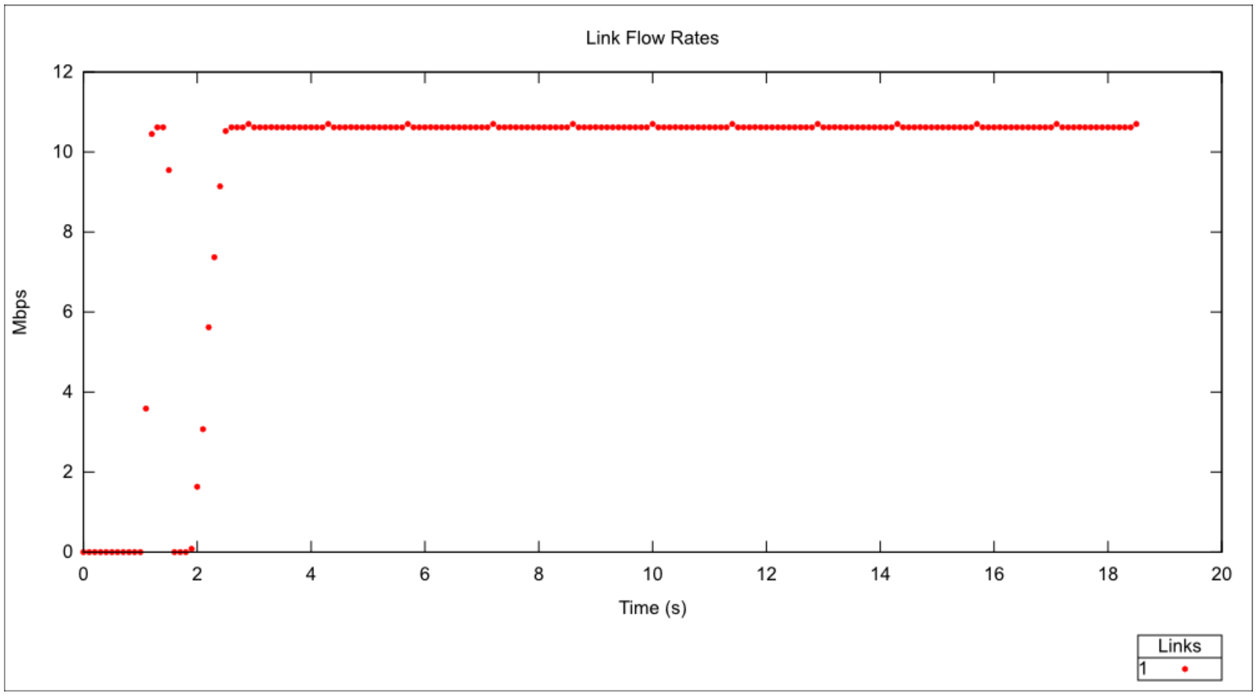

Figure 25: TCP RENO, Case 0: Host Send Rate

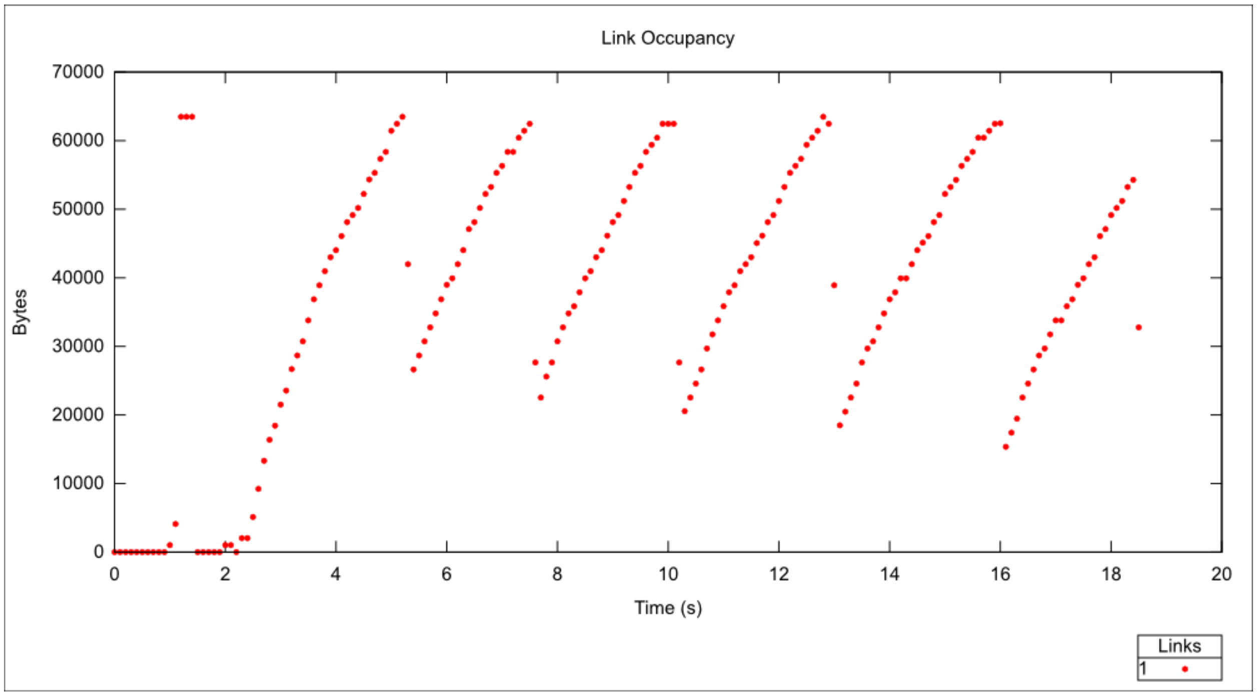

Figure 26: TCP RENO, Case 0: Host Send Rate

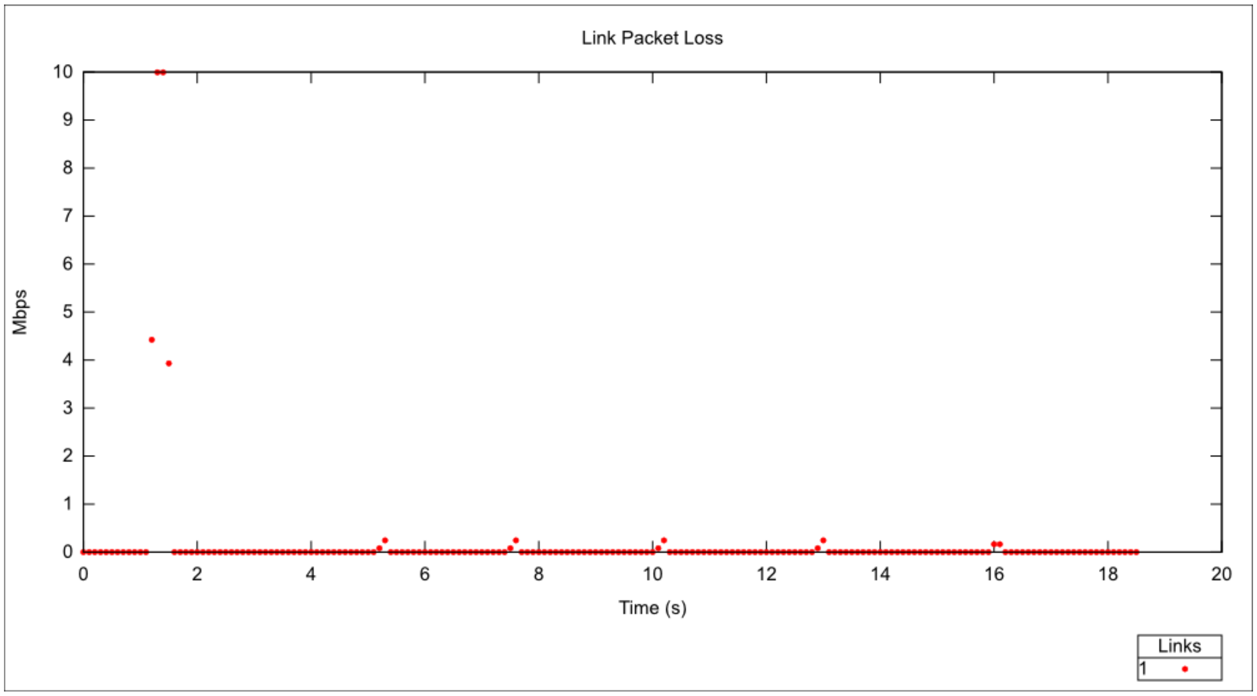

Figure 27: TCP RENO, Case 0: Host Send Rate

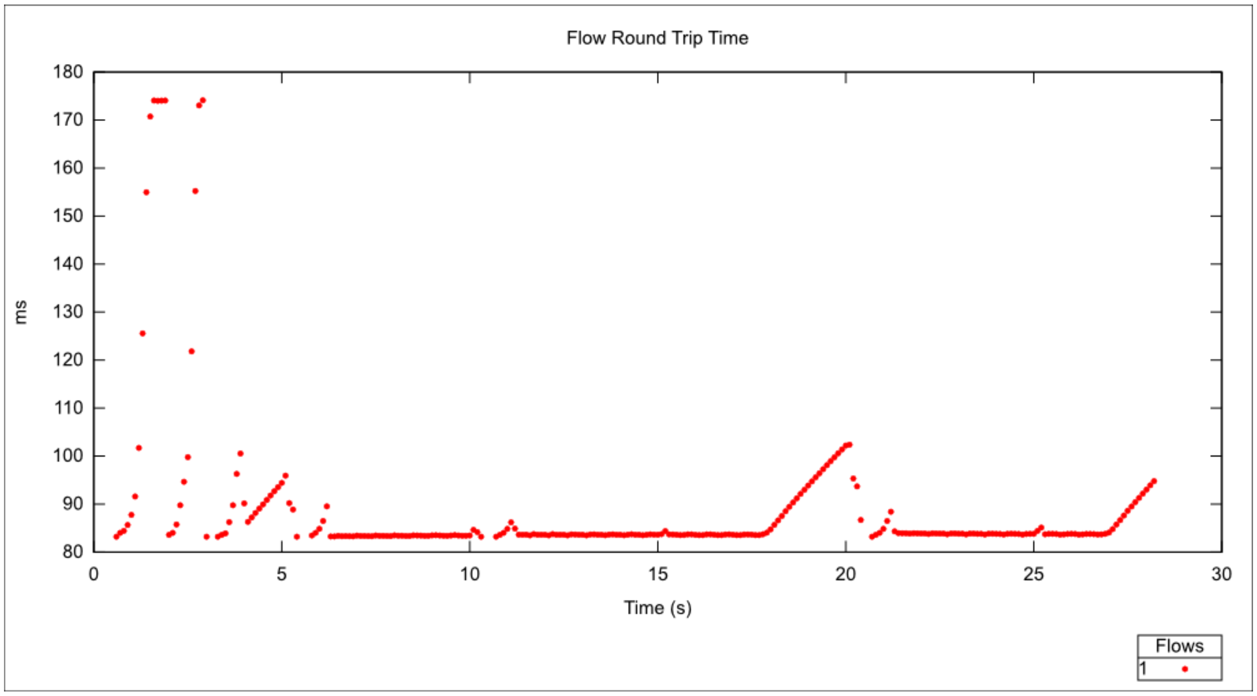

Figure 28: TCP RENO, Case 1: Flow Round Trip Time

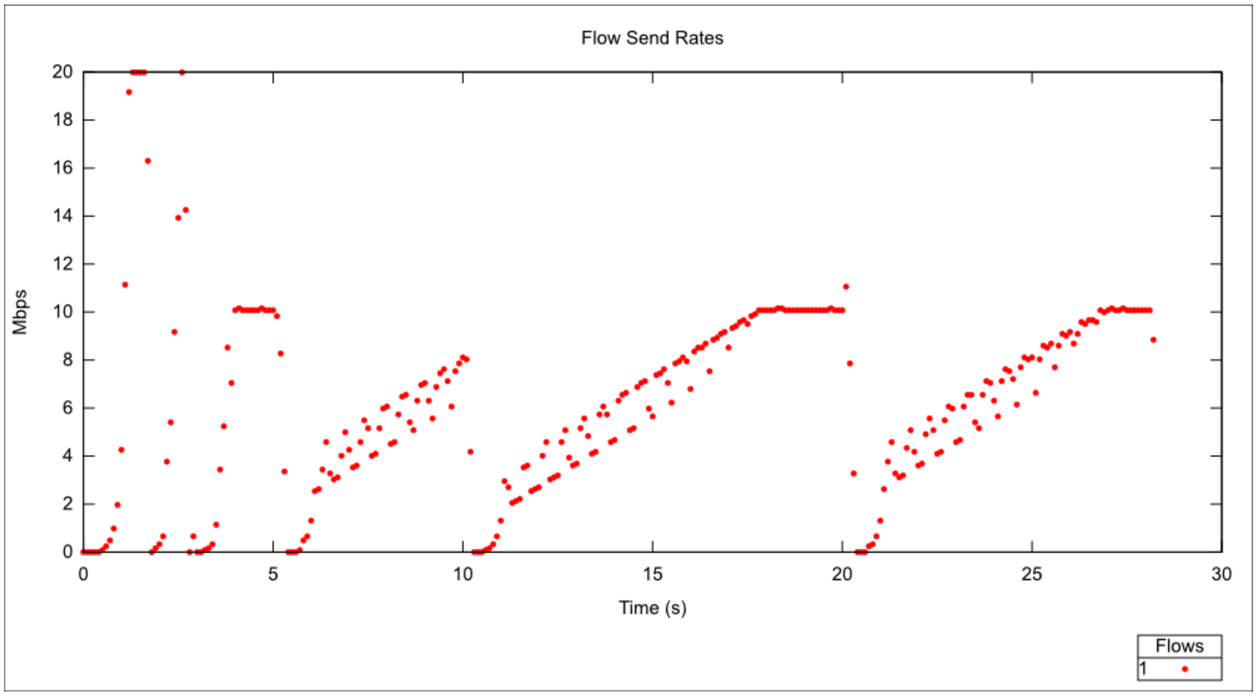

Figure 29: TCP RENO, Case 1: Flow Send Rates

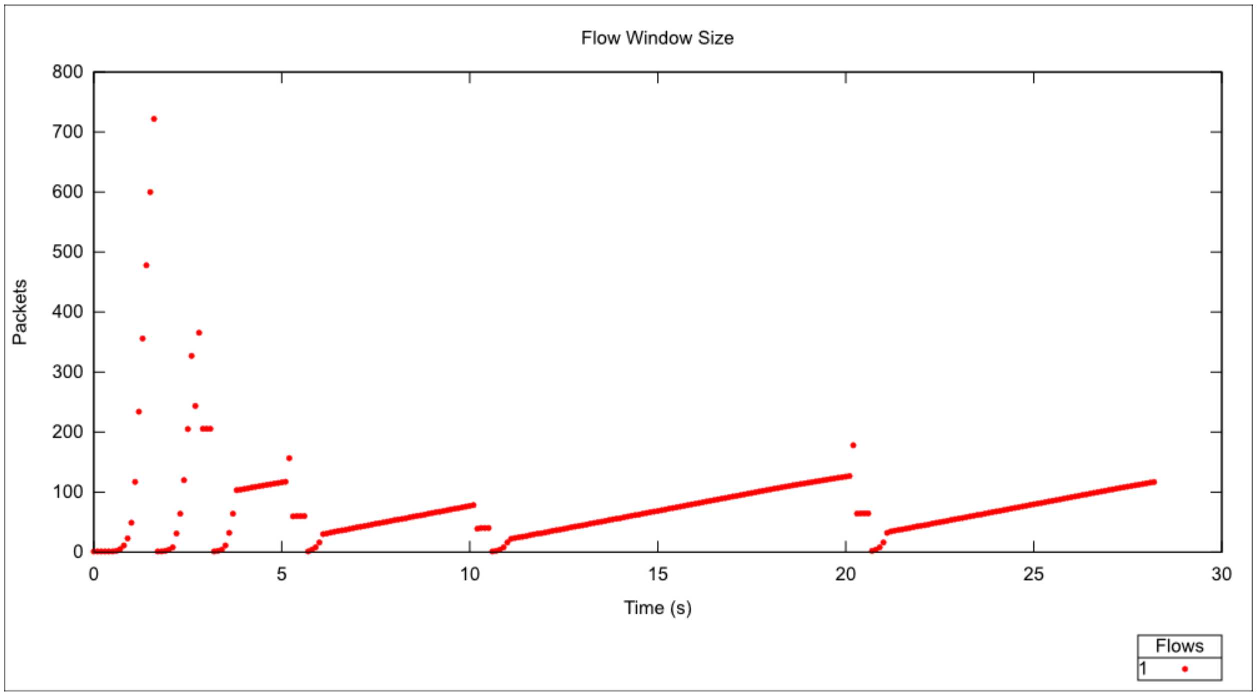

Figure 30: TCP RENO, Case 1: Flow Window

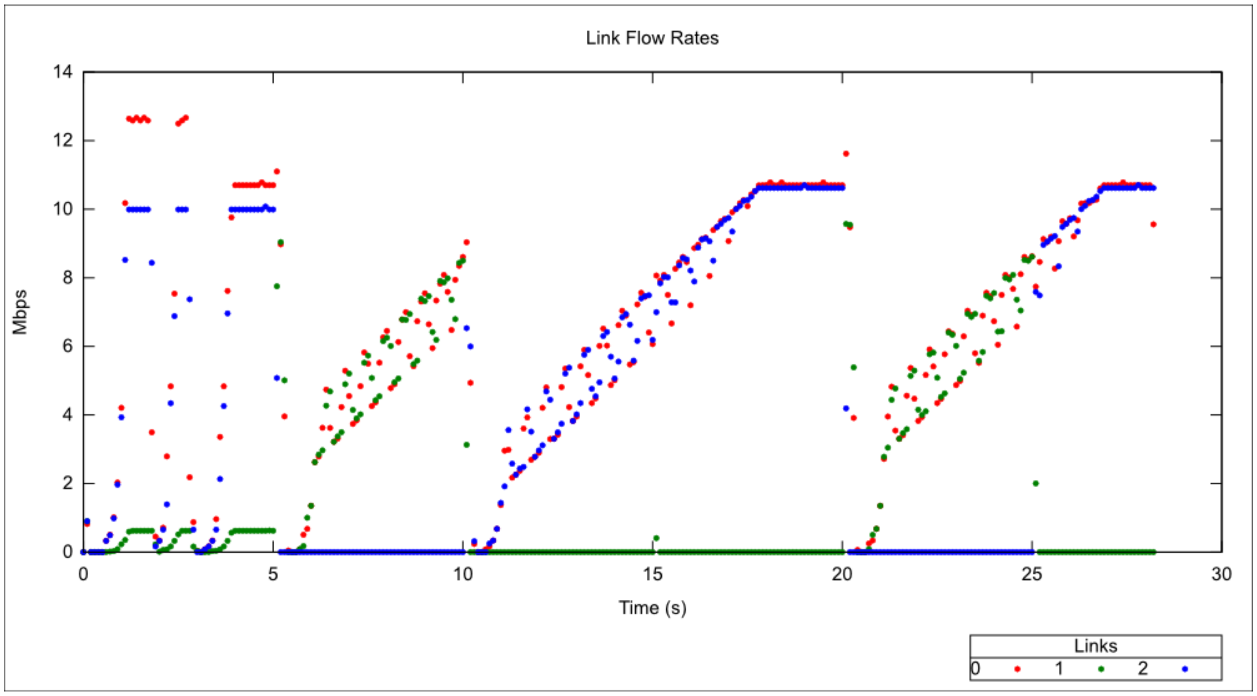

Figure 31: TCP RENO, Case 1: Host Send Rate

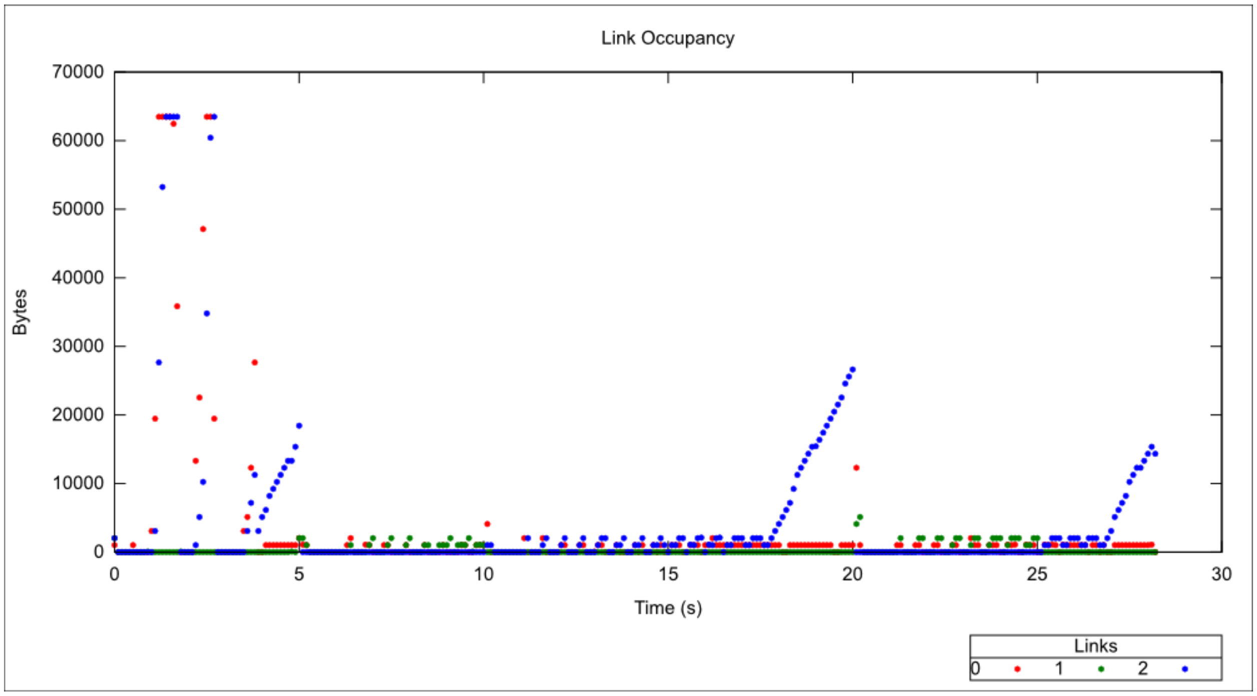

Figure 32: TCP RENO, Case 1: Host Send Rate

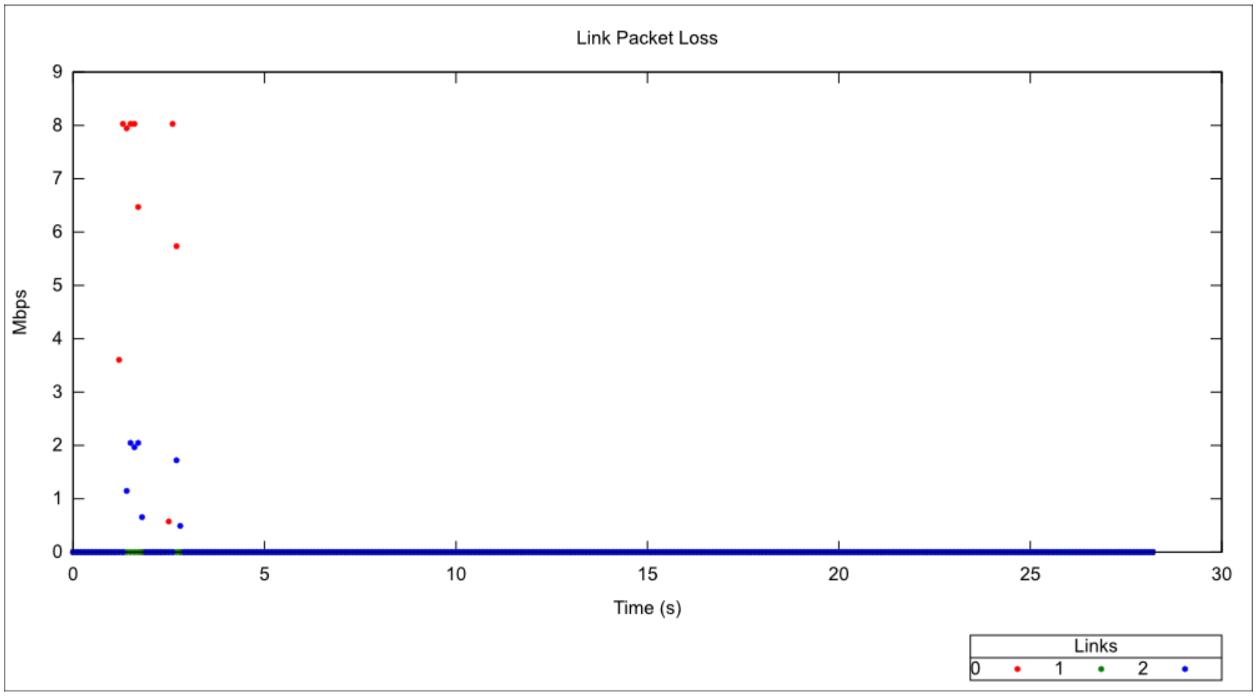

Figure 33: TCP RENO, Case 1: Host Send Rate

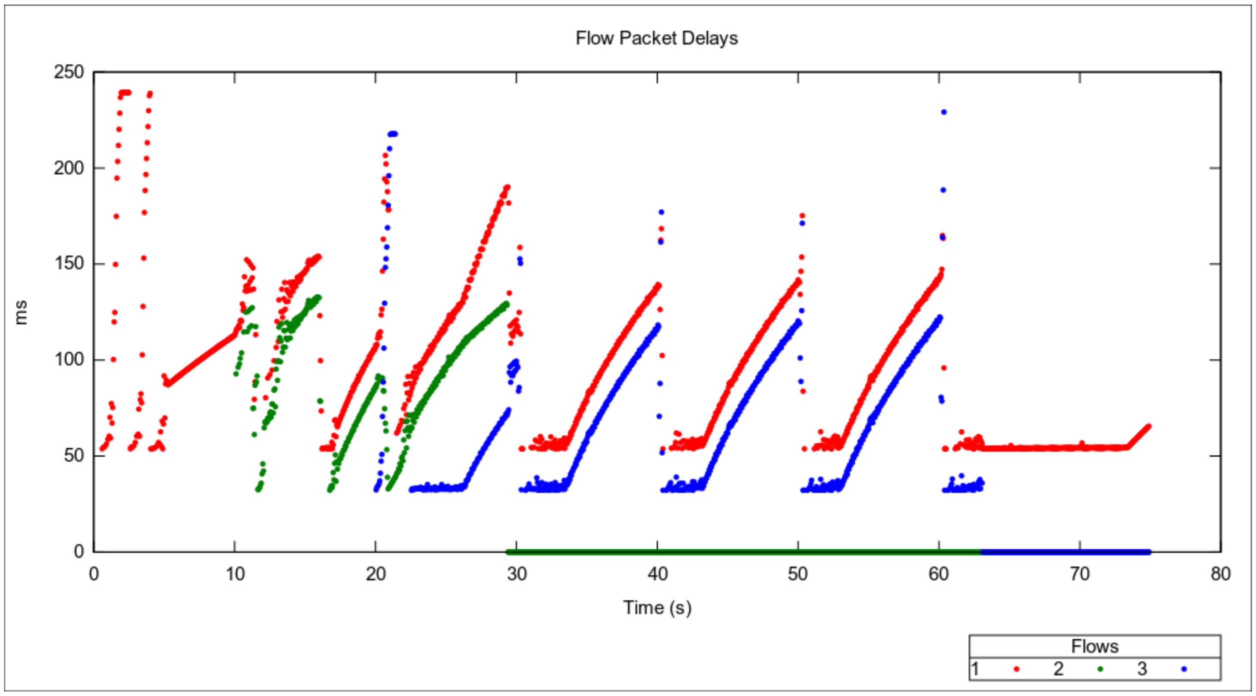

Figure 34: TCP RENO, Case 2: Flow Round Trip Time

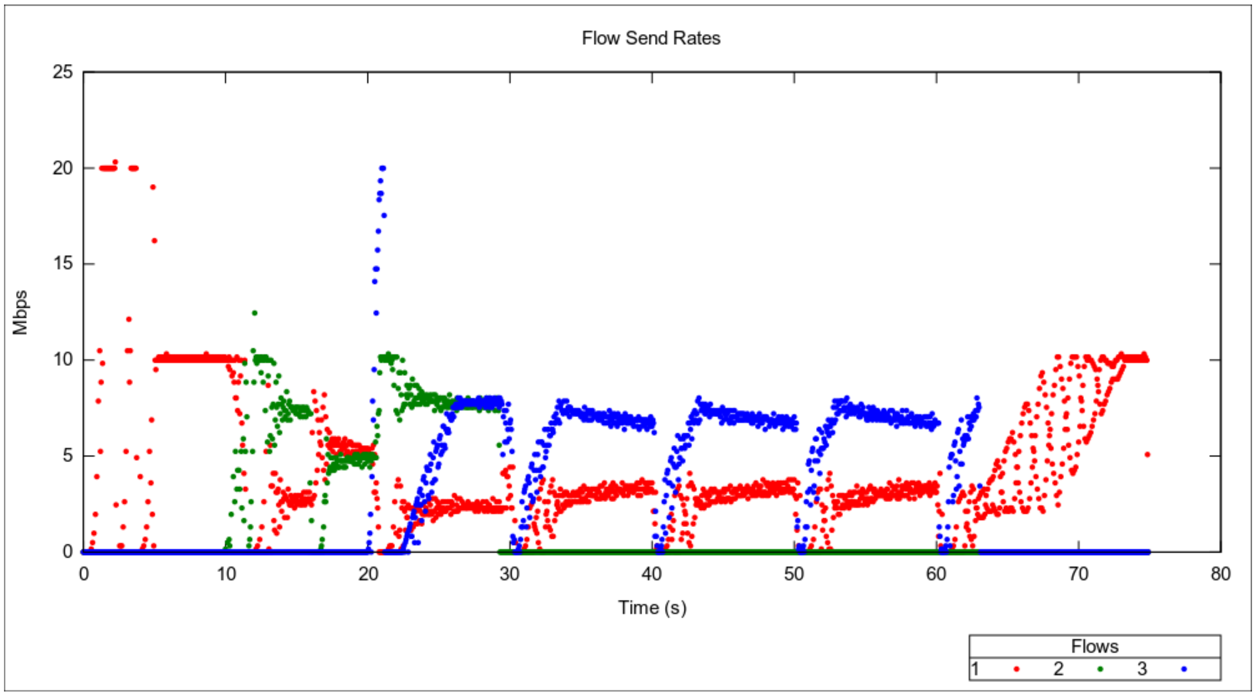

Figure 35: TCP RENO, Case 2: Flow Send Rates

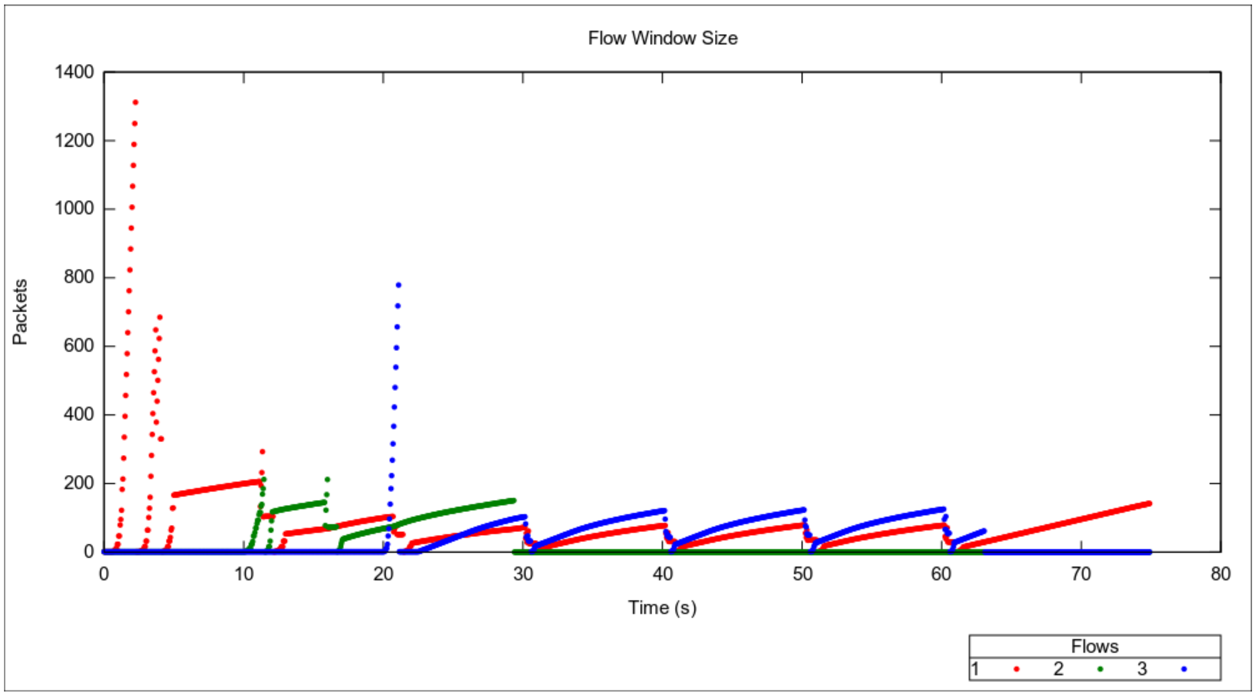

Figure 36: TCP RENO, Case 2: Flow Window

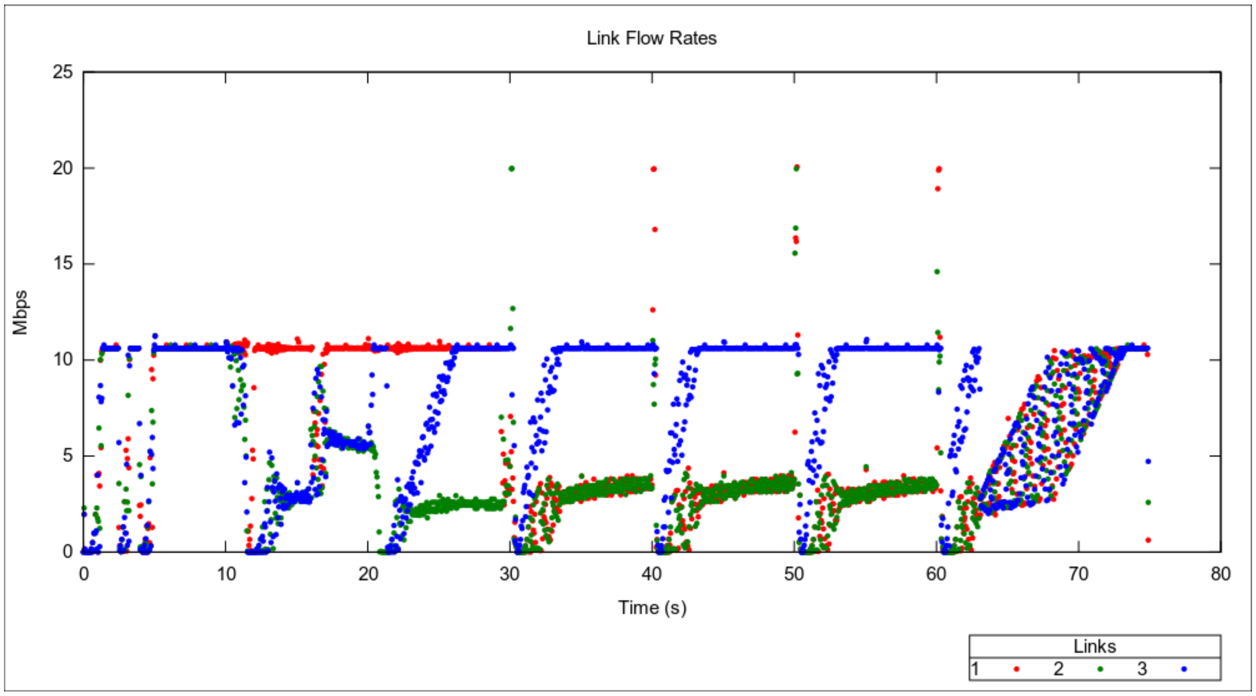

Figure 37: TCP RENO, Case 2: Host Send Rate

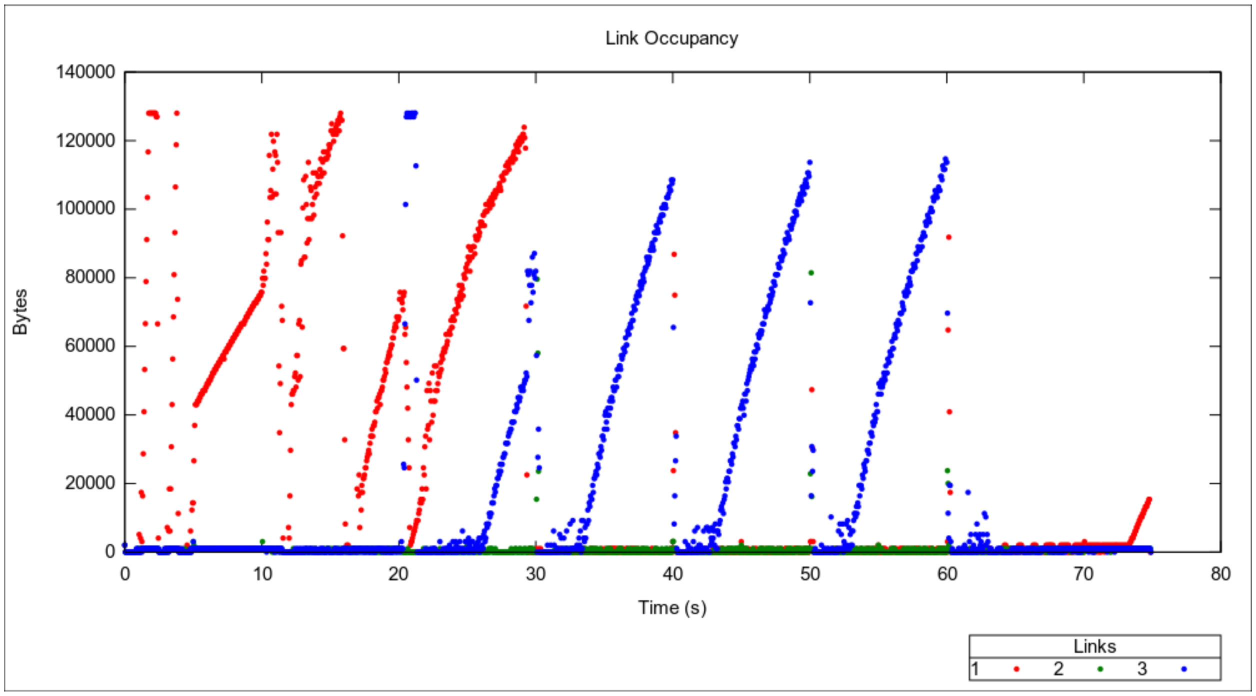

Figure 38: TCP RENO, Case 2: Host Send Rate

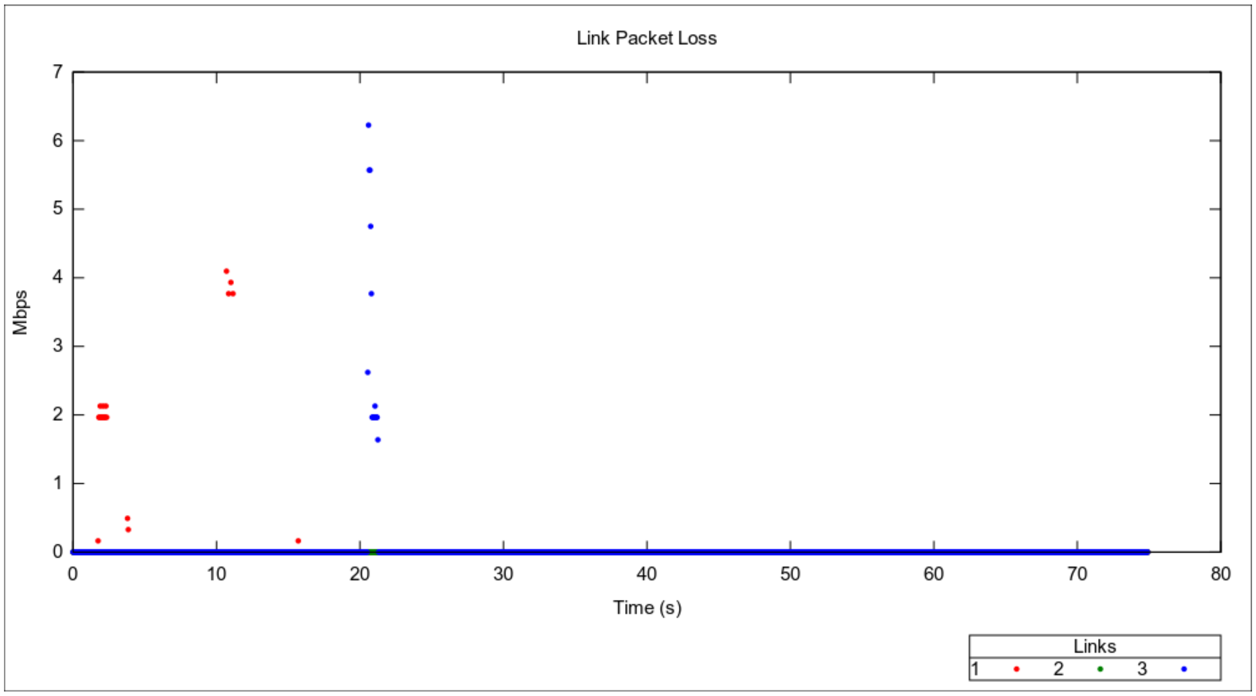

Figure 39: TCP RENO, Case 2: Host Send Rate