**CS 143 Simulation Final Report**

**The Greatest Network Simulation of All Time**

Hongjian Lan, Yamei Ou, Samuel Richerd

Jan Van Bruggen, Junlin Zhang

12/10/2014

**0. Division of Labor**

Hongjian: Host, Router Tests

Yamei: Router, Router Algorithms, Packets

Sam: Inputs, Controller, Link

Jan: Controller, Outputs, Repository, Manager,

Junlin: Flow, Transport Layer Algorithm, Flow Experimentation, Test Cases

**1. Running the Simulation**

The simulation runs on Python 2.7 with SimPy 3.0.5 and matplotlib 1.4.2 required.

The simulation is run by executing run.py, optionally with “case” or “duration” options specified.

The options are specified with “-c”/”--case” or “-d”/”--duration”, each followed by a numeric value.

The case value is the case number of the desired input file in cs143sim/cases/ (defaulted to 0).

The duration value is the simulation duration in seconds (defaulted to 10).

Example: “C:\cs143sim> C:\Python27\python.exe run.py -c 12 --duration 34.56”

**2. Input Parsing**

We chose to create a custom input file structure (rather than use a common format such as JSON) because we wanted to make the input file easily human readable and easily editable. This ease of use of the input greatly increases the usefulness of our network simulator since it allows a user to rather intuitively interact with the input file parser.

**Format:**

The input file itself is must be a text file, and will be parsed on a line by line basis. There can be only three types of lines:

* Empty lines: lines that contain only whitespace, or tabs, or absolutely nothing at all.
* Keyword lines: these are the most used in the file, they may include any amount of leading whitespace but the first word must correspond to a valid keyword. If the keyword describes an attribute (such as “RATE” or “SRC”) then the parser will look to the next word for the value for that attribute. Note that words are separated by at least one space.
* Comment lines: a comment line is included in the input file for readability to the user, a line that begins with “//” is ignored by the input parser. Note that inline comments are not recommended.

**Attribute Keywords**:

There are two types of keywords: entity keywords, or attribute keywords. An entity keyword defines what type of entity is being described by the subsequent attribute keyword lines. Note that the input parser temporarily buffers the attributes it receives for a given entity until it reaches a second ID attribute, or an empty line, at which point it takes the stored attribute values and creates the original entity. If a second ID attribute is specified, the parser creates the first entity and begins recording attributes for a second one. Not all attributes need be redefined if they share the same value. Subsequent IDs of an entity will share all attributes until they are redefined. The attribute keywords are:

* ID - Defines the ID label for that entity, this is required for every entity type.
* IP - Defines the IP address for the entity.
* SRC - Source’s ID for this entity (applies to FLOW)
* DST - Destination’s ID for this entity (applies to FLOW)
* DATA - Specifies the amount of data for this entity (FLOW) in MB.
* START - specifies a starting time for the entity
* ALGORITHM - Specifies a TLA identifier (FLOW) as an integer.
* CONNECTS - Specifies two ID’s that are connected by this Link.
* RATE - Specifies the rate of transmission over the LINK in Mbps.
* DELAY - Specifies the propagation delay due to that LINK in ms.
* BUFFER - Specifies the size of the LINK buffer (in KB).

**Entity Keywords:**

* HOST - This describes a host, a host requires an ID, and IP attribute.
* FLOW - Flows require ID, SRC, DST, DATA, START and ALGORITHM attributes.
* LINK - Links require ID, CONNECTS, DELAY, RATE attributes
* ROUTER - Router requires ID and IP address attributes.

**Example:**

// This is an example file:

HOST

ID H1

IP 192.168.1.1

ID H2

IP 192.168.1.2

ID H3

IP 192.168.1.3

ROUTER

ID R1

IP 192.168.1.4

LINK

ID L1

RATE 10

DELAY 10

BUFFER 64

CONNECTS H1 R1

ID L2

CONNECTS H2 R1

ID L3

CONNECTS H3 R1

// the above block allows re-use of the DELAY, RATE, and BUFFER

// attributes for multiple LINKs

FLOW

ID F1

SRC H1

DST H2

DATA 20

START 1.0

FLOW

ID F2

SRC H1

DST H3

DATA 24

START 1.3

**3. Controller**

**Class**:

* Actor dictionaries (Dictionaries of actors keyed by their names)
* Record dictionaries (Dictionaries of output recording lists keyed by their names)

**Controller parses an input case file, creates actors, and runs a ControlledEnvironment**

make\_actor()

* Creates an actor in the ControlledEnvironment
* Adds the actor to the corresponding actor dictionary

record\_output\_variable( )

* Called by actors (actors reference Controller through the ControlledEnvironment)
* Records the values passed by actors

**4. Host**

**Class**:

* Address (Unique IP Address to identify hosts)
* Flows (List of flows on this host)
* Link (The link connected to this host)

**Host is mainly in charge of sending and receiving packets for flows on it.**

1. send( )

* This is the reaction after receiving a packet from a flow.
* Add the packet received to its link’s buffer.

2. react\_to\_packet\_receipt( )

* This is the reaction after receiving a packet from its link.
* First check if the destination of the packet received is the host itself. If not, discard the packet.
* Then check whether it’s a DataPacket or RouterPacket. Discard the packet if it’s a RouterPacket.
* Finally check whether it’s an AckPacket.

o Is a regular DataPacket: Hand the packet to the flow whose source and destination are the packet’s source and destination.

o Is an AckPacket: Hand the packet to the flow whose source and destination are the packet’s destination and source.

**5. Link**

We simulate a two-way link by two one-way links, one from the source of the two-way link to its destination, the other from the destination of the two-way link to its source.s

**Class**:

* Source & Destination (Link’s source & destination)
* Delay (The propagation delay of the link)
* Rate (Link’s link rate)
* Buffer (Link’s buffer)
* Busy (To indicate if the link is currently removing packet from the source)
* Utilization (To record the fraction of the link’s capacity in use)

**Link is in charge of sending packets from its source to its destination**.

1. add( )

* Check if the link is busy now.

o If it is, then call self.buffer.add( ) to add the packet to its buffer.

o If not, then call self.send( ) to send the packet.

2. react\_to\_link\_available( )

* Get packet from its buffer if it’s not empty.
* Call send( ).

3. send( )

* Divide the packet’s size by link rate to get the transmission delay. Add transmission delay and propagation delay together to get the total delay for the packet.
* Call PacketReceipt( ). The destination will receive the packet after the total delay.
* Call LinkAvailable( ). The link will be available again after the transmission delay.

**6. Buffer**

**Class:**

* Link (The link this buffer belongs to)
* Packets (Packets in the buffer)
* Capacity (The capacity of the buffer)
* Current\_level (The current occupancy of the buffer)

**Buffer is in charge of storing packets waiting to be sent by the link.**

1. get( )

* Hand the first packet in the buffer to the link to transmit and update the buffer’s current occupancy.

2. add( )

* Check if the sum of its current occupancy and the size of the packet exceeds its capacity

o If so, then drop the packet.

o If not, put the packet in the buffer, and update the buffer’s current occupancy.

**7. Flow**

**Class:**

* Source & Destination (Flow’s source & destination)
* Amount (The amount of data the flow is going to transmit)
* Tla (The congestion control algorithm this flow is using)
* Rcv\_expect\_to\_receive (The packet the flow expects to receive)
* Rcv\_received\_packets (The list of packets the flow has received)

**Flow is in charge of congestion control.**

1. make\_ack\_packet( )

* Use go back N.
* Check if the number of the packet received is the same with rcv\_expect\_to\_receive.

o If so, update the rcv\_expect\_to\_receive, add the packet to rcv\_received\_packets, and make AckPacket with updated parameter.

o If not, just make AckPacket with the same rcv\_expect\_to\_receive.

2. react\_to\_packet\_receipt( )

* This is the reaction after receiving a packet from the host.
* Check whether the packet received is a AckPacket.

o If so, call self\_tla\_react\_to\_ack( ).

o If not, call self.make\_ack\_packet( ) to make AckPacket and call send\_packet( ) to send it.

3. time\_out( )

* When time out happens, call self.tla.react\_to\_time\_out( ).

4. react\_to\_flow\_start( )

* When it’s time for the flow to start, call self.tla.react\_to\_flow\_start( ).

**8. Transport Layer Algorithm**

**Brief description of classes**:

There are two classes of TLA: TCPTahoe and TCPVegas.

1. **TCPTahoe**

Class TCPTahoe implements TCP Tahoe, TCP Tahoe with fast retransmit, and TCP Reno. It implements Go-Back-N as the error control algorithm.

**Main Parameters:**

* enable\_fast\_retransmit and enable\_fast\_recovery:

parameter that determine what algorithm is running.

|  |  |  |
| --- | --- | --- |
| enable\_fast\_retransmit | enable\_fast\_recovery | algorithm |
| False | False | TCP Tahoe |
| True | False | TCP Tahoe  with fast retransmit |
| False | True | TCP Reno |

* rtt\_alpha, rtt\_beta:

rtt\_alpha determine the updating rate when estimating the average value of rtt.

rtt\_beta determine the updating rate when estimating the divergence of rtt.

For every acknowlegement, we calculate the rtt by timestamp, then update rtt\_avg and rtt\_div using following formula:

rtt\_div = ( 1 - rtt\_beta ) \* rtt\_div + rtt\_beta \* abs(rtt\_new - rtt\_avg)

rtt\_avg = (1-rtt\_alpha) \* rtt\_avg + rtt\_alpha \* rtt\_new

The default values of rtt\_alpha and rtt\_beta are 0.125 and 0.25, which are recommended by RFC6298

**Main Functions:**

react\_to\_flow\_start(), which gives response to event FlowStart.

react\_to\_ack(), which is called when an acknowledgement packet is received.

react\_to\_time\_out(), which gives response to event PacketTimeOut.

send\_new\_packets()

reset\_timer()

1. **TCPVegas**

**Main Parameters:**

* enable\_fast:

When enable\_fast is False, the instance runs as TCP Vegas.

When enable\_fast is True, the instance runs as FAST TCP.

* vegas\_alpha, vegas\_beta, vegas\_gamma, fast\_alpha:

vegas\_alpha and vegas\_beta are parameters of TCP Vegas.

fast\_alpha is a parameter of FAST TCP.

vegas\_gamma is the parameter used by both TCP Vegas, and FAST TCP.

* rtt\_alpha, rtt\_beta:

They are the same as those in TCPTahoe

**Main Functions:**

react\_to\_flow\_start()

react\_to\_ack()

react\_to\_time\_out()

react\_to\_vegas\_time\_out()

send\_new\_packets()

reset\_timer()

**Implementation**

**reset\_timer()**

This function is called when new packets are sent, the algorithm is restarted. The function manages the PacketTimeOut event, so that only the latest PacketTimeOut evenet is able the get response from react\_to\_time\_out().

**send\_new\_packets()**

send\_new\_packets() has a parameter cnt, which is the maximum number of packets can be sent by a single call of the function. The funcion trys send as many packets as possible until the number of packets sending equals windows size, or the maximum number of packets can be sent is reached, whichever is reached sooner. If the function sends at least one packets, it will call reset\_timer().

**react\_to\_flow\_start()**

This function is callsed when a FlowStart event is trigged, to start the algorithm. It only calls send\_new\_packets() to start sending packets.

**react\_to\_ack()**

This function is the most important part. The function is called whenever received a acknowledgement packet. The steps of the function are shown below:

1. The function check if the packet is already acknowledged. If so, the function will skip all the steps and return.
2. The function check if the packet is a duplicate acknowledgement. If so, the function will perform fast retransmit or fast recovery, depending on which one is enable (by parameter enable\_fast\_retransmit, enable\_fast\_recovery).
3. The function cleans up the list of packets sendings.
4. If neither fast\_retransmit nor fast\_recovery is performed, the function checks if it is slow start state now. If so, the function will increase the window size by 1 and check if it should leave slow start state. The criteria of TCPTahoe and TCPVegas are different. In TCPTahoe, it leaves slow start state when W>slow start treshold. In TCPVegas, it leavs slow start state when

W / rtt\_base - W / rtt > vegas\_gamma / rtt\_base

Which indicates that the packets starts to line up in the queue.

1. If it is not in slow start, TCPTahoe will increased the window size by 1 / W (addtive increase). TCPVegas doesn’t do this. react\_to\_vegas\_time\_out() takes charge of maintaining window size in TCPVegas.

In TCPVegas, the function will set a timeout event for react\_to\_vegas\_time\_out() if it has not been set yet.

**react\_to\_time\_out()**

The function react\_to\_time\_out() is called when a PacketTimeOut event is triggered.

In TCPTahoe, if the event is a valid one (the latest PacketTimeOut event), which indicates that there is a packet loss which is not handled by fast retransmit or fast recovery, the function will restart the algorithm, set the slow start treshold to half of current window size, decrease the window size to 1, and enter the slow start state.

In TCPVegas, it only halfs the wendow size.

**react\_to\_vegas\_time\_out()**

This function is used in TCPVegas to update the window size. It is called when a VegasTimeOut event is triggered. It will set up a VegasTimeOut event which will be triggered after rtt (the latest rtt), so that the funcion will be called every rtt.

When performing TCP Vegas, the function increases the window size by one when

W / rtt\_base - W / rtt < vegas\_alpha / rtt\_base

and decreases the window size by one when

W / rtt\_base - W / rtt > vegas\_beta / rtt\_base

By doing this, it try to keep the queue lengh in some range related to vegas\_alpha and vegas\_ beta.

When performing TCP Vegas, the function changes the window size as

W = W \* rtt\_base / rtt + fast\_alpha

Which is also a feedback system to try keep the queue length at some point related to fast\_alpha.

**9. Router**

**Class**:

* Address(Unique IP address to indentify routers)
* Links (list of links connected to this router)
* Table( It’s is dict structure in python and its value is a tuple structure. Key-->destination, Val-->(metric, next hop) )
* Default\_gateway(link[0], forward packet to next default hop when there is no specific route in router table)
* Update\_time(time interval to periodically generate router packet)

**Router Initialization**:

We mainly need to initialize router table. When initializing router table we create spaces for every host in testcases, set their distance to be 1 if the host is connected to the router, otherwise to be infinity if there is no direct connection.

We will generate first bunch of router packet immediately after initializaion.

We assume that there is enough time to update router table before first flow starts.

**Router is mainly in charge of two tasks**:

1. react\_to\_packet\_receipt()

This is a reaction after receiving a packet from links in simulation.

There are several steps following:

1. Tell what type of the packet is
   1. DataPacket==>map\_route()
   2. RouterPacket==>generate\_ack\_router\_packet()
   3. Acknowledgement of RouterPacket==>Update\_router\_table()
2. Call different funtions according to type of packet
   1. map\_route(): Get the destination of packet, then look up current router table, find next hop and send the packet to next link.
   2. generate\_ack\_router\_packet(): This is a function to react after a neighbor router receiving a routerpacket, it will generete a new router packet which parameter of acknowledgement is True, router\_table is a copy of current neighbor routing table, timestamp keeps the timestamp of original RouterPacket it received.
   3. Update\_router\_table(): Compare router table from acknowledgement of router packet received from neighbor routers and the current router. Update router table if there is a shorter path. The rule of update is Bellmen-Ford algorithm which we will explain later.
3. react\_to\_router\_table\_outdated()

* Set a timer(with constant time interval for each testcases) to send RouterPacket periodically so that we are able to update routertable.
* Generate\_router\_packet (source, timestamp, routertable): Parameter of source records IP address of router so that its neighbor can recognize where is the router packet from and send acknowledgement back; Parameter of timestamp keeps the current timestamp when generating this packet, we could use this timestamp to calculate link delay.
* Send to all links connected to this router.

**Routing algorithm: Bellmen-Ford algorithm**

Static routing: metirc based on hops( 1 hop for each link )

Dynamic routing: metric based on link delays ( the difference of timestamps )

For each host destination, if the metric + distance in neighbor router table is small than current distance in present router table, then set distance of present router table to be metric + distance in neighbor router table, and set the next hop of the route to host destination to be the IP address of neighbor router.

**Test result:**

For testcase1 based on hops:

‘0’ denotes H1, ‘1’ denotes H2

At 1190 R1 received RouterPacket from R3

{'1': (3, 'R2'), '0': (1, '0')}

At 1190 R2 received RouterPacket from R4

{'1': (2, 'R4'), '0': (2, 'R1')}

At 1190 R4 received RouterPacket from R3

{'1': (1, '1'), '0': (3, 'R2')}

At 1190 R3 received RouterPacket from R4

{'1': (2, 'R4'), '0': (2, 'R1’)}

For testcase2 based on hops:

At 1190 R1 received RouterPacket from R2

{'S3': (3, 'R2'), 'S2': (1, 'S2'), 'S1': (1, 'S1'), 'T2': (2, 'R2'), 'T3': (4, 'R2'), 'T1': (4, 'R2')}

At 1190 R2 received RouterPacket from R3

{'S3': (2, 'R3'), 'S2': (2, 'R1'), 'S1': (2, 'R1'), 'T2': (1, 'T2'), 'T3': (3, 'R3'), 'T1': (3, 'R3')}

At 1190 R3 received RouterPacket from R4

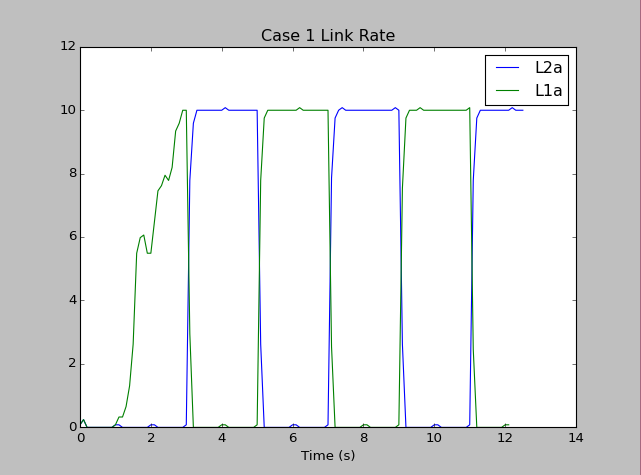
{'S3': (1, 'S3'), 'S2': (3, 'R2'), 'S1': (3, 'R2'), 'T2': (2, 'R2'), 'T3': (2, 'R4'), 'T1': (2, 'R4')}

At 1190 R4 received RouterPacket from R3

{'S3': (2, 'R3'), 'S2': (4, 'R3'), 'S1': (4, 'R3'), 'T2': (3, 'R3'), 'T3': (1, 'T3'), 'T1': (1, 'T1')}

From the result based on hops, we see that router table will converge to what it supposed to after a proper time.

For testcase1 based on link rate:



For the dynamic case, we see that switch its routes between link1 and link2 due to congestion status.

**10. Events**

FlowStart

* A timer which tells the flow when to start.

LinkAvailable

* A timer which tells the link it’s available because the link has finished sending the packet.

PacketReceipt

* A timer which tells the receiver it receives a packet.
* Has a reference to the packet in its attribute “value” (this is how packets are passed).
* It works for both host and router, which makes debugging more convenient.

PacketTimeOut

* A timer which calls tla.react\_to\_time\_out.

VegasTimeOut

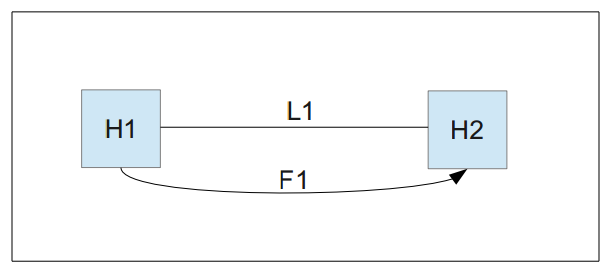
* A timer which calls tla.react\_to\_vegas\_time\_out .

RoutingTableOutdated

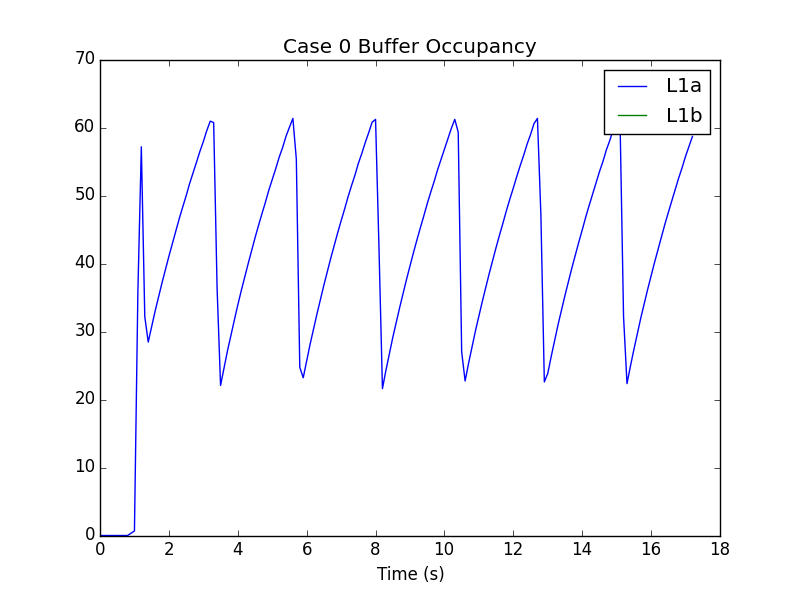
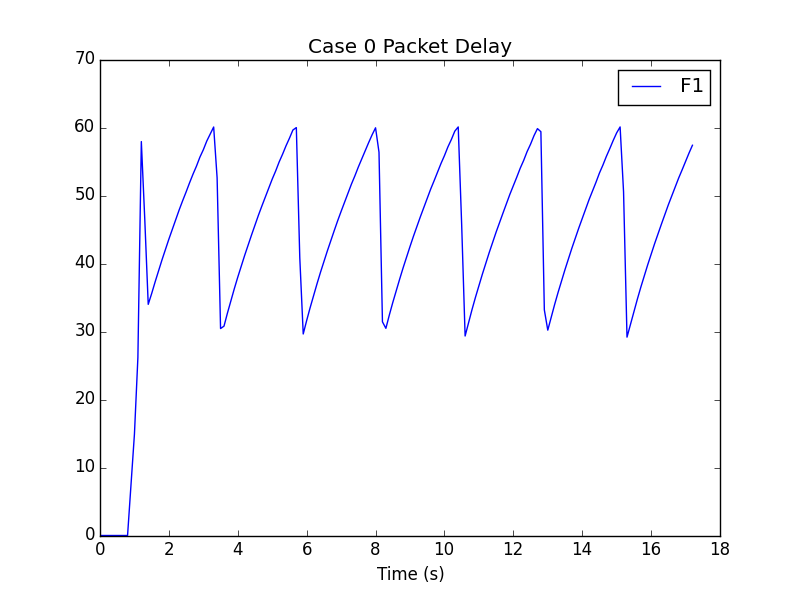
* A timer which tells a router to start updating its routing table by making RouterPackets.

**11. Test Cases and Analysis**

Test Case 0



When using TCP Tahoe with fast retransmit, when it enters equilibrium, the queue length of L1 should always go from half of the buffer size (ignoring the slow start) to the buffer size, linearly, then drops back to half of the buffer size. The packet delays should show the same pattern.

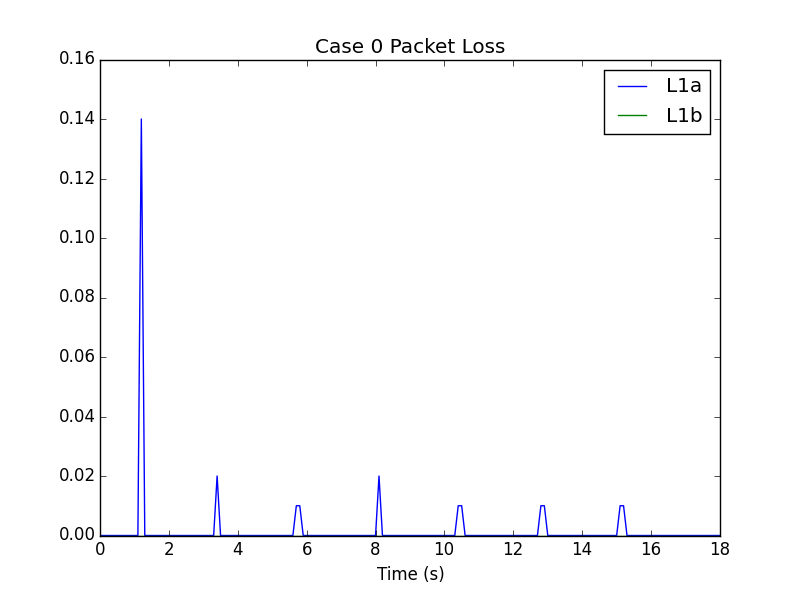
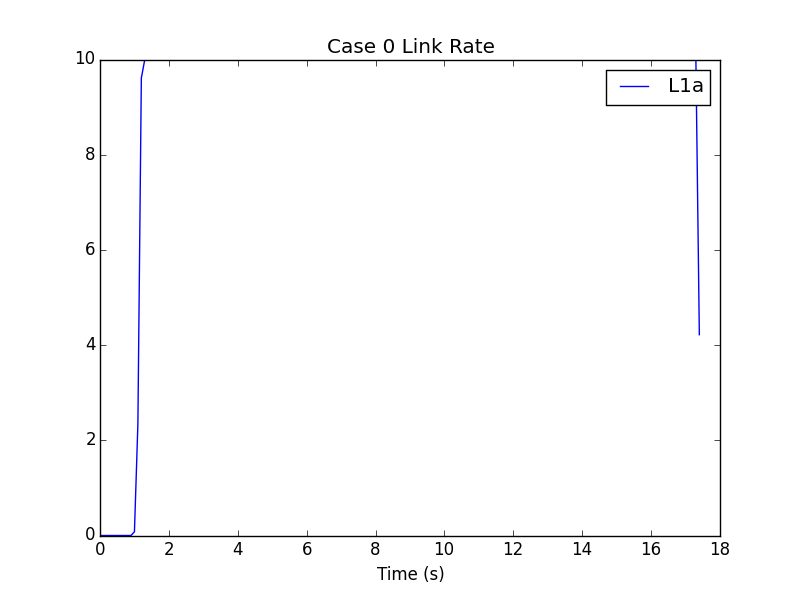


We can calculate the packet delay as below:

The buffer size is 64KB, which is 64 packets. It takes about to transmit a packet. If we consider the transport delay, the packet delay should be around from to . Which fits our graph well.

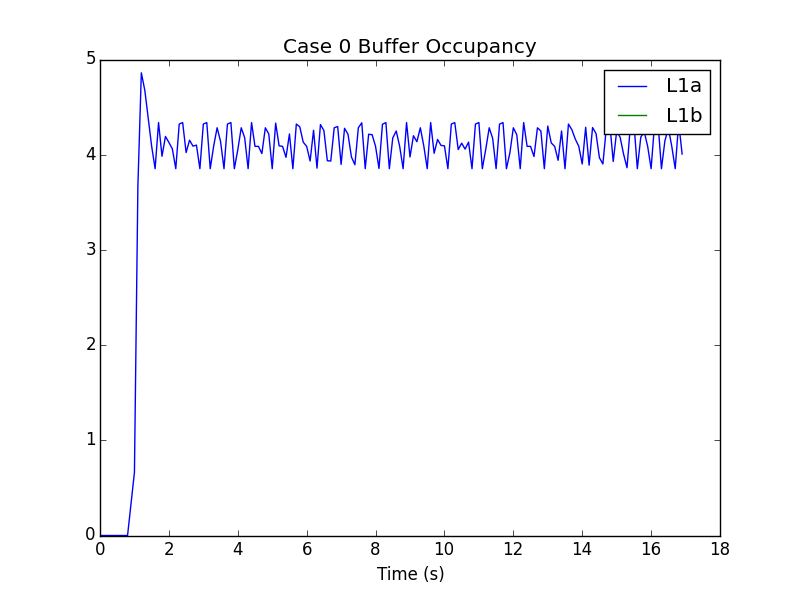
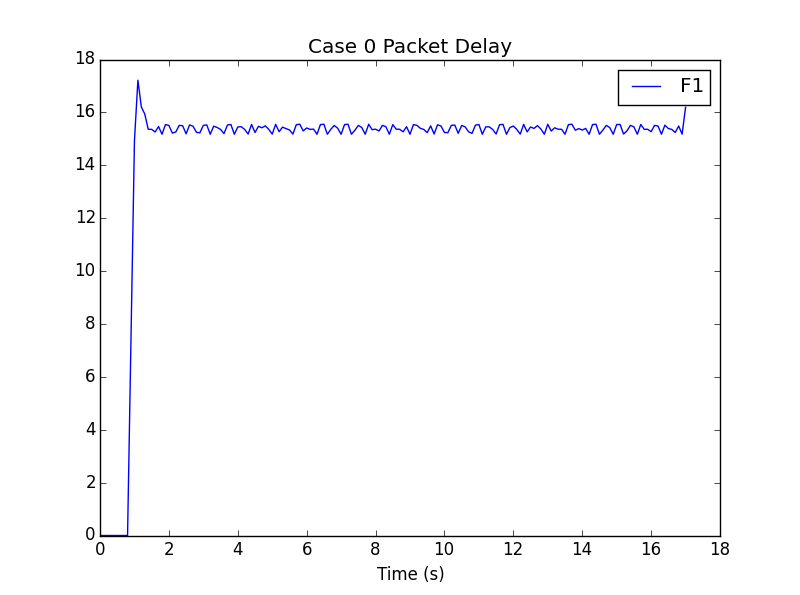
Once the equilibrium state is reached, the link rate will be nearly fully used. So the total time for transmitting data will be around. Which also fits our graph.

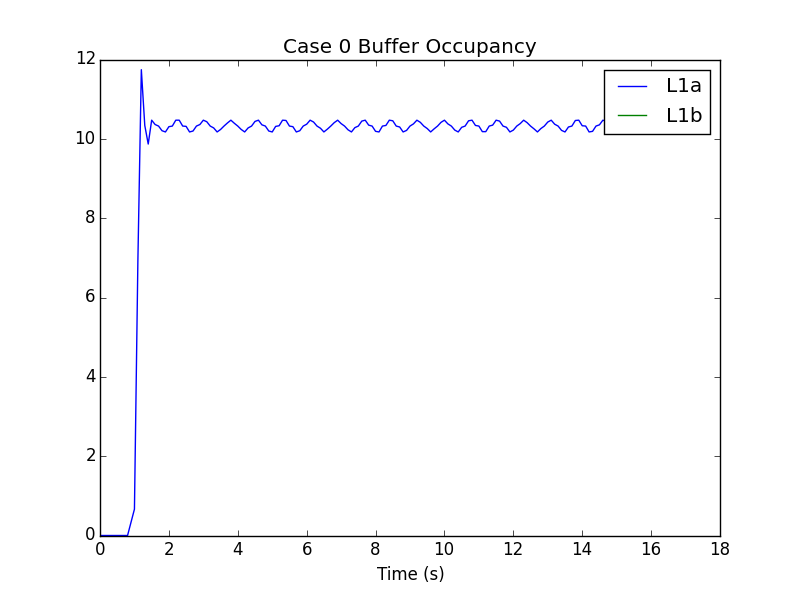
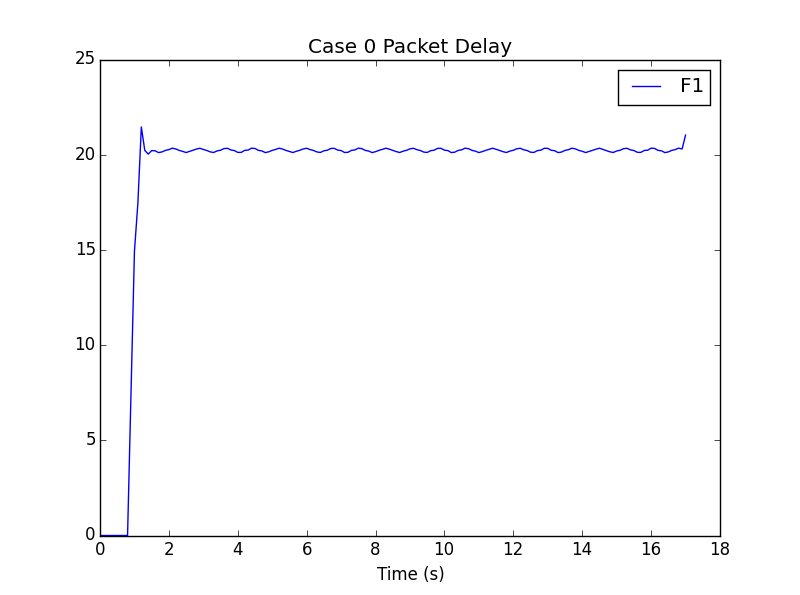
The disadvantage of TCP Tahoe is the periodical packet loss. We can see that in our graph.



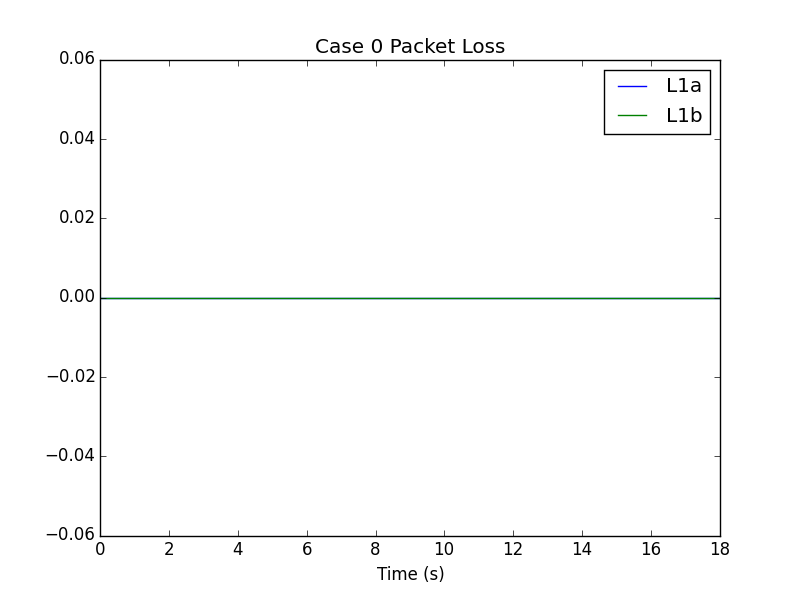
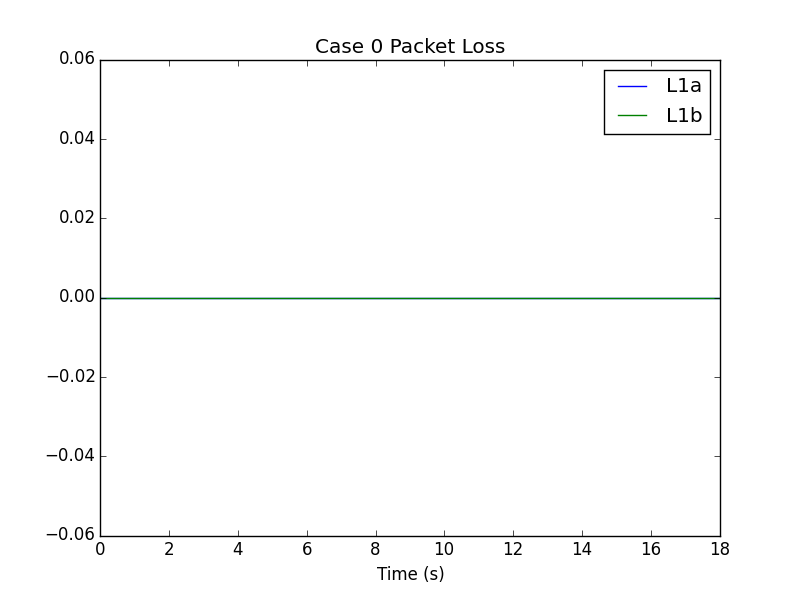
Therefore, we can conclude that the algorithm runs as what we can calculate analytically.

When using fast TCP, when equilibrium is reached, the queue length is around (as what we have already calculated from our previous homework). And the packet delay is around . The result are roughly the same as our simulation results.

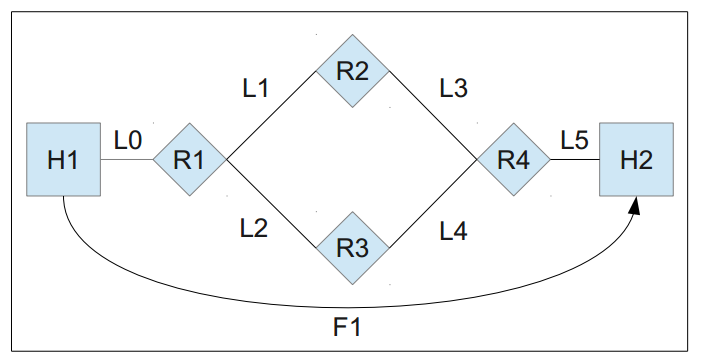




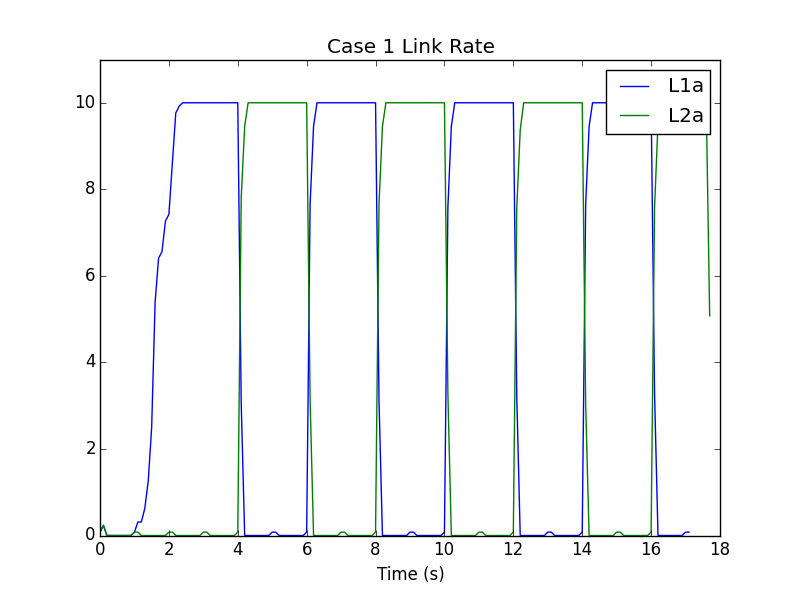
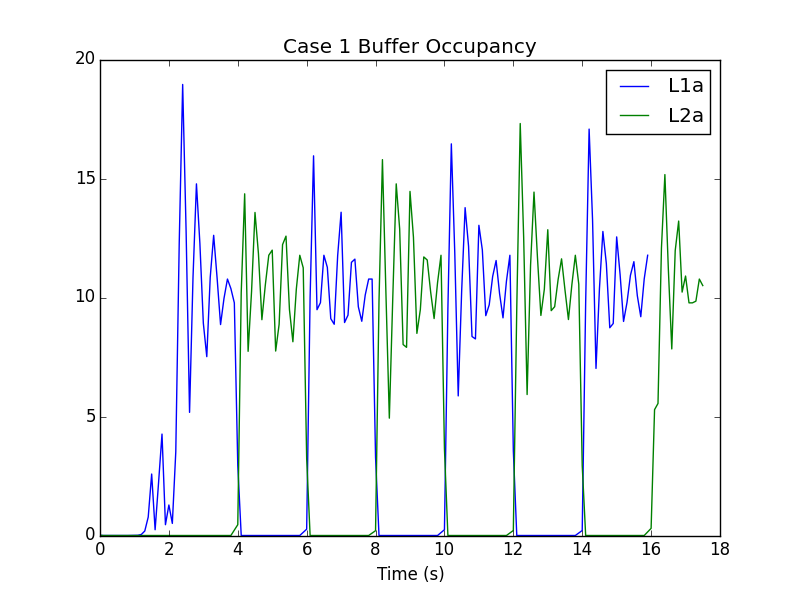
We can see that the packet delay of FAST TCP is much less than that of TCP Tahoe. The other advantage of FAST TCP over TCP Tahoe is that it don’t need periodically packet loss. Actually, in our simulation, both in and in , no packet loss is detected.



Test Case 1

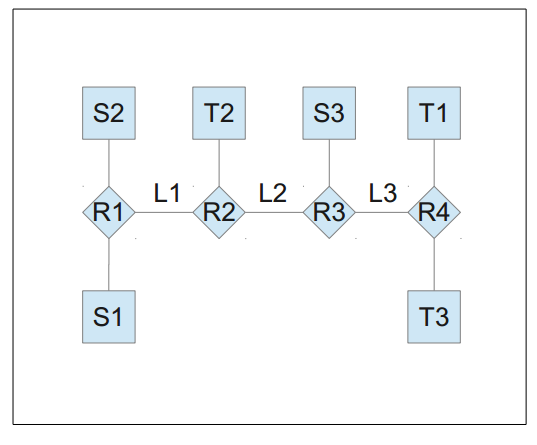


Based on Test Case 1, we analysis some issues related to dynamic routing (based on FAST TCP, ). When dynamic routing is used, the flow change the route between H1->R1->R2->R4->H2 and H1->R1->R3->R4->H2, so we can see the link rate of L1 and L2 are exchanged periodically. It should not make much difference to the flow rate. So it will still take around 16 seconds to transmit the data.

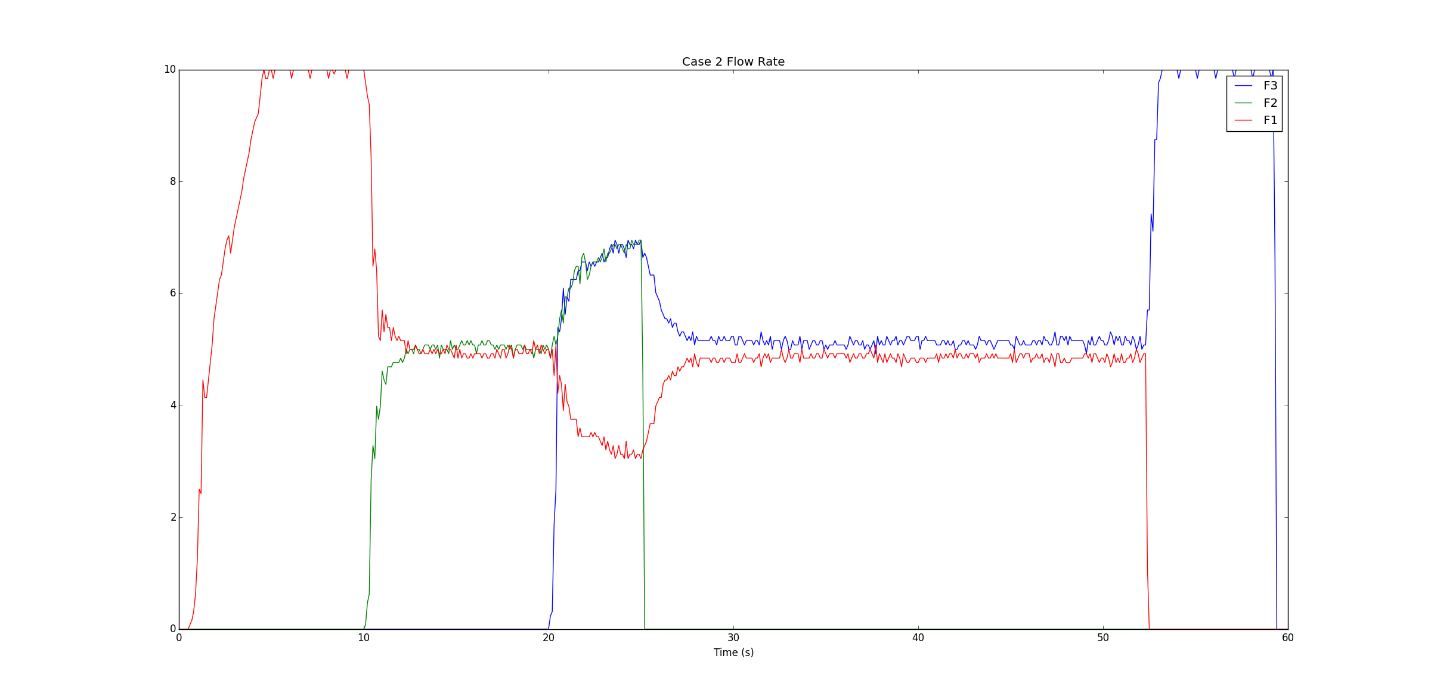


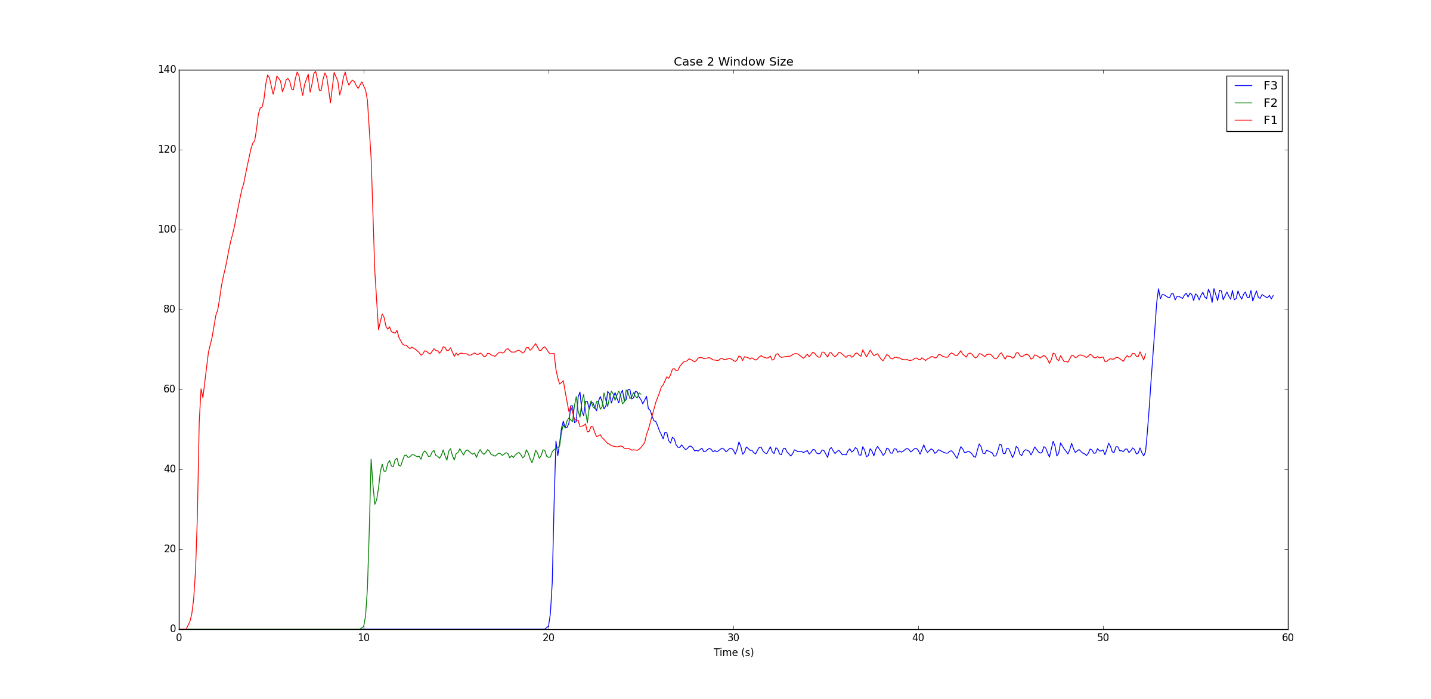
We can see that the oscillation of buffer size is much heavier than in test case 0. It might be caused by that before it runs into a more stable state, the routing has changed.

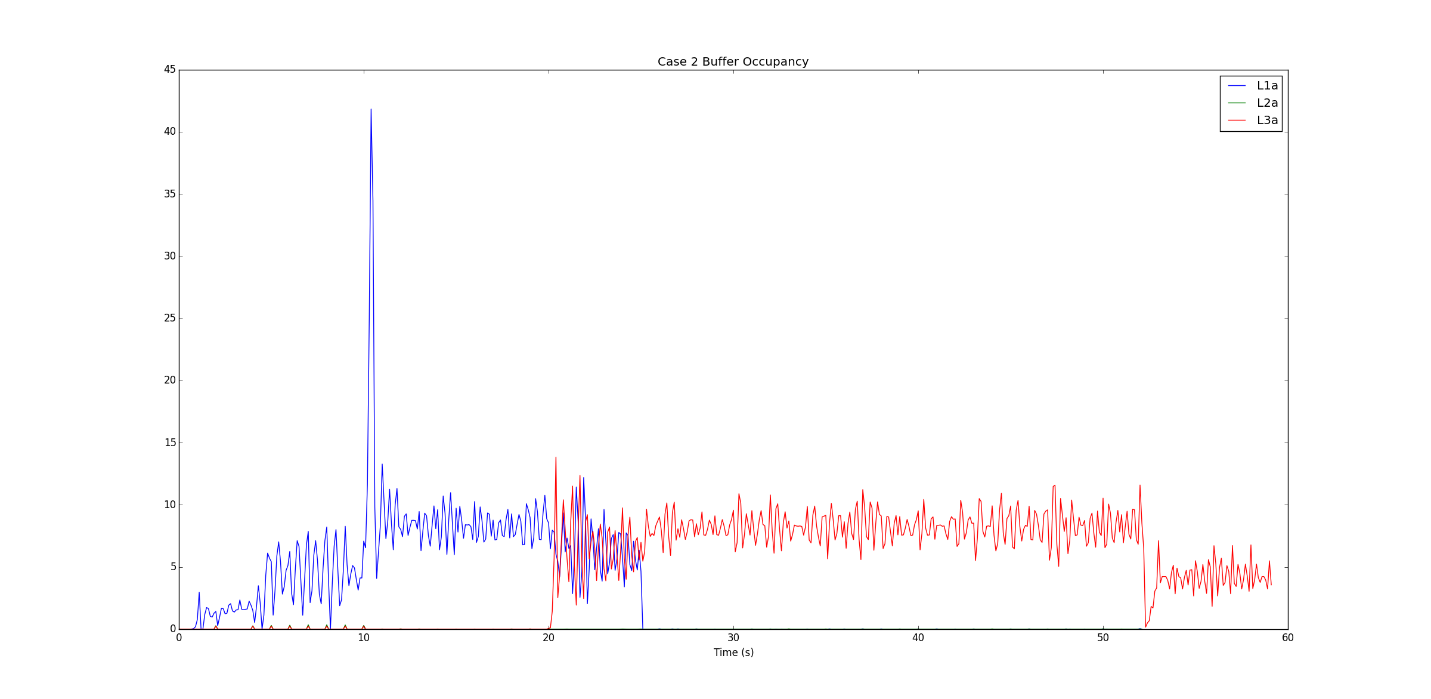
Test Case 2



In this case, we calculate different equilibrium point.







Calculation (basically the same as hw3, except for some parameters)

From , we can get

The corresponding continuous model of this FAST is shown below:

where

( is what the algorithm actually using, calculated by the smallest rtt)

0S~10S: Only flow 1 working. All packet are buffered at

Let

Then

Therefore

Then

Throughput rate of flow 1:

From Little’s law, Queue length of link 1:

Queue length of link 2 and link 3 is zero (from assumption).

10S~20S:

As , , , would not be met at the same time. Intuitively, we consider , , , since this is the only not trivial state for .

Therefore, we have

Let

Then

Since we have

We can get

Throughput of flow 1:

Throughput of flow 2:

Queue length of link 1:

Queue length of link 1:

Queue length of link 1:

20S~

Since , , the only not trivial steady state for is

We have

Let

Then

We have

Since

We have

The solution is

Then

Throughput of flow 1:

Throughput of flow 2:

Throughput of flow 3:

Queue length of link 1:

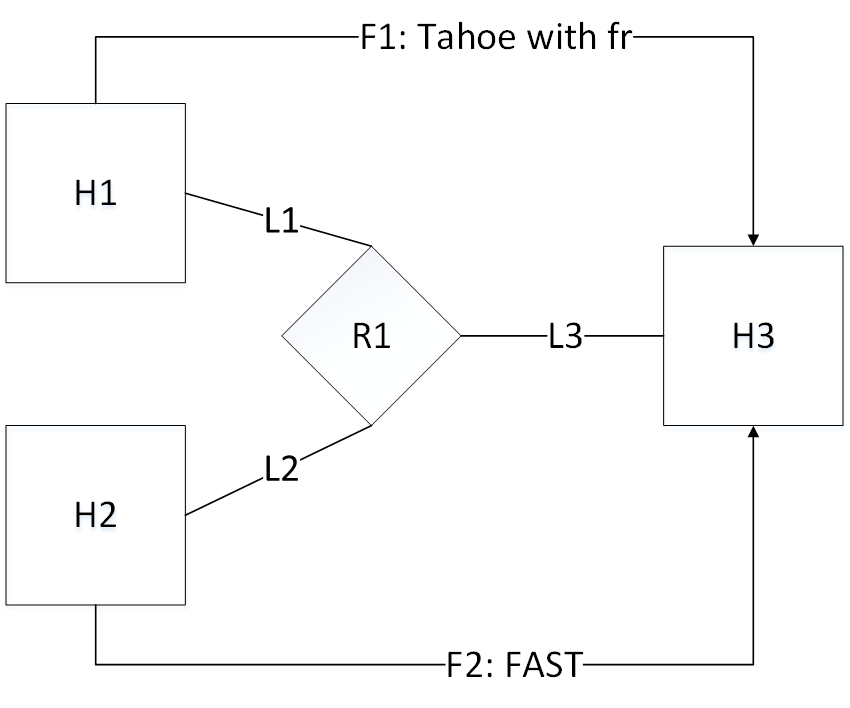
Queue length of link 1:

Queue length of link 1:

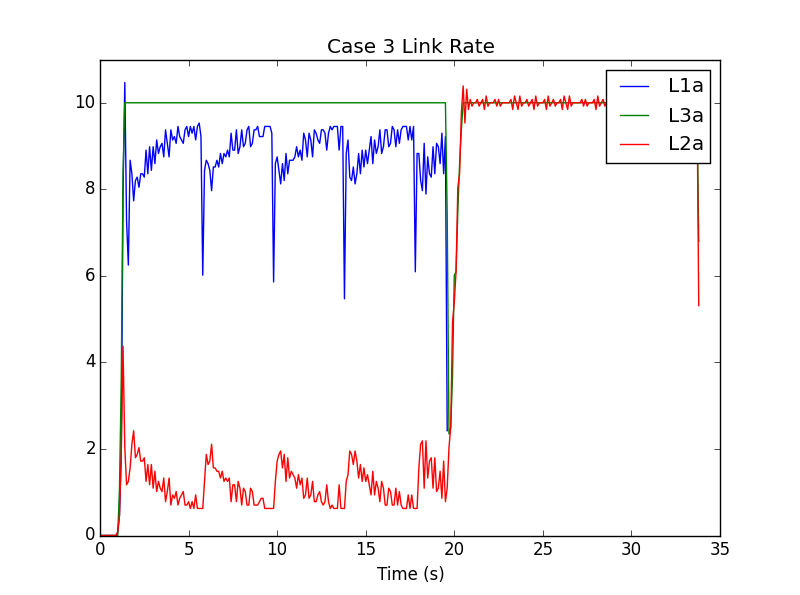
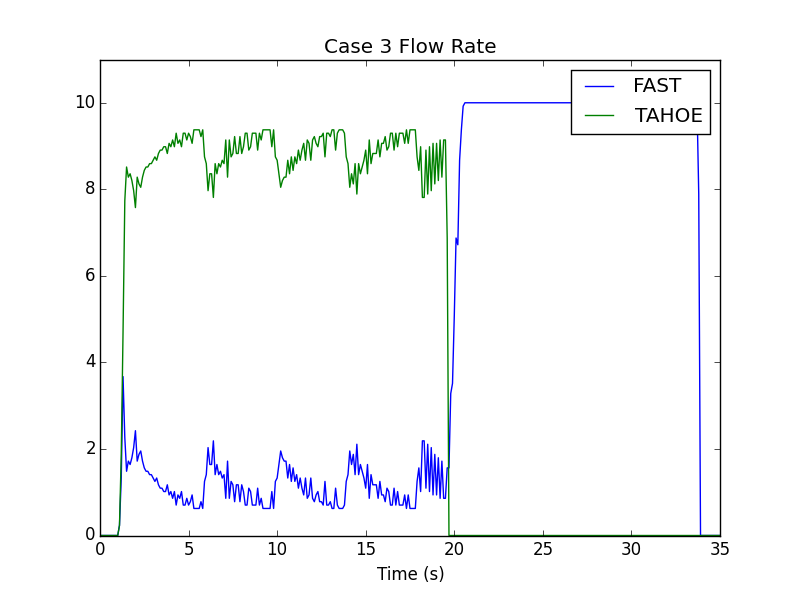
Then , when only Flow 1 and Flow 3 are running ,it’s approximately the same as Flow 1 and Flow 2.

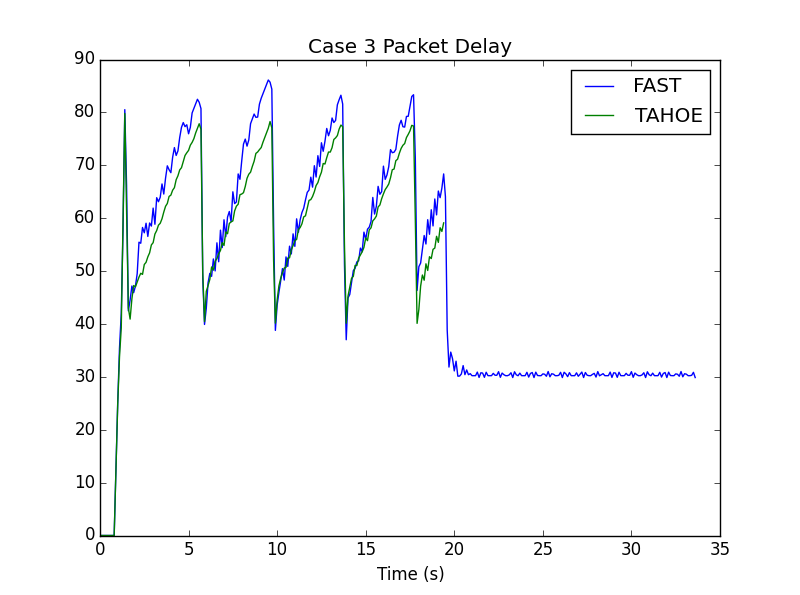
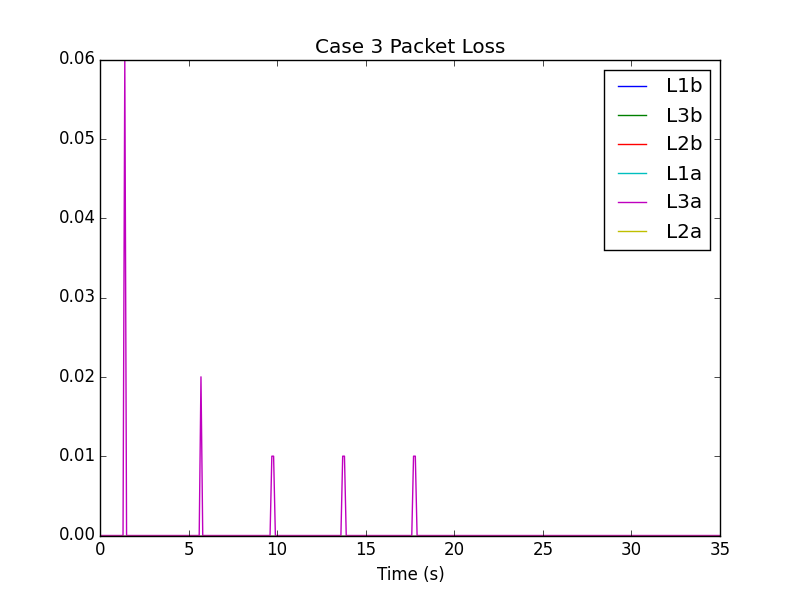
**OUR SIMULATION RESULTS FIT THE ANALYTICAL RESULTS PERFECTLY**

Test Case 3



This case is run for showing what happens when two different protocols run in the same system.





From the simulation, we can see that when both flows are running, Tahoe dominates the system. And we can see the packet loss and packet delay due to Tahoe. When Tahoe is not running, the delay decreases a lot.

**12. Our process**

**Milestone 1 (Week 5)**

**Progress**

* Chose Python and SimPy because all the team members are familiar with them.
* Decided the architecture of the project (Flow, Host, Link, Buffer, Router) and implement these classes.
* Used two one-way links to simulate the two-way link.
* Implemented static routing algorithm.
* Decided the units and constants.
* Set up a repository on GitHub.
* Set up a documentation site on ReadTheDocs.
* Set up a test fixture using PyTest.

**Obstacles**

* None, as we had just begun and were making fast initial progress.

**Milestone 2 (Week 8)**

**Progress**

* Implemented the Bellman-Ford algorithm using number of hops as the metric.
* Finalized our input structure.
* Built a Controller and events to run the simulations on SimPy.
* Ran the test case 0 and 1 and the outputs seemed correct.
* Implemented Tahoe and Reno.

**Obstacles**

* Coordinating everyone’s first attempts at writing the code, as we needed to agree on many little implementation details.
* The routing tables didn’t update correctly when we tried to use packet delay as the metric.
* We saw multiple packet losses during one RTT.

**Milestone 3 (Week 10)**

**Progress**

* Fixed the bug of multiple packet losses during one RTT.
* Refined the way the routers update their routing tables (sending an update request to the router’s neighbors and waiting for replies) so the result of test case 1 was correct.
* Implemented Fast TCP and Vegas.
* Drew graphs of outputs using matplotlib.
* Experimented with multiple transport layer algorithms in one simulation.

**Obstacles**

* The values for flow rate are not consistent with the values for link rate.
* The values for window size don’t match the analytical expectations.

**After final presentation (Week 11)**

**Progress**

* Cleaned up our code.
* Fixed flow/link rate inconsistency bug by counting only DataPackets for the flow rate.
* Fixed window size discrepancy by including transmission delay when calculating analytical expectations so they match our simulation results perfectly.
* Wrote more documentation.
* Finished this final report. :)

**What we learned**

* Start repository/documentation/test setup early, so mid- and late-project coding is much easier and faster.
* Consequences of minor differences between transport layer algorithms.
* Constant email communication through a group email list is effective.
* Improved debugging skills.
* The ease of setting up a simple simulation of a seemingly complicated system.
* The importance of logical, abstracted architecture elements.
* Deeper understanding of the structure and behavior of internet-like networks.