Computer and Communication Networks

Group 3

“TCP Tahoe with More Realistic Time Simulation and Packet Reordering”

12/23/2010

Steven Carvellas

Jeril Jose

Project Contribution

Steven Carvellas:

- Algorithm Design:
- Coding: 90%
- Debugging: 50%
- Report Preparation: 70%

Jeril Jose:

- Coding: 10%
- Report Preparation: 30%
- Debugging: 50%

Table of Contents

1. Introduction.....4

2. Algorithm Design.....8

3. Implementation.....13

4. Results and Discussion.....17

5. References.....21

1. Introduction

TCP congestion control is key in most network systems implementing the TCP/IP protocol for reliable data transmission. To deal with the different complexities and styles of network systems, whether there are wireless systems, prioritized traffic, or vast link speed changes, different congestion algorithms are designed and implemented to deal with various conditions and implementations that may be involved in a network system. TCP Tahoe is one of the more long standing and straightforward congestion algorithms, while less widely used is largely the base for study (TCP Tahoe has become less favorable than today's most common congestion algorithm, TCP NewReno.)

To provide a brief background, Standard TCP Tahoe was developed with key features implemented for congestion control. *Packets*, or datagrams, are transmitted between senders and receivers through a pipeline medium, which could potentially be an ethernet wire or wireless medium, for example. These packets are sent over intervals called *round trip times* (RTT's) that include the time a packet is sent from the sender and the time it is acknowledged by the receiver. TCP begins with *slow start* in which the congestion window is increased by one MSS on every acknowledged packet, and *multiplicative decrease*, in which after a packet times out, the CongWin reduces to 1 MSS and the SSThresh is reduced. Out of order segments, or *dupACKs*, are sent by the receiver and counted by the sender, when more than three are received the sender acknowledges a packet loss and adjusts parameters as well as resends oldest unacknowledged segment. When the congestion window exceeds SSThresh, *congestion avoidance* kicks in (also known as *additive increase*) in which the congestion window begins to increase again. If a packet is sent and an acknowledgement is not received by the retransmission timeout interval (RTO), *exponential backoff* occurs where the next timeout interval is doubled, until an acknowledgement is received, than the timer is restarted and calculated using a different method. This process continues to repeat, and this repetitive process is what is known is congestion control, and is the heart of TCP.

The **EffectiveWindow** parameter is determined by this equation:

$$\textbf{EffectiveWindow} = \min \{ \text{Congwindow}, \text{RcvWindow} \} - \text{FlightSize} \qquad (1.1a)$$

This effectively changes our definition of the number of packets that may be sent per RTT to the speed determined by the link and segment size, as long as it is <= EffectiveWindow, which is dynamic and continuously changes.

$$\textbf{FlightSize} = \text{lastByteSent} - \text{lastByteACKed} \qquad (1.1b)$$

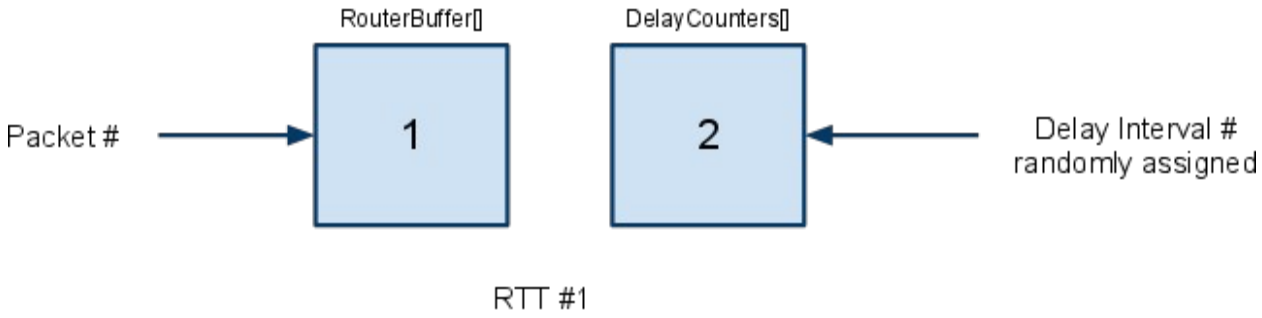
Where the flight size contains the amount of data that has been sent, but not yet acknowledged.

CongWin = Current CongWin Multiplier x MSS (maximum segment size)

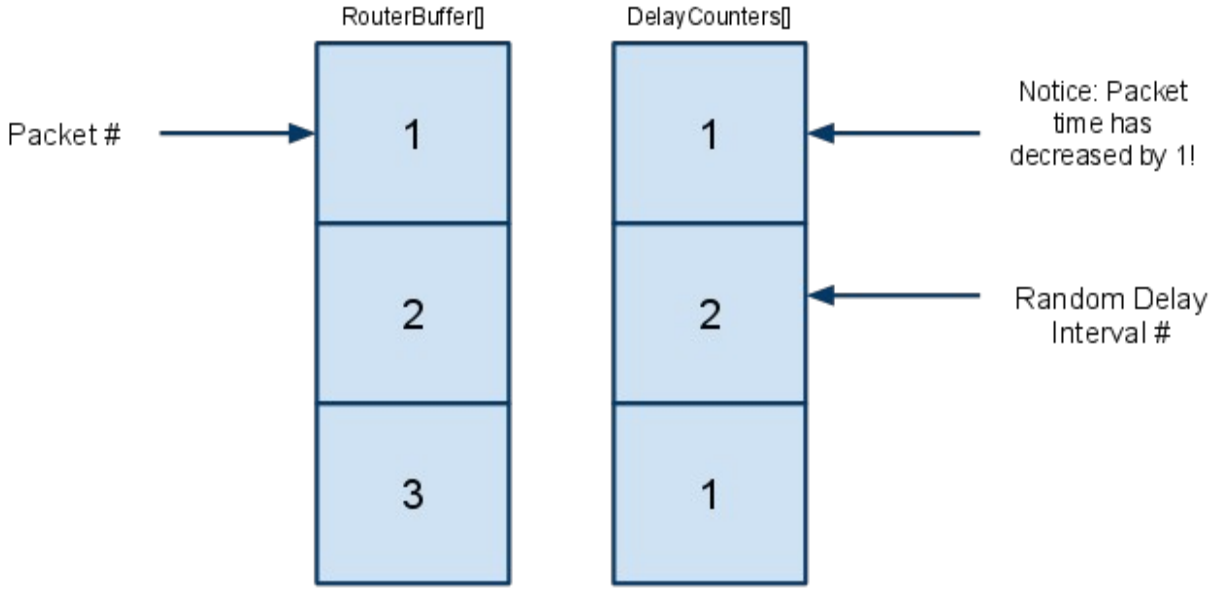
From the above equations, the EffectiveWindow controls the amount of data sent at a time, data cannot be sent when FlightSize is equal to either the Congwindow or RcvWindow. It is also not possible to have a negative EffectiveWindow. However, the more ACKs the receiver obtains, the smaller the flightsize, as well as a larger CongWindow. Larger CongWindow and smaller FlightSize means larger EffectiveWindow, in which we can send more data.

These key terms defined above will be used to describe the behavior of several experiments conducted within this paper.

This document explains the process and implementation behind modifying the TCP Tahoe simulator in the example implementation. In the modified version of this project, the task is to simulate packet reordering by having two different buffers stored in the router. One buffer holds packets before they are sent to the receiver, and another buffer holds the packet delay in RTT's. These arrays are called, respectively, RouterBuffer_[] and DelayCounters_[]. Below is a figure showing a few sample packets in the buffer:


RouterBuffer[]
DelayCounters[]
Packet #
1
2
Delay Interval #
randomly assigned

RTT #1


RouterBuffer[]
DelayCounters[]
Packet #
1
2
3
1
2
1
Notice: Packet
time has
decreased by 1!
Random Delay
Interval #

RTT #2

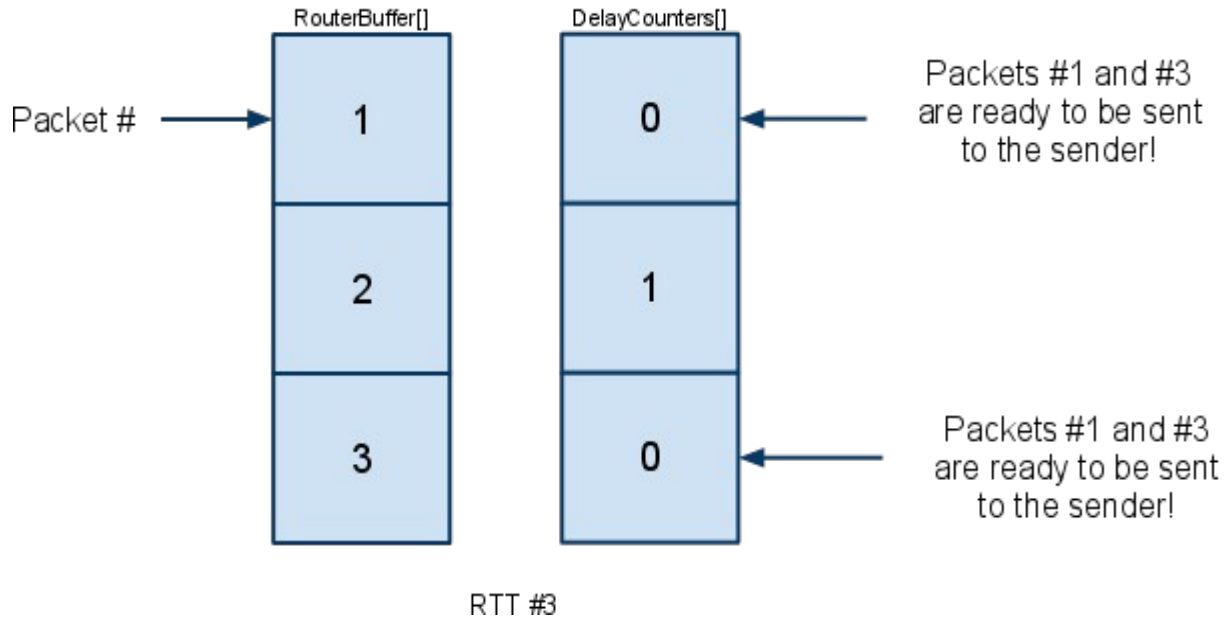


Figure 1.1: *Packet Reordering*

To simulate reordering, once a packet enters the router stage from the sender, this packet has a random number applied to it to specify timing. The router then checks all the packets in the buffer and sees if this timer is 0. Once a packet has a time of 0 remaining, this packet is forwarded to the receiver, which then sends an acknowledgement. The purpose of this project is to see how the packet reordering affects TCP behavior as compared to the originial TCP simulator implementation where each packet is received by the router in order, as well as how several of the TCP parameters are affected, such as EffectiveWindow, SSThreshold, the number of dupACKs sent, and the amount of times TCP needs to enter congestion avoidance/slow start.

2. Algorithm Design

To implement different packet timings to simulate packet reordering at the router level, the Random number generator object included with Java is used to generate small integers of delay specified on the command line. This project includes different types of probability curves that may be selected at the command line to compare different models and how they affect the results.

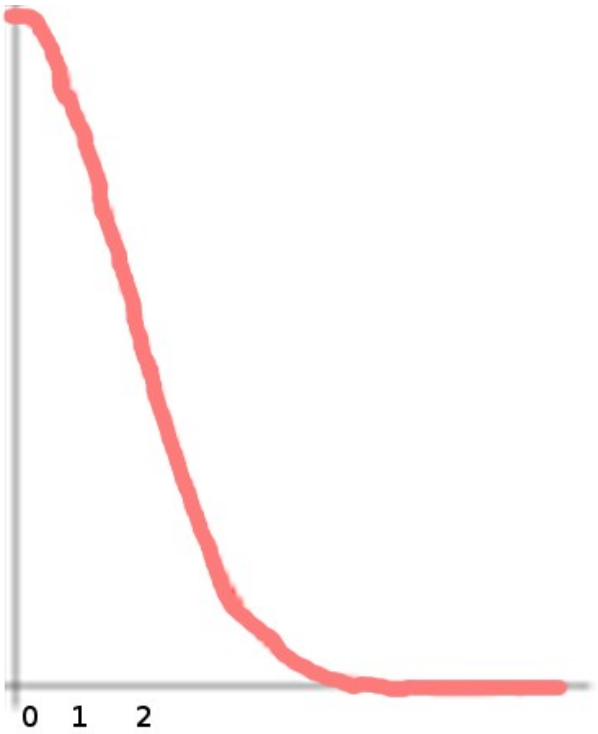
The common implementation for the project favors smaller delay counters, to do this we use a Gaussian curve with a mean of 0 and a distribution that the user enters on the command line, any negative number returned by this distribution has the absolute value applied to make this a positive integer.

$$P(x) = \left[\frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \right]$$

$$where\ \sigma^2\ is\ the\ variance\ and\ \mu\ is\ the\ mean,\ on\ the\ interval\ (-\infty,\infty)$$

Equation 2.1: *The equation for the Gaussian probability while rejecting negative numbers*

And the graph for the distribution below:



Graph 2.1: *Gaussian Probability Curve*

Another model that may be entered in the command line is the exponential decay model, which is slightly different than the Gaussian curve because it even more heavily favors lower numbers by causing a rapid decline in slope that can be observed in the graph below.

$$N(t) = N_0 e^{-\lambda t}.$$

Equation 2.2: *The equation for the Exponential probability*

where the half life of material can be determined using the following formula:

$$\tau = \frac{t_{1/2}}{\ln 2} = 1.442695040888963 \cdot t_{1/2}.$$

Equation 2.3: *Using half-life to adjust the slope of decay in the exponential curve*

After rejecting numbers greater than the user provided maximum delay.

Both methods are similar in that they more accurately simulate packet reordering by having it happen less frequently, instead of at completely randomized intervals.

As the results are randomized and there will be different results on each run, there is no methodical way of predicting the results.

The program accepts three different command line parameters, *number of RTT's, Maximum Delay Count, and Random Number Probability Curve*.

Number of RTT's specifies how many RTT's the simulation runs for. To provide meaningful output, this needs to run for a long amount of time. This report uses 100 iterations for most of the data obtained in the results section.

Maximum Delay Count is how the end user can control what the largest delay that can be randomly generated can be. Typically to provide a slightly functioning TCP algorithm, this value needs to be small, around 2 or 3. You should avoid putting a high number otherwise TCP will timeout and not provide any useful information.

Random Number Probability Curve selects which type of probability curve to use to generate random numbers for the DelayCounts. *1* will select the Gaussian method described above, any other number will select the Exponential Decay model.

Below are two flow charts showing altered areas of the code:

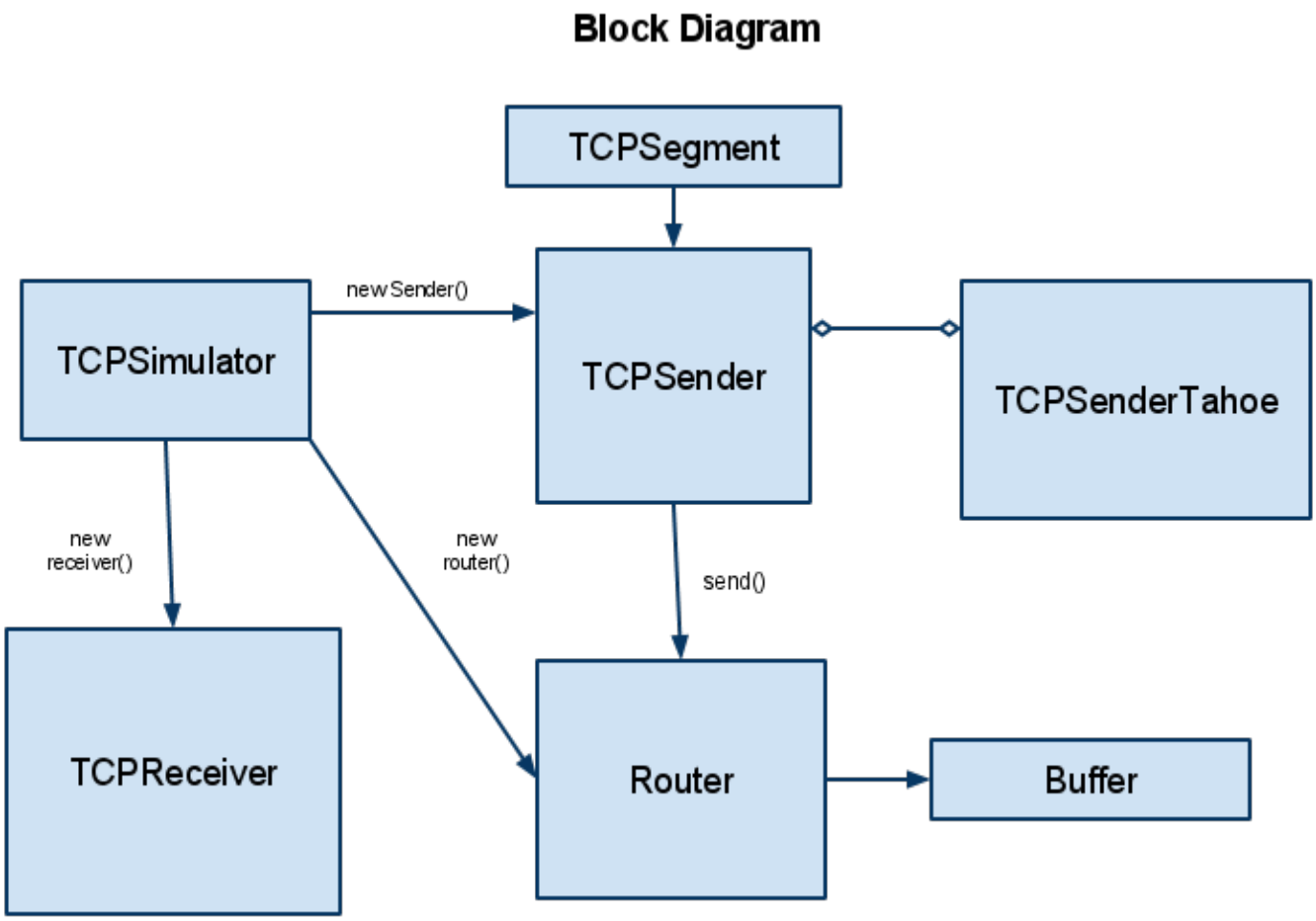


Figure 2.1: Block Diagram

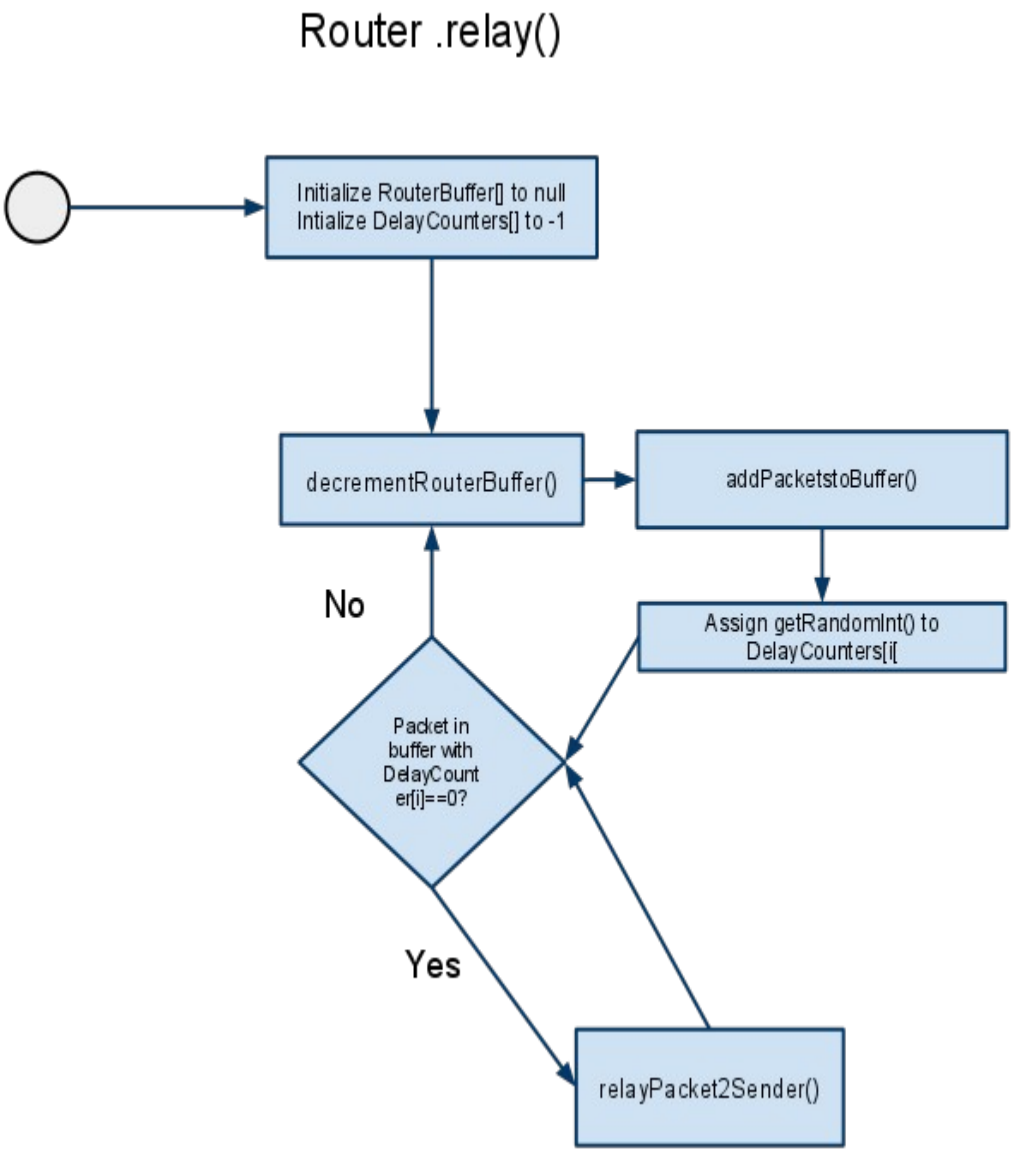


Figure 2.2: Router

3. Implementation

The project was implemented using the Java programming language and the reference code given on the course website to implement TCP Tahoe with a more realistic time simulation and packet reordering. The project was developed using the Eclipse IDE. The README file submitted along with the source code describes how to run the program and modify the user-controllable parameters.

The source code contains 5 classes. TCPSimulator, TCPSender, TCPReceiver, TCPSenderTahoe and Router. Only the Router class is modified for this project

Router

The router is initialized by TCPSimulator class.
Two extra fields, RouterBuffer[] and DelayCounters[], are added to the Router class:

```
RouterBuffer_ = new TCPSegment[buffer_size];
DelayCounters_ = new Integer[buffer_size];
```

The constructor of Router intializes the array RouterBuffer[] to null and DelayCounters[] to -1:

```
for (int x=0; x<buffer_size; x++)
{
	RouterBuffer_[x] = null;
	DelayCounters_[x] = -1;
}
for (int x=0; x<buffer_size; x++)
{
	RouterBuffer_[x] = null;
	DelayCounters_[x] = -1;
}
```

The relay() method in Router processes the packets sent by sender method.

```
public void relay(TCPSegment[] packets_) {
```

```
                decrementRouterBuffer(); // STEP 1!
                addNewPackets2Buffer(packets_); // STEP 2!
                relayPacket2Sender(packets_); // STEP 3!
}
```

First the method decrements all values in DelayCounters[] by 1. If the value of DelayCounters[] is -1 then it is considered a null packet.

```
public void decrementRouterBuffer()
{
        int i=0;
        while (DelayCounters_[i] != -1)
        {
                DelayCounters_[i++]--;
        }
}
```

Once the value of DelayCounters[] is 0, this means the packet is ready to be sent to the receiver.

The second step is to copy all packets from the packets_[] array. The packets_[] array contains all the packets sent by the sender for that particular RTT, as well as which packets will get relayed to the sender after the router stage. These packets are append to the array RouterBuffer[]. After appending all the packets to RouterBuffer[] , all the elements in packets_[] array is set to null to prevent buffered packets from getting sent to the receiver. For each packet added to RouterBuffer[], the corresponding index in DelayCounters[] is filled with a random number.

Finally we loop through all elements in RouterBuffer[] and DelayCounters[] and see if there is any packet in RouterBuffer[] with a corresponding delay of 0 in DelayCounters[]. If there is, that packet and its delay are removed from RouterBuffer[ ] and DelayCounters[] respectively and the packet is added to packets_[] array to be relayed to the sender. The elements of RouterBuffer[] and DelayCounters[] are shifted left to fill the gap created by removing the packet with a delay of 0.

```
if (DelayCounters_[i] == 0)
{
        while (packets_[freespot] != null) freespot++;
                packets_[freespot] = RouterBuffer_[i];
                input_count--;
                // Shift remaining elements
                for (int x = i; x < buffer_size-1; x++)
                {
                RouterBuffer_[x] = RouterBuffer_[x+1];
                DelayCounters_[x] = DelayCounters_[x+1];
```

```
                }
                RBindex--;
                i--;
}
```

In the end the packets_[] array is forwarded to the receiver class. This repeats for a number of RTTs specified by user. This effectively simulates packet reordering at the router stage. The reordering is done to simulate when different packets arrive at different time. This reordering was expected to create many duplicate acknowledgments, therefore switching the router continuously between slow start and congestion avoidance.

Attached is a User Manual (README) file for the program:

Running TCP Tahoe with More Realistic Time Simulation and Packet Reordering
==========

Requires Java 2 or higher, can be compiled with the command javac TCPSimulator.javac.
Then the code can be executed using:

java TCPSimulator (iterations) (delay) (probabilitymodel)

Where each option in parenthesis can be replaced with a parameter explained below.

(iterations) can be replaced with the number of RTT's to run the simulator until.
The value tested for development of this program is 100.

(delay) specifies the maximum random number generated for each packet's DelayCounter.
The higher this number is, the more chances increase of sender timeouts.
A good value to put here is between 1-3 to effectively simulate packet reordering.

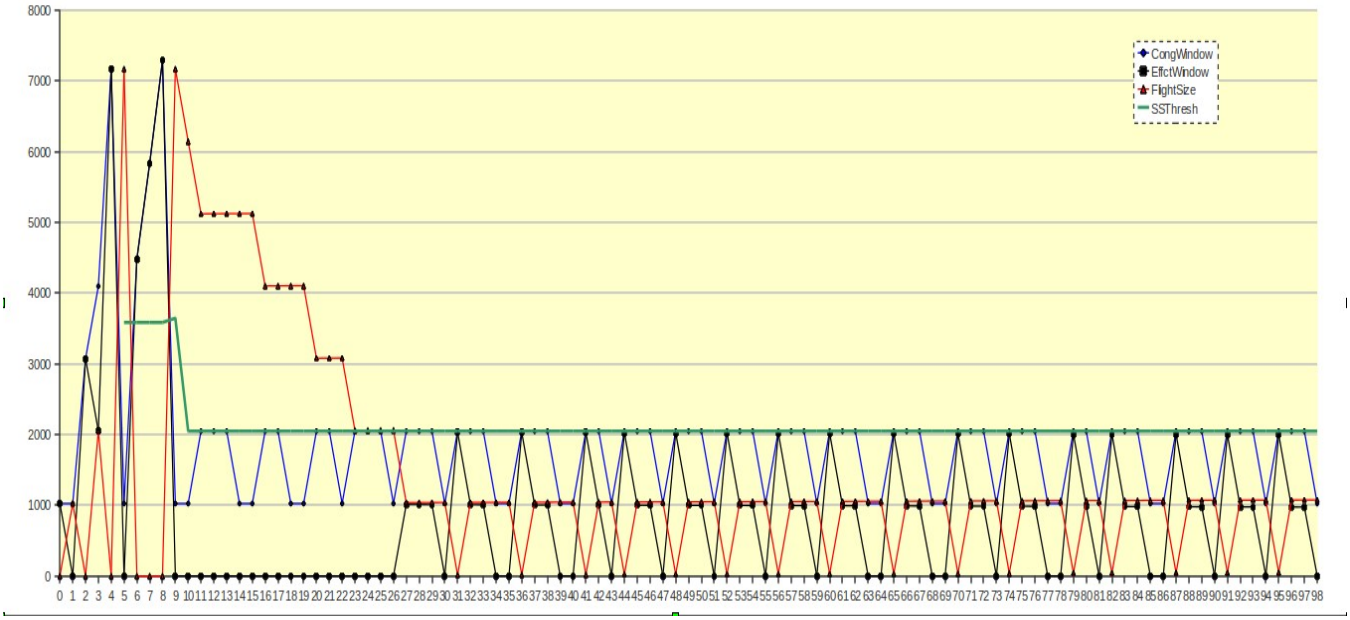
(probabilitymodel) specifies how random numbers are selected. The program currently contains
two different models for selecting random numbers, either by using a random Gaussian distribution or
a random exponential decay distribution.

0 uses an exponential curve, while 1 uses a gaussian curve.

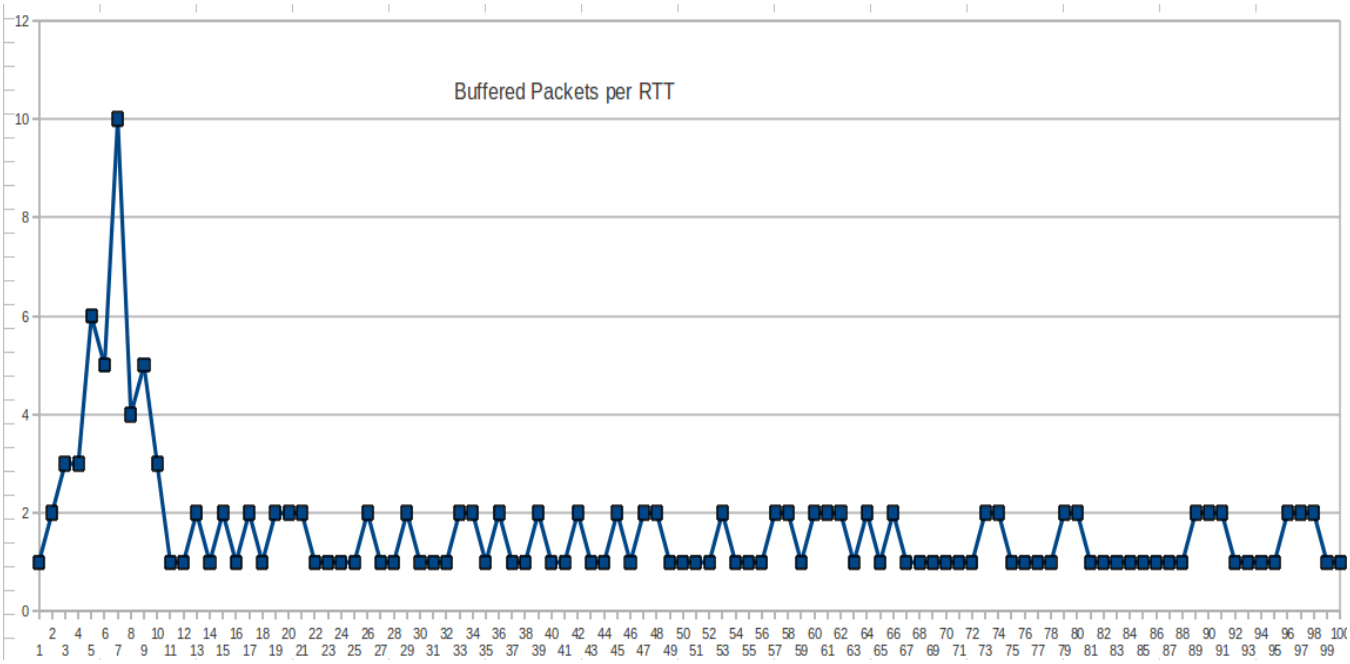
To show more verbose output, edit the Router.java file and replace the verbose_output boolean
with the value true to show more descriptively what occurs with the packet reordering stage.

4. Results and Discussion

To simulate how the TCP simulator handles packet reordering, the program is ran with the parameters *100* for RTT's, *2* for maximum packet delay, and *1* to get Gaussian randomized numbers. The following charts show what the output of the TCP Simulator is:



Graph 4.1: TCP Tahoe with Packet Reordering Simulation

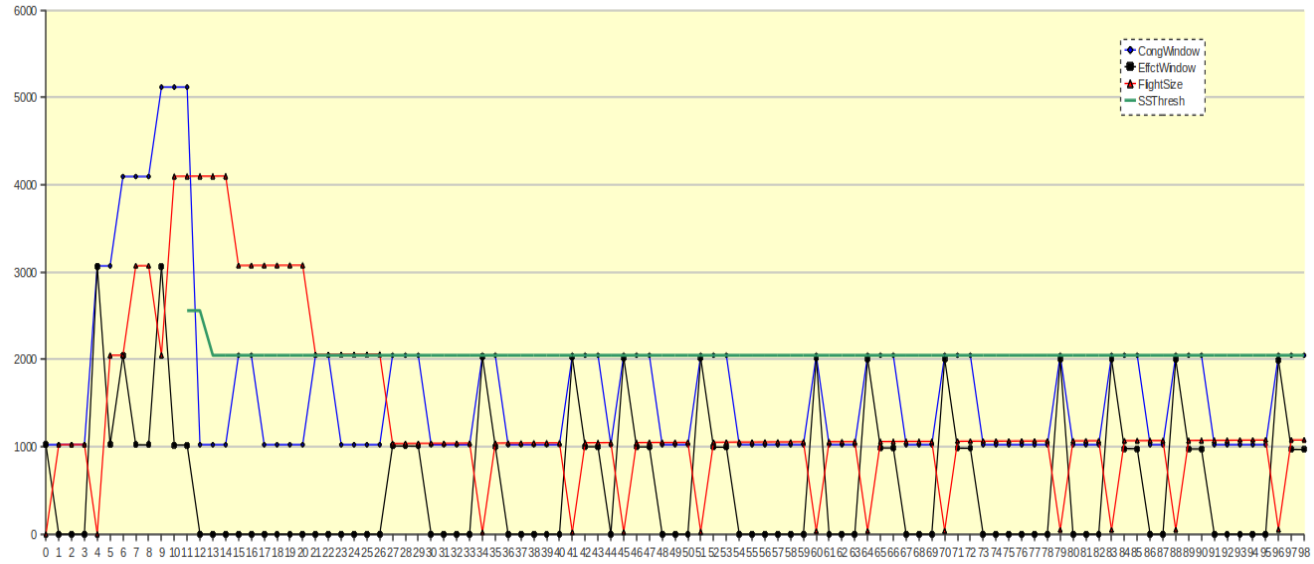


Graph 4.2: Packets Remaining in the Buffer with MaxDelay = 2

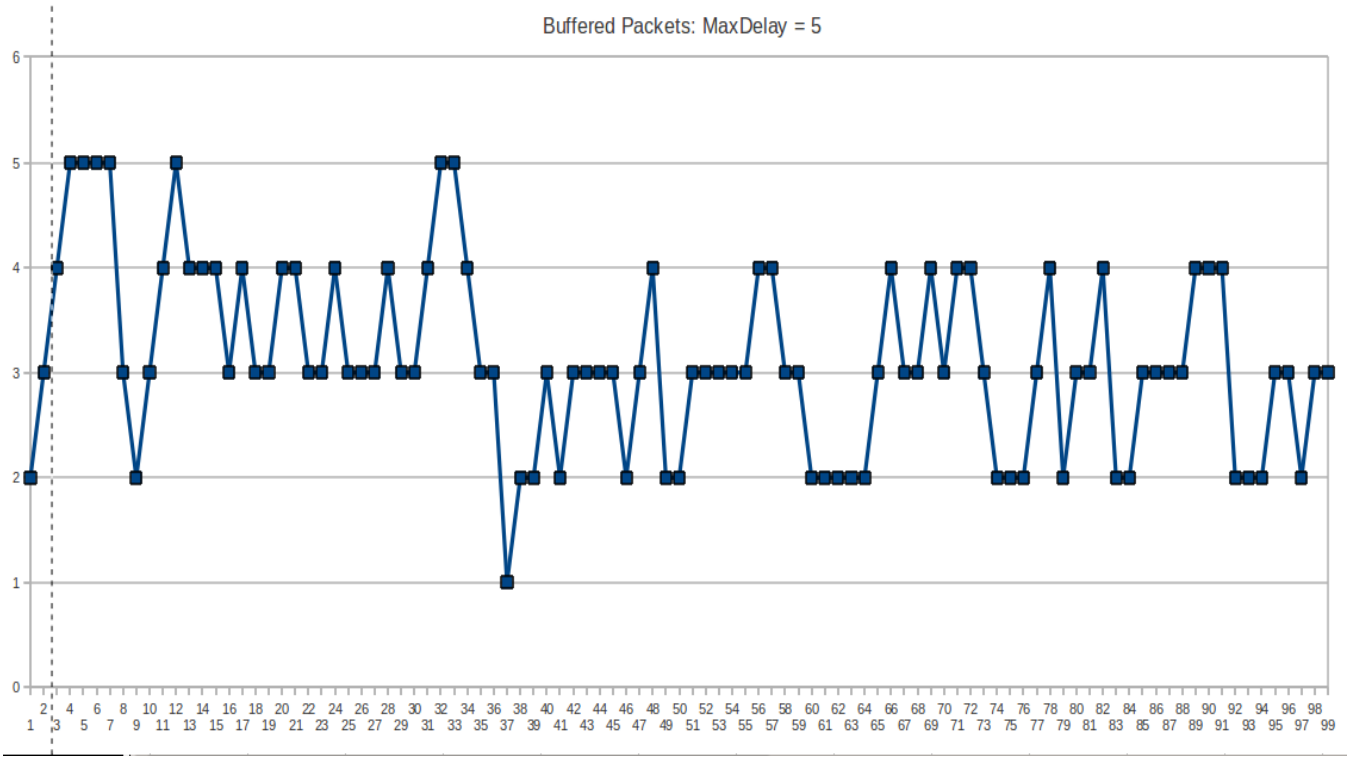
Upon observation of the graphs, we see that when TCP is in slow start mode, the window size begins to grow very large which makes more packets sent per RTT. Because more packets are delayed, this is why there is an increase in buffered packets per RTT in the beginning of the simulation. At one point however, the *flight size* begins to grow very large (the number of packets that have gone unacknowledged) due to the larger amount of reordered packets, and by the formula and process described above this makes the TCP sender go into Congestion Avoidance, which can be oberved by both of the above graphs. This allows a relatively low amount of buffered packets, and allows the values to balance out to prevent timeouts on data transfer.

One thing that is interesting about the above two graphs is that you can almost overlay the line for the EffectiveWindow with the buffer size graph. This allows you to determine what transfer mode TCP Tahoe is in just by observing the packets remaining in the buffer. The sender utilization in this case averages about to 5%, which is less than the simulator code from the website running unmodified. This makes sense because the reordered packets should cause delay in information being sent through the TCP stream, therefore lowering the server utilization. Another glance at the program output shows that the connection sends a lot of 1-byte “keep-alive” packets to avoid timeouts with our simulator.

Another experiment we will run is to determine what happens when we increase MaxDelay. In theory, this should lower server utilization and it should take a longer amount of time for each packet to reach the sender. Below are the results of the trial ran for this experiment:



Graph 4.3: TCP Tahoe Simulation with MaxDelay = 5



Graph 4.4: Packets Remaining in the Buffer with MaxDelay = 5

This time it takes a longer amount of time for the simulator to enter congestion avoidance, however you can observe that overtime the TCP values tend to normalize to the same as the first trial. However, there are a larger amount of packets that are buffered as depicted in the

second graph. This gives a smaller utilization rate as predicted because it takes a longer amount of time for data to be sent from the sender to the receiver.
The server utilization determined for this trial is 2%.

This leads to the conclusion that packet reordering in general is unfavorable, is it causes longer transmission delays, TCP begins to enter states where less throughput can be achieved, therefore slowing down the whole transmission process. There have been attempts to work around the inevitable packet reordering that tends to happen when sending over line medium, such as the document cited in the references that defines a new standard of TCP called TCP-PR, developed by students at the University of Carolina and the University of Delaware. This TCP version uses a similar method mentioned in the paper where packet timings are calculated and sent to the router, these timings can be used to rearrange packets in their correct order at the receiving end therefore allowing a consistent stream of data to be sent. This could be particularly useful over a noisy line medium, such as Wi-FI, where the likelihood of packet reordering and retransmission is much higher than over standard ethernet cable.

5. References

- Marsic, Ivan. *Computer Networks, Performance and Quality of Service*. Rutgers University, Apr 28 2010 edition. http://www.ece.rutgers.edu/~marsic/books/QoS/book-QoS_marsic.pdf

- M. Allman, V. Paxson, W. Stevens. *TCP Congestion Control*. Network Working Group, 1999. http://www.apps.ietf.org/rfc/rfc2581.html

- W. Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. 2001. http://www.apps.ietf.org/rfc/rfc2001.html

- Various Authors, *TCP-PR, TCP for Persistent Packet Reordering*. 2003. http://citeseerx.ist.psu.edu/viewdoc/download%3Fdoi%3D10.1.1.77.2354%26rep%3Drep1%26type%3Dpdf&rct=j&q=packet%20reordering%20tcp&ei=I8gTTbXFDoSglAfnupRJ&usg=AFQjCNHKjKurxHdKft44MNvW-8PkX2YKdw&sig2=n9o-09Tzwi04LWodSv9UOg