Lab 2 Report

Nathan Davis

Feb 20, 2016

1 Reliable Transport

1.1 Network Setup

For the networks in this paper, we set up two nodes. We used the same structure for each part and varied the queue length and loss rates. The configuration looks like this:

```
# n1 -- n2
#
n1 n2
n2 n1

# link configuration
n1 n2 1Mbps 1000ms
n2 n1 1Mbps 1000ms
```

1.2 Sender Side TCP Implementation

We used the transfer.py file from bene/examples as a base point to test my tcp implementation.

Beginning with the send method of TCP we put all the data it received into the send buffer.

```
    def send(self,data):
        ''' Send data on the connection. Called by the application. This
            code uses the SendBuffer to send incrementally '''
        self.send_buffer.put(data)
        self.queue_packets()
```

We created a custom method called queue_packets, used only internally, that sends as many packets as possible, given the constraints of data availability and window size.

```
    def queue_packets(self):
        ''' Send the next set of packets if there's any to send. Limit based
            on window size and the number of outstanding packets '''
        while True:
            if self.send_buffer.available() <= 0:
                break
            if self.send_buffer.outstanding() + self.mss > self.window:
                break
            packet_data = self.send_buffer.get(self.mss)
            self.send_packet(packet_data[0], packet_data[1])
```

The queue_packets method relies on the send_packet method given in the original TCP implementation. We did not change that method at all. However, it must be noted that the send_packet method sets a timer for the first packet it receives. This timer triggers a retransmission after the alloted timeout. We'll discuss later how that timeout is determined.

The retransmission method gets the next data to send, based on what's been ACKed, and sends that in a packet. We set the timeout to double so that it won't cause network congestion if there's major loss somewhere along the line. We also set another timeout in case this retransmission is lost.

```
1     def retransmit(self,event):
2         ''' Retransmit data. '''
3         self.trace("%s (%d) retransmission timer fired" % (self.node.hostname,self.source_add
4         packet_data = self.send_buffer.resend(self.mss)
5         self.send_packet(packet_data[0], packet_data[1])
6         self.timeout = self.timeout * 2
7         self.trace("doubled the timeout to %f" % self.timeout)
8         self.timer = Sim.scheduler.add(delay=self.timeout, event='retransmit', handler=self.re
```

We expect the receiver to send ACKs and we handle those using the method below. We use the slide method on the send_buffer to mark that data as "received" and we send the next set of packets using the queue_packets method. We also set the sequence so that we can keep track of what the receiver has confirmed.

The handle_ack method also handles RTT calculations which we'll discuss later.

```
1     def handle_ack(self,packet):
2         ''' Handle an incoming ACK. '''
3         ...
4         self.sequence = packet.ack_number
5         self.send_buffer.slide(packet.ack_number)
6         self.cancel_timer()
7         if self.send_buffer.outstanding() or self.send_buffer.available():
8             self.timer = Sim.scheduler.add(delay=self.timeout, event='retransmit', handler=se
9             self.queue_packets()
```

1.3 Receiver Side TCP Implementation

The receiver uses the ReceiveBuffer to keep track of data in order. When it receives a packet, it puts the packet in the receive buffer and sends an ACK for that data. It also delivers any data that's ready to the application.

```
1     def handle_data(self,packet):
2         ''' Handle incoming data. This code currently gives all data to
3             the application, regardless of whether it is in order, and sends
4             an ACK.'''
5         self.trace("%s (%d) received TCP segment from %d for %d" % (self.node.hostname,packet
6         self.receive_buffer.put(packet.body, packet.sequence)
7         data = self.receive_buffer.get()
8         self.ack = len(data[0]) + data[1]
9         self.app.receive_data(data[0])
```

```
10          self.send_ack(packet.sent_time)
```

The method that sends an ACK is a simple packet with the ACK attribute set.

```
1       def send_ack(self, packet_sent_time=0):
2           ''' Send an ack. '''
3           packet = TCPPacket(source_address=self.source_address,
4                              source_port=self.source_port,
5                              destination_address=self.destination_address,
6                              destination_port=self.destination_port,
7                              sequence=self.sequence,ack_number=self.ack,
8                              sent_time=packet_sent_time)
9           # send the packet
10          self.trace("%s (%d) sending TCP ACK to %d for %d" % (self.node.hostname,self.source_ac
11          self.transport.send_packet(packet)
```

1.4 Timing Calculations

These calculations are implemented inside the handle_ack method.

Whenever a packet is sent from the sender side, it's given a timestamp. That timestamp is transferred to the ACK on the receiver side and sent back. By comparing that timestamp to the current time, we can get a sample RTT.

```
1       # Calculate Sample RTT
2       sample_rtt = Sim.scheduler.current_time() - packet.sent_time
3       self.trace("sample round trip time %f" % sample_rtt)
```

We used the sample RTT to calculate the estimated RTT. If the estimated RTT has not been set yet, we set the estimated RTT to the sample RTT.

The calculation uses exponential backoff to place a heavier weight on more recent samples. The way it is calculated, it takes a percentage of its current value and adds in the new average from the sample RTT. This means that over time, samples that were taken earlier are given less weight than samples taken more recently.

The calculation uses alpha as the control for how much each portion should be weighted.

```
1       # Calculate the new estimated RTT
2       alpha = 0.125
3       if not self.estimated_rtt:
4           self.estimated_rtt = sample_rtt
5       else:
6           self.estimated_rtt = (1 - alpha) * self.estimated_rtt + alpha * sample_rtt
```

Then we use the sample RTT and the estimated RTT to calculate a deviation. If the deviation has not been set yet, we initialize it to half of the estimated RTT.

```
    # Calculate the deviation of the sample RTT
    beta = 0.25
    if not self.deviation_rtt:
        self.deviation_rtt = self.estimated_rtt/2
    else:
        self.deviation_rtt = (1 - beta) * self.deviation_rtt + beta * abs(sample_rtt - self.es
```

Finally, we use that deviation of the RTT to calculate the timeout. That timeout is used when setting a retransmission timeout. As the RTT increases, so does the timeout. This gives packets sufficient time to reach the destination and for an ACK to come back and helps to avoid overfrequent retransmissions.

```
    # Calculate the Retransmission Timeout (RTO)
    self.timeout = self.estimated_rtt + 4 * self.deviation_rtt
    self.trace("changed the timeout to %f" % self.timeout)
```

1.5 Testing

For the testing we used the network described above. We set the window size to 3000 bytes and tested with loss rates of 0%, 10%, 20%, and 50%.

We run these using the following command. The -f flag specifies which file to send, the -w flag specifies how large the window should be, and the -l flag specifies the loss.

```
    python transfer.py -f test.txt -w 3000 -l 0.0
```

The results for the test using 0% loss were

```
    You imported the correct tcppacket
    You've imported the right tcp file
    0 n1 (1) sending TCP segment to 2 for 0
    0 n1 (1) sending TCP segment to 2 for 1000
    ...
    0.0732 n2 (2) sending TCP ACK to 1 for 10000
    0.0832 sample round trip time 0.020800
    0.0832 changed the timeout to 0.024803
    Total time: 0.083200 seconds

    File transfer correct!
```

Each time a packet successfully completes a trip, we see the output notifying us that a sample round trip time has been taken and that the timeout has changed. This shows us that the timeout is always slightly more than tha round trip time. In a real-world implementation we would constrain the timeout to a minimum of 1 second, however it's usefull to see that in certain circumstances, the timeout may be much smaller to improve performance.

The other tests had the following results. These show us that loss rate is not always determinate of speed. Notice how the test at 20% loss was actually faster than the test at 10%. Running these tests multiple times gave different speeds, with the exception of 0% loss.

```
Loss Rate    Transmission Time
0.00         0.083200 seconds
0.10         0.142825 seconds
0.20         0.085600 seconds
0.50         240.983621 seconds
```

The trend is clear however that the higher the loss, the greater the time it takes to transmit in general.

Running these same tests on a slightly larger file with a larger window, we see the same trend.

```
python transfer.py -f internet-architecture.pdf -w 3000 -l 0.00
```

```
Loss Rate    Transmission Time
0.00         3.578016 seconds
0.10         7.461133 seconds
0.20         10.506584 seconds
0.50         127486.273711 seconds
```

2 Experiments

2.1 Changing Window Size

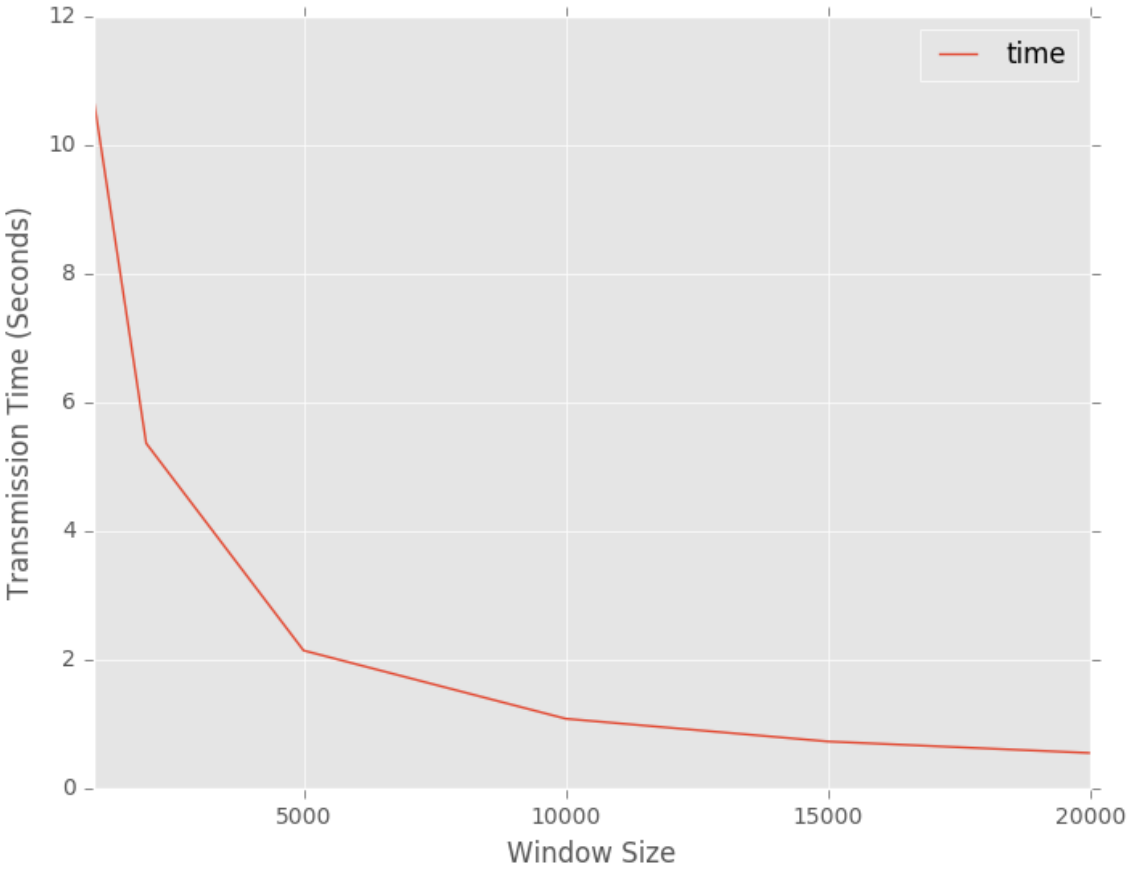
For this portion, we sent the larger test file usind different window sizes. The loss rate was kept constant at 0% and we look at the throughput and an average of the queuing delay.

We used the command line to run the expirement with different window sizes. The -f flag was used to designate a large pdf file. The -q flag was used to toggle into the network with a queue limit of 100. The -l flag is used to ensure the loss is 0% and the -w flag is used to change the window sizes. We used window sizes of 1000, 2000, 5000, 10000, 15000, and 20000.

```
python transfer.py -f internet-architecture.pdf -q -l 0.0 -w 1000
```

2.2 Overall Time

The first thing we observed is that the experiments with larger window sizes had a much smaller overall transmission time than those with smaller window sizes. The graph below illustrates how the total time decreases quickly in the beginning and then levels off as it approaches zero.

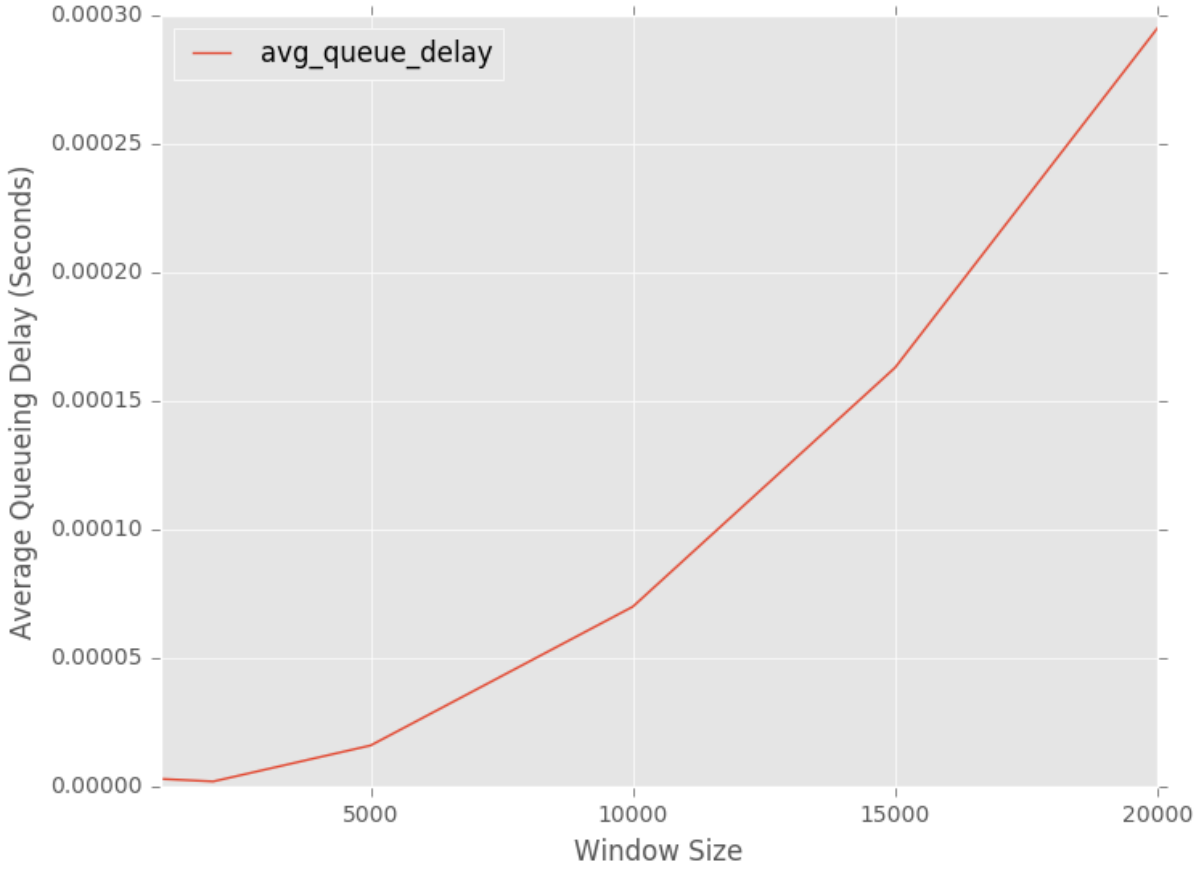


2.3 Average Queueuing Delay

The average queueing delay increased exponentially as the window size increased. This is because the sending side puts as many things in the queue as possible. Some of the packets may have even been dropped due to the lack of space in the queue. That queueing delay wouldn't even be accounted for in the graph below.

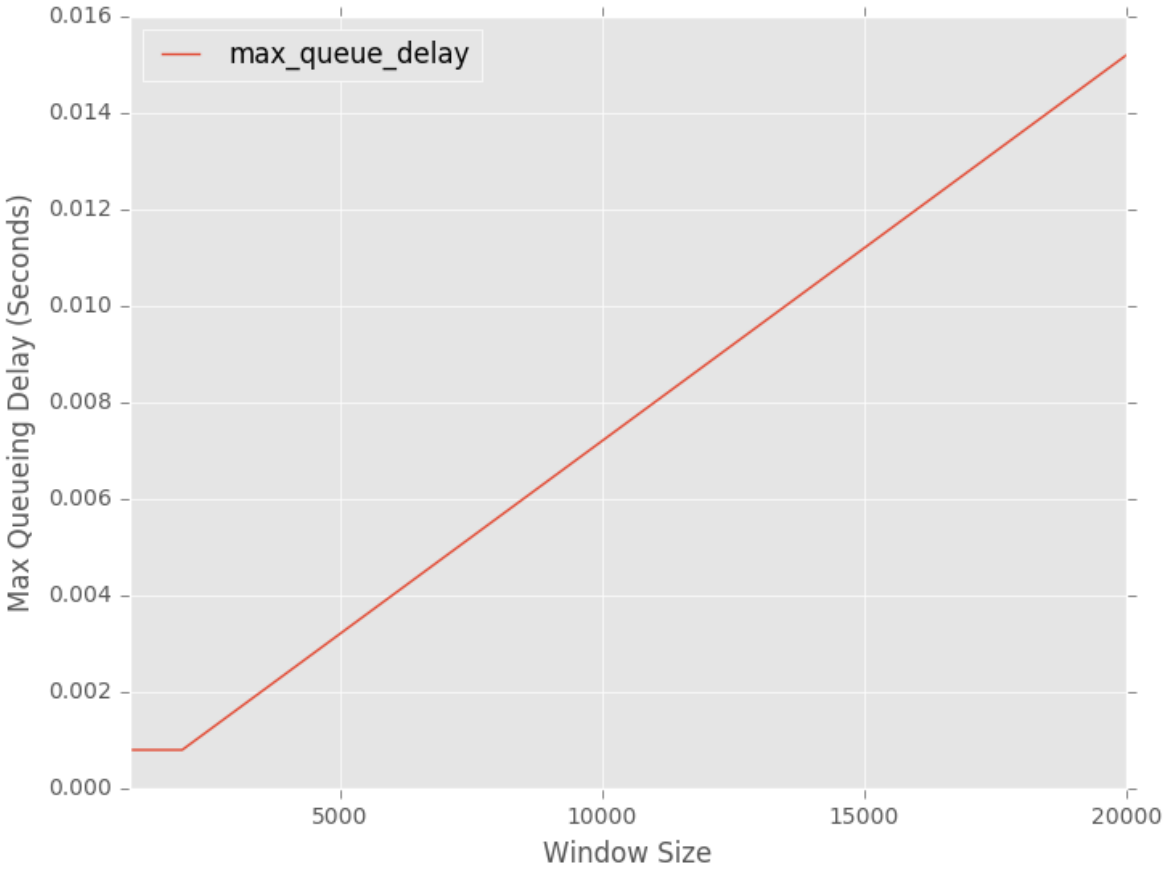
The graph below represents an average queueing delay for all the packets that were received by the receiver. It does not take into account the queueing delay for ACKs or lost packets.

Because the queuing delay increases exponentially, we could expect it to offset the window at some point and cause the overall time to increase. We would definitely expect to see higher values if more than one connection was running at a time. In a real network, multiple machines are sending and receiving and so routers can become congested. However, in this simulation we have only one bi-directional connection.



2.4 Max Queueuing Delay

Another measure we found interesting was the max queueing delay. This shows a linear increase and is most likely caused by the first set of packets sent. They would all be placed in the send queue at once and so the last of those packets would be likely to have the maximum queueing delay.



2.5 Throughput

Finally, the throughput shows a linear increase. In general, this would point to a trend that larger window sizes tend to be faster. The fact that the throughput growth appears to be lindear was a little surprising as the size of the file is the same for every run and the time is shown to decrease rapidly at first and then taper off. We expected the throughput to increase rapidly at first and then taper off as it approaches a maximum.