Lab: Reliable Transport

Spencer Gardner

February 18, 2015

1 Introduction

In this lab, we present a Python implementation of Reliable Transport using Transport Control Protocol (TCP). We then discuss a pair of tests used to verify that the implementation performs as expected. Thereafter we present experiments designed to observe the behavior of the implementation.

2 Reliable Transport

To start, let's take a look at the implementation of Reliable Transport. Two main components, a sender and a receiver, are the basis for the protocol. The sender wishes to transmit data to the receiver reliably, so as to be sure that no data will be lost. To facilitate this transfer and reliability, the sender and receiver agree upon a protocol.

2.1 Sender

The vital functions belonging to the Sender are as follows:

2.1.1 send

This method takes each piece of data passed to it by the application layer, buffers it, starts a timer, and then calls send_any_available to handle the logic for actually sending the packet.

2.1.2 send_any_available

This method handles the logic for sending of packets. It consists of a loop that stops only when there is no available data or when the outstanding number of packets is greater than the send window. Inside the loop, we calculate the amount of data to send and send a packet.

2.1.3 send_packet

This method handles the actual creation and sending on the wire of the packet. After the packet is sent, if a timer has not been created, we create one and set it to call the retransmit method when it fires.

2.1.4 handle_ack

This method is called when the Receiver sends an ACK back. We set the sequence number, slide the send buffer, and then call send_any_available. If the send buffer has 0 packets remaining, the timer is canceled.

2.1.5 retransmit

This method is called when the retransmit timer fires. We also calculate new RTO/Timeout values here based on the old timeout value.

2.2 Receiver

The vital functions belonging to the Receiver are as follows:

2.2.1 handle_data

This is the method that receives the data from the sender. We add the data to the receive buffer, pass the in order data to the application, calculate an ACK number to send back, and then call send_ack

2.2.2 send_ack

This method actually sends an ACK back to the sender. We create a TCP packet for the ACK and send it based on the parameters calculated in handle_data.

3 Tests

We test sending two files, test.txt and internet-architecture.pdf over an implementation of TCP with a fixed RTO timer of 1 second. We tested the average time it took to send these two files of networks with 0%, 10%, 20%, and 50% loss rates. Since the values for each file transfer varied based on the random loss characteristics, we averaged each transmission time over several iterations. Since the timer is fixed, the results in the 2 graphs below are similar.

3.0.3 Test 1

The first test consisted of a 10mbps link with 10ms propagation time. We sent a small file, test.txt, over the wire with the aforementioned loss rates. The window size here was 3000B.

3.0.4 Test 2

The second test consisted of a 10mbps link with 10ms propagation time as well. We sent a larger file, internet-architecture.pdf, over the wire at the same loss rates. The window size in this case was 10,000B.

4 Dynamic Retransmission Timer

A dynamic retransmission timer was implemented to make TCP respond more efficiently to packet loss. Instead of naively sending packets, we capture

Below, we see the main calculation for setting self.timeout. This code is placed within the handle_ack method, such that whenever an ACK is received from the receiver, the timeout can be dynamically set. We first calculate the sample round trip time (sample_rtt) of the lat packet sent. Then, we calculate an estimated_rtt based on the previous estimate. Thereafter, we calculate the deviance from an the sampled RTT. We then set a new timeout value based on the formula in RFC 2988 and limit it to between .2 and 120 seconds.

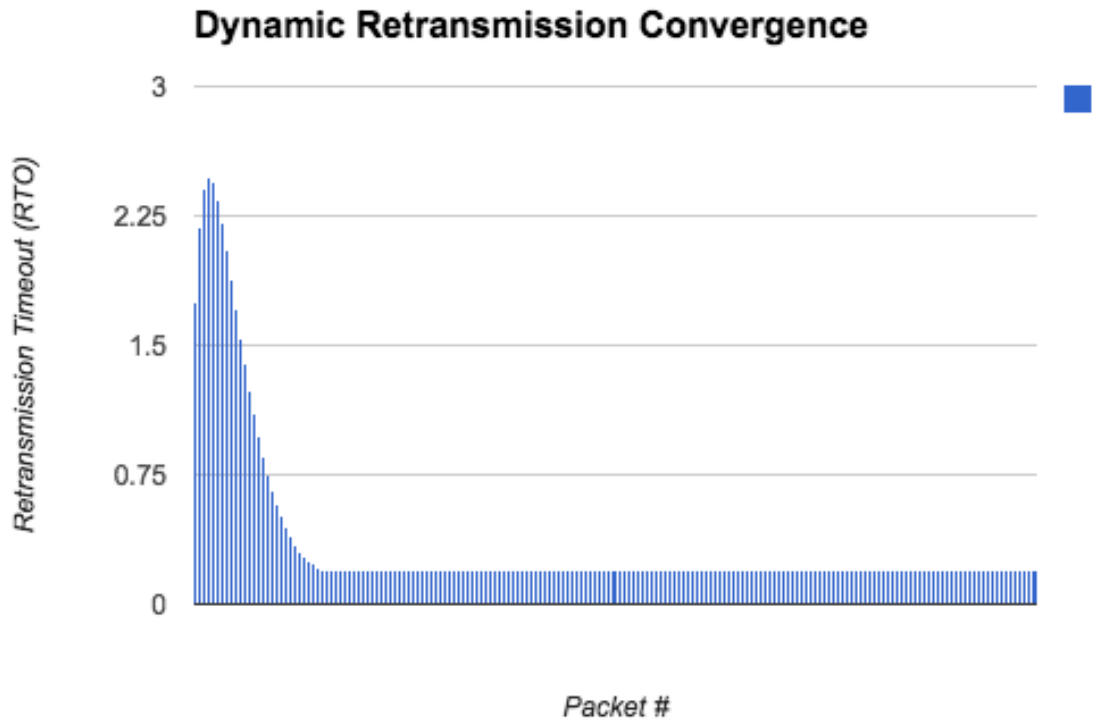
```
# calculate sample_rtt for last packet
currTime = Sim.scheduler.current_time()
prevTime = self.round_trip_map[packet.ident]
sample_rtt = currTime - prevTime

# calculate estimated rtt and deviation from rtt
self.estimated_rtt = (1-.125) * self.estimated_rtt + (.125 * sample_rtt)
self.dev_rtt = (1-.25) * self.dev_rtt + .25*abs(sample_rtt - self.estimated_rtt)

# set new RTO/timeout. limit to range of [.2,60]
self.timeout = self.estimated_rtt + 4 * self.dev_rtt
self.timeout = max(0.2, min(self.timeout, 120))
```

The below code is called every time the retransmit method is called. It doubles the timeout up to 120 seconds from a minimum value of .2 seconds.

```
self.timeout = max(.2, min(2*self.timeout, 120))
```

4.1 Dynamic Retransmission Results

Below we see a graph of the dynamic retransmission in action. As the the transmission of a file starts, we see that the timeout is fairly high. However, once the ACKs begin to be received by the sender, it realizes it can lower the transmission timeout value. After a certain period of time about 20% through the file transmission, the timer reaches a convergence point of about .25 seconds.



4.1.1 Debug Output

Below we see a listing of debug output detailing some of the dynamic retransmissino timer's behavior. We see that, for example, a new timeout will be set and then recalculated after ACK's are received.

```
0.0624 new timeout: 2.406250
0.0732 n2 (2) received TCP segment from 1 for 3000
0.0732 application got 1000 bytes
0.0732 n2 (2) sending TCP ACK to 1 for 4000
0.0832 n1 (1) sending TCP segment to 2 for 4000
0.0832 new timeout: 2.474609
0.094 n2 (2) received TCP segment from 1 for 4000
0.094 application got 1000 bytes
0.094 n2 (2) sending TCP ACK to 1 for 5000
```

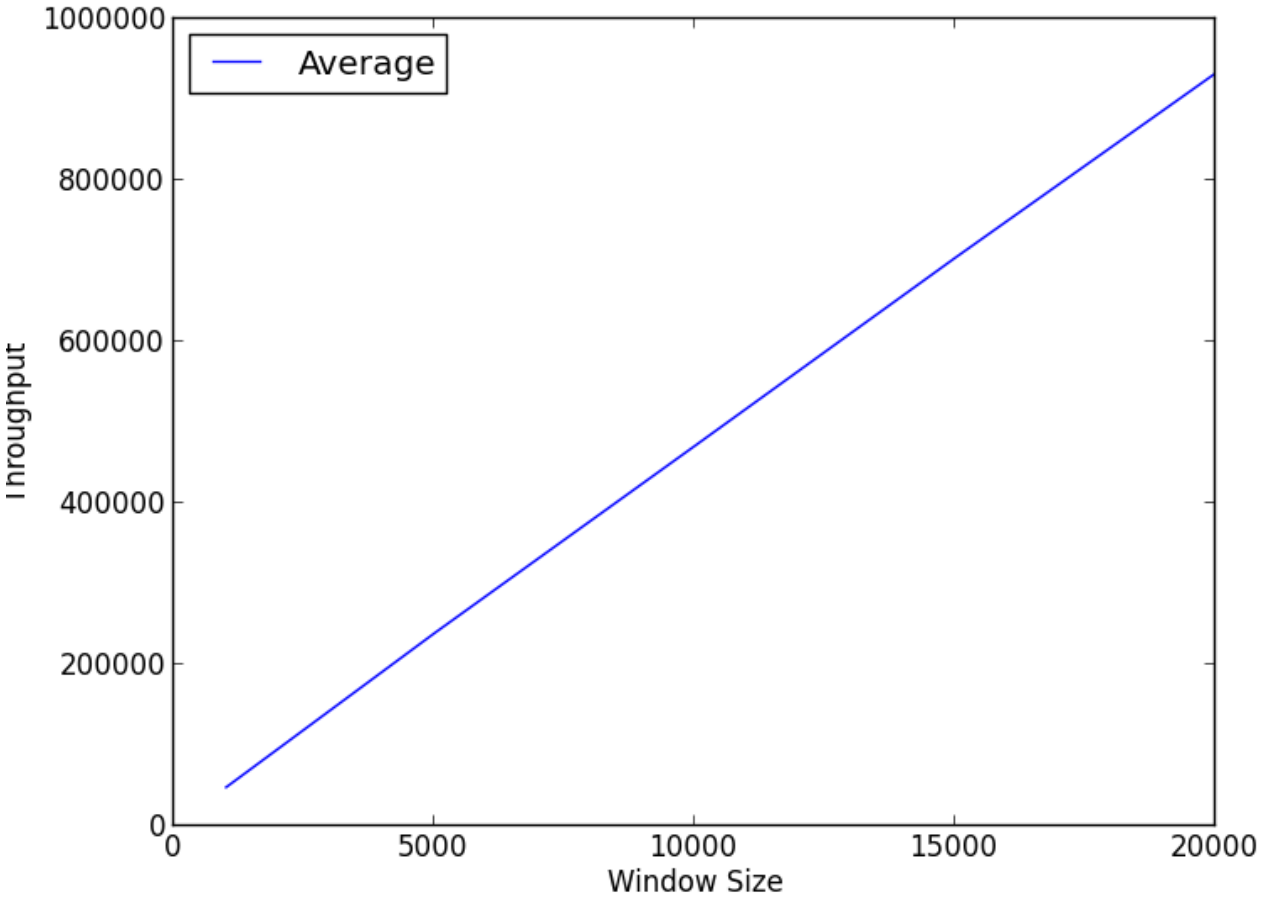
```
0.104 n1 (1) sending TCP segment to 2 for 5000
0.104 new timeout: 2.442139
...
...
...
7.6271556455 n1 (1) retransmission timer fired
7.6271556455 new timeout: 0.400000
7.6271556455 n1 (1) sending TCP segment to 2 for 49000
7.6379556455 n2 (2) received TCP segment from 1 for 49000
7.6379556455 application got 1000 bytes
7.6379556455 n2 (2) sending TCP ACK to 1 for 50000
8.0271556455 n1 (1) retransmission timer fired
8.0271556455 new timeout: 0.800000
```

5 Experiments

Lastly, we perform two experiments to observe the behavior of our implementation. We will discuss an experiment measuring throughput with increasing window sizes. Then, we'll discuss average queueing delay with increasing window sizes.
In both experiments, we use a link at 10mbps with 10ms propagation delay and a queue size of 100 packets with 0% loss. Then, we use window sizes of 1000, 2000, 5000, 10000, 15000, and 20000 bytes.

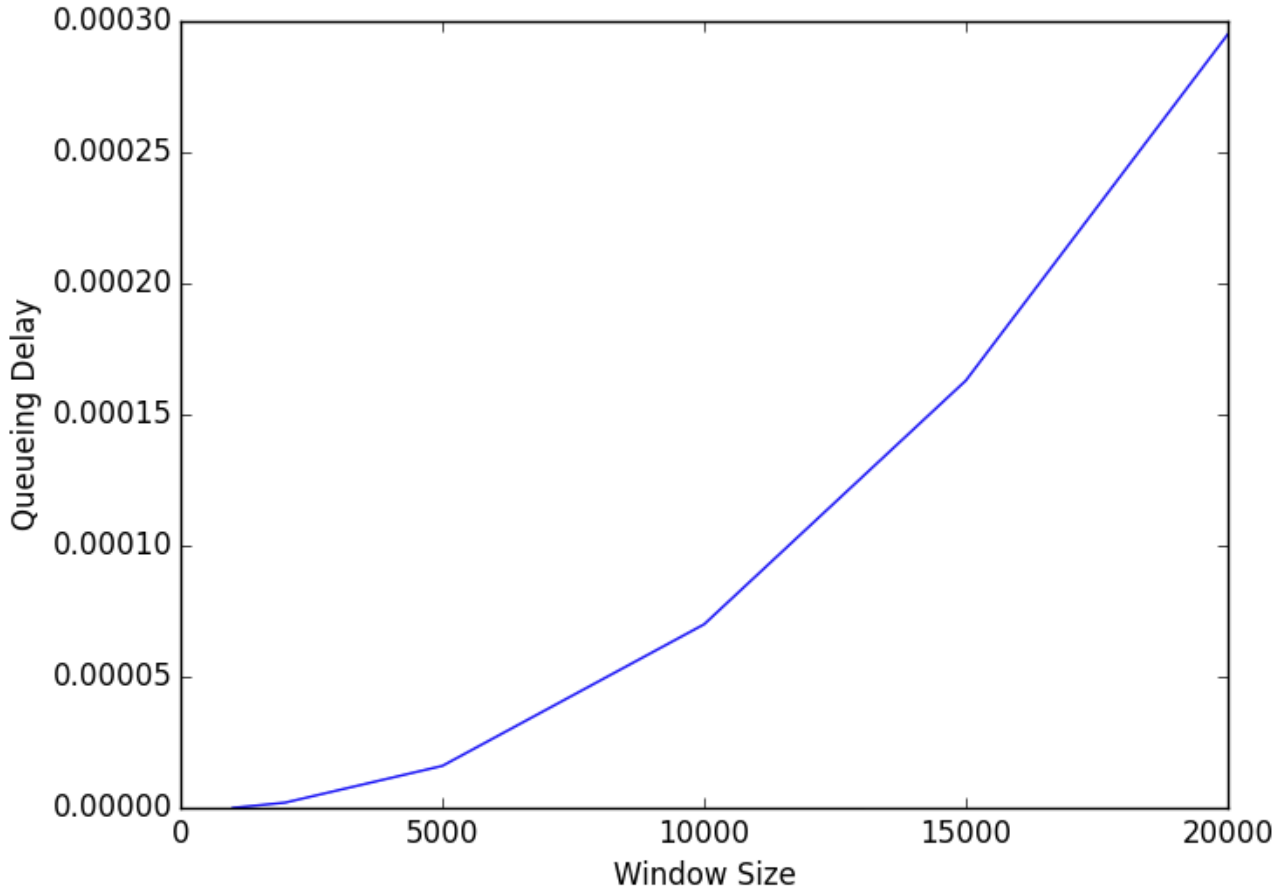
5.1 Throughput

In this experiment we are measuring throughput through the network. This is measured by taking the total number of bits delivered over the network and dividing it by the time it took to do so. We can see that throughput increases linearly as window size increases. This is to be expected, since there is no loss in the network.



5.2 Queueing Delay

In this experiment, we measure queueing delay as window size increases. We see that the delay increases somewhat linearly as window size is increased. However, the scale for increase is quite small. As we vary window size by a couple of orders of magnitude, queue size is increasing by only .0005 seconds. While this behavior could lead to large queuing delays given arbitrarily large window sizes, for practical purposes in this case, it is not a significant contributor to overall delay in the network. Propagation delay, for example, could contribute much more significantly to overall delay.



6 Conclusion

From the implementation, tests, and experiments run on TCP, we see that by varying window size, queue size, and timers, we can coax TCP to behave differently under varying circumstances. This will be useful in further study to understand TCP's behavior in order to use it to its full potential.