CS460 Lab 2 Report: Reliable Transport

Michael K. Eagar

February 23rd, 2014

1 Reliable Transport Implementation

In order to implement the reliable transport protocol I started with Dr. Zappala's Stop and Wait implementation, and built upon it. In order to guarantee reliable transport I implemented a sliding window for sending packets. I removed the *send()* method that was used to initialize the sending of packets, and replaced it with a *load_buffer* and a *window_init* method to load the data of the file being sent, and begin the sending process. I modified *transfer.py* to call these methods instead of the send method from *stopandwait.py*. My protocol is saved in the file *my_rtp.py*.

The Window initialization method calls the *send_if_possible()* method, which has been modified to check if there are any open slots in the window to send additional packets, and sends one by calling *send_one_packet()*. When a packet is sent the retransmit timer is set if it is not currently running. If the timer expires before the packet it's timing is ACKed, then that packet will be re-sent. Here is the *window_init()* method:

Listing 1: window_init()

```
                for i in range(self.window_size):
                        self.send_if_possible()

        def slide_window(self, ack_number):
```

When the packet is received at it's destination node the sequence number is checked against the one the node is expecting. If the packet sequence is greater than or equal to the sequence the node is expecting, then the packet is added to the receive buffer and the buffer is sorted. If the first packet in the buffer is the one expected, then it is written to disk and the buffer is checked for subsequent packets until a missing one is detected. An ACK is then sent for the next packet that the receiving node expects.

When an ACK is received at the sending node it is passed to the *handle_packet()* method and it is checked to see if it is within the window range of packets sent out. If it is, the sending node calls *slide_window()* to slide the window forward to send more packets. In the *slide_window()* method the number of packets outstanding is decremented depending on the ACK number received. The window then slides forward so that its start is at the next packet after the ACK received. Additional packets can then be sent depending on slots available in the window. The process of sending packets, receiving ACKS and moving the window forward repeats until the entire file has been received at the destination node. Here are the *handle_packet()* method and the *slide_window()* method:

Listing 2: handle_packet()

```
        def handle_packet(self,packet):
                # handle ACK
                if packet.ack_number > self.window_start and packet.ack_number
                   <= self.sequence:
```

```
            # this acks new data, so advance the window with slide_window()
            Sim.trace("%d received My_RTP ACK from %d for %d" % (packet.destination_address,packet.source_address,packet.ack_number))
            self.slide_window(packet.ack_number)

        # handle data
        if packet.length > 0:
            self.pkts_rcvd += 1

            Sim.trace("%d received My_RTP segment from %d for %d" % (packet.destination_address,packet.source_address,packet.sequence))
            # if the packet is the one we're expecting increment our
            # ack number and add the data to the receive buffer

            if packet.sequence >= self.ack:
                self.receive_buffer.append(packet)

                if packet.sequence == self.ack:
                    self.receive_buffer = sorted(self.receive_buffer, key=lambda TCPPacket: TCPPacket.sequence)
                    while self.receive_buffer and (self.ack == self.receive_buffer[0].sequence):
                        pkt = self.receive_buffer.pop(0)
                        self.increment_ack(pkt.sequence + pkt.length)
                        # deliver data that is in order
                        self.app.handle_packet(pkt)

            # always send an ACK
            self.send_ack()

            if packet.queueing_delay > self.pkt_q_delay_threshold:
                self.queueing_delay += packet.queueing_delay

            print "\n[Average Queuing Delay so far:", str(self.queueing_delay / self.pkts_rcvd) + "]"
            print "\n[Total Queueing Delay so far:", str(self.queueing_delay) + "]\n"
```

Listing 3: slide_window()

```
    def slide_window(self, ack_number):
        packets_acked = int(math.ceil((ack_number - self.window_start)/self.mss))
        self.packets_outstanding -= packets_acked
        self.window_start = ack_number
        for i in range(self.window_size - self.packets_outstanding):
            self.send_if_possible()
        self.cancel_timer()
        if ack_number < len(self.send_buffer):
            self.timer = Sim.scheduler.add(delay=self.timeout, event='retransmit', handler=self.retransmit)
        else:
                self.timer_set = False
```

2 Testing

In order to test my reliable transport protocol implementation I used Dr. Zappala's *transfer.py* file as the basis for setting up my tests. I changed transfer.py to accept additional command line parameters, *Queue size* and *Window size*. These parameters are used to set up the simulated network and the window size for the protocol. The links between the two nodes are set with a bandwidth of 10 Mbps, and a propagation delay of 10 ms.

I then wrote a python script to run the eight tests in two sets by passing different command line parameters to *transfer.py*:

1. Set 1 - Window size of 3000 bytes (3 packets), transfer *test.txt*
   (a) Loss rate of 0%
   (b) Loss rate of 10%
   (c) Loss rate of 20%
   (d) Loss rate of 50%
2. Set 2 - Window size of 10000 bytes (10 packets), transfer *internet-architecture.pdf*
   (a) Loss rate of 0%
   (b) Loss rate of 10%
   (c) Loss rate of 20%
   (d) Loss rate of 50%

The python script is *run_lab2_testing.py*, and must be run in the **lab2** directory. I saved the output results from my tests in the file *testing.txt* which is in the **lab2/data** directory.

My reliable transport protocol passed all of the tests, successfully transferring the specified file each time.

3 Experiments

In order to run the experiments as required in the lab specifications, I used the same network configuration as used in the testing section, but set the *queue size* to be 0%. I wrote a python script to run the experiments by once again passing command line arguments to *transfer.py*, varying the window size at 1000, 2000, 5000, 10000, 15000, and 20000 bytes. These window sizes correspond with 1, 2, 5, 10, 15, and 20 packets respectively. The script is *run_lab2_experiments.py* and must be run in the **lab2** directory. The resulting output from the experiments is in the file *experiments* in the **lab2/data** directory.

Figure 1 shows the average queuing delay per packet as window size is increased. It shows that as window size increases, so does the average queuing delay. It seems to follow an exponential curve, but more data would be required in order to verify this.

Figure 1 - Average Queuing Delay vs Window Size

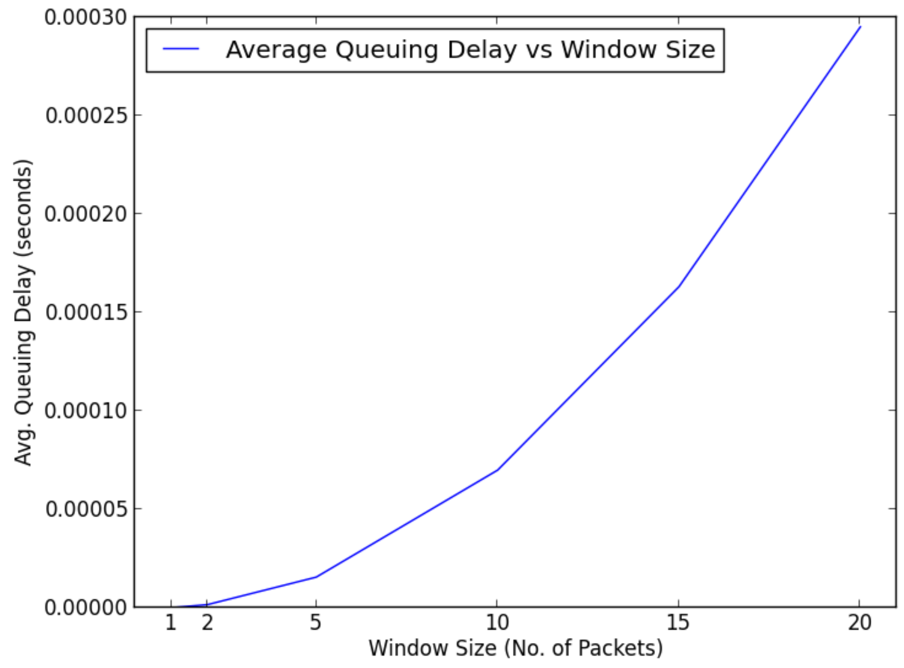
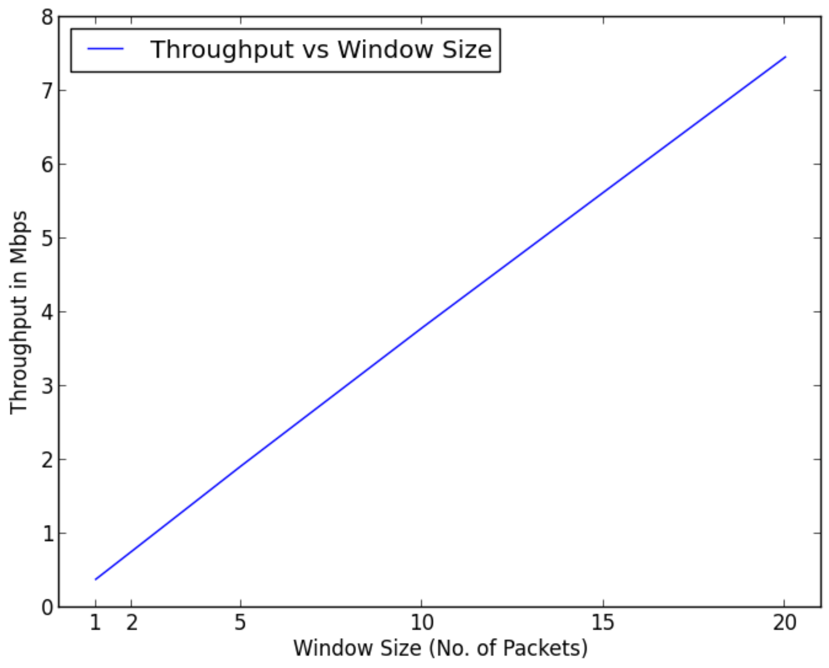


Figure 2 below shows the throughput of the link as the window size increases. For the window sizes used in the experiment, throughput increases at a linear rate. This was interesting to me, as I had expected the throughput to cap off. It could be that larger window sizes would need to be tested in order to see the that in the results.

Figure 2 - Throughput vs Window Size


Throughput vs Window Size
Throughput in Mbps
Window Size (No. of Packets)