Lab 2 Report
Reliable Transport

Kyle Pontius

1 Introduction to TCP Reliability

One of TCP's most powerful features is its ability to send data reliably over a connection, even with loss. This is a critical feature due to the packet-centric nature of our internet architecture today. There have been several innovations that contributed to making this function properly including congestion control windows, dynamic retransmission timers, and cumulative acknowledgements. There are a few areas of interest that we're going to touch on this report. First, my implementation of TCP Reliability and the related tests. Second, my fixed and dynamic timer systems. Third, the experiment results and their explanation.

2 My TCP Implementation

Taking my network implementation from simply transferring a file without loss to functioning with reliability required several steps and new features. First, the congestion window was set to a fixed size. The size of the window represents the number of packets that are allowed to be outstanding. This feature is important because it affects the size of the queues on the link and also the rate at which data gets sent. Second, a timer is used to determine if the packet was lost. There were two timers used in this lab, the first was a fixed timer that was set at one second while the other was a dynamic timer, the behavior of which I'll discuss later. The timer in general is significant because it enables TCP to recover after a packet is lost.

Now to describe my specific implementation of reliability, we'll attempt to follow the data flow chronologically. First, the data is read into the program in 1000 byte increments, or less, and passed to TCP. This is the MSS, or Maximum Segment Size. TCP then takes the data and attempts to send the packet if the outstanding number of bytes is less than or equal to the window size. When this occurs, the timer is reset. If it can be sent, the packet is immediately constructed and sent. Otherwise, the data remains in TCP's send buffer until a later point. At this point, two primary behaviors will occur, the receiver will send a responding ACK to notify the sender that a given packet has been properly received. The ACK itself is cumulative, or in other words, the ACK number represents the next packet sequence number anticipated by the receiver. When an ACK is received with new data, the timer is restarted. Now the program evaluates the number of outstanding packets and if the number outstanding is less than the window size, additional packets are sent until the congestion window size is matched. Notably, I also had to added a method which cancelled the timer if the last ACK had been received, otherwise the program would assume the data hadn't been received, and would continue to forward identical packets. The primary alternative to sending data when ACKs arrive back is the retransmission timer. Regardless of the amount of time, it always operates the same way when it retransmits, that is, when the timer fires it sends out the last packet requested by the receiver. The timer is also reset at this point. At the same time, it discards all packets noted as outstanding, treating them as though they were never sent. This will enable TCP to quickly resend those once the current packet has been ACK'd. Once a new ACK received, the packets pick up transmitting where the receiver's ACK requested them (again, cumulative ACKs).

3 Test Results

Now I'd like to walk through the process of executing the tests that I've set up. The tests are two separate Python scripts, one for the *internet-architecture.pdf* file and the other for *test.txt*. To run the tests, navigation to the */lab2/project/examples/* folder. Then run the scripts by passing in the loss argument. As per the spec, the *transfer-internet-achitecture.py* and *transfer-test.py* tests will run with the specified link speed (10 Mbps) and propagation delay (10 ms), as well as the recommended 10,000- and 3,000-byte congestion window sizes, respectively. If you run the tests yourself, the resulting console print out will reflect the longer times required to transmit files with greater loss rates. Additionally, when the transfer is complete, the program will print out the success message if the diff finds the sent and received files to be identical. Example terminal commands for executing the program include:

```
python transfer-internet-architecture.py --loss 0.5
python transfer-test.py --loss 0.0
```

My results when running these tests were congruent with the expected results. That is, as loss increases, the time required to transmit a file grows significantly. The timer in the first table is fixed at 1 second, while the results in the second table utilize the dynamic timer. Here are my results:

Fixed Timer (1 sec)		
Loss Rate	Test.txt: Total Transfer Time (sec)	Internet-Architecture.pdf: Total Transfer Time (sec)
0%	0.77	1.08
10%	454.05	2195.39
20%	1234.26	4296.12
50%	10774.63	24278.06

Dynamic Timer		
Loss Rate	Test.txt: Total Transfer Time (sec)	Internet-Architecture.pdf: Total Transfer Time (sec)
0%	0.77	1.08
10%	14.99	67.23
20%	65.26	200.19
50%	1663.70	6275.80

The results of these two sets of tests speak largely for themselves. When the loss rate is 0% there is no loss so the timer never expires, thus the results are identical. However, the difference quickly becomes apparent as the loss rate increases, with the dynamic timer consistently ahead in execution speed in every test. An interesting behavior to note was that since the loss is a random percentage of all packets sent, I observed that the range in possible transfer times increasingly fluctuated as you increased the loss rate. For example, the range of possible transfer times for 10% was something like 8 seconds to 20 seconds; while the range of possible transfer times for 50% loss was on the order of 1100 seconds to well over 2500 seconds. I believe this is due to the difference in TCP's behavior in various situations, such as back-to-back loss vs. distributed loss. Definitely an interesting thing to consider.

4 Dynamic Timer

I implemented the dynamic timer using the equations provided by the ***RFC 2988***. To calculate RTO, I did the following:

- RTO = 3 (until the RTT can be established)
- Starting from the first RTT measurement:
    - SRTT (smoothed RTT) = R
    - RTTVAR (RTT variation) = R/2
    - RTO = SRTT + K*RTTVAR
        - In the spec, this step computes using max(G, K*RTTVAR); however, that wasn't necessary in our implementation as granularity isn't used.
        - K = 4
- After the first RTT measurement:
    - RTTVAR = (1 – beta) * RTTVAR + beta * |SRTT – R`|
    - SRTT = (1 – alpha) * SRTT + alpha * R`
    - RTO = SRTT + K*RTTVAR

An effective Dynamic Timer backs off based on the response from the network's needs. This is different than the congestion control we build in Lab 3; it is specifically for the timer itself. The result of not backing off the timer is that TCP's timer repeatedly ends prematurely, causing duplicate packets to be sent unnecessarily. However, if the timer waits too long then the lost packets take a great deal of time to be resent and potentially valuable time is lost. Ideally, using the equations above, the timer will settle into a balanced state that is optimized for the network. In my implementation, the equations above are packaged into methods that get called when ACKs a received by the sender. The RTO is recalculated, then that value is used for subsequent timer values. Finally, in the case where retransmission occurs, the timer needs to be backed off. This is done by doing: RTT *= 2. Please note, I chose to bound the timer at 1 second as its lowest time and 1 minute as the highest time.

On a side note, while it wasn't required in the spec I wanted to test extreme conditions for *transfer-test.py*, when loss was 90% vs 99%. Both tests were done with the dynamic timer for best possible results. The 90% loss test resulted in an execution time of about 509126 seconds, or 5.89 days. The test with 99% loss executed in about 64307459 seconds, or 2.04 years! Clearly that climb to near 100% packet loss results in exponential transfer times, as anticipated.

Below is an example of the debugging output from the program that shows the dynamic timer's functionality. In this case, we're seeing both the back-off when several retransmissions occur back-to-back as well as the recalculation where the timer settles back to the proper time. While this output only consists of the validated timer values, the calculated RTO was often below 1 second, but as I noted before 1 second is the lowest allowed time. For my implementation, at or around 1 second was frequently the value that the timer converged to. Right at the end, it's clear that retransmission occurred several times, quickly backing off the timer. I've included my discussion about the difference in file transfer times between fixed and dynamic timers in the previous section with the test results. For the following results, I used the *transfer-test.py* and set the loss to 70%. Here are my results:

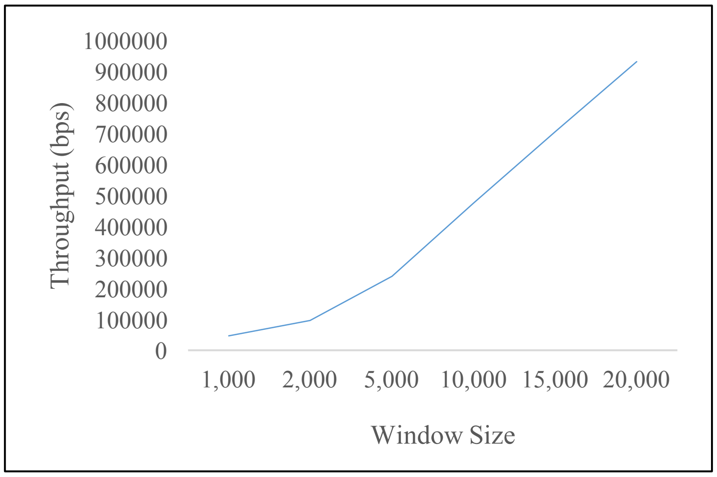
```
50370.0136 1 seconds
50371.0136 2 seconds
50373.0136 4 seconds
50377.0136 8 seconds
50385.0136 16 seconds
50385.0344 1 seconds
```

```
50386.0344 2 seconds
50386.0552 1 seconds
50386.076 1 seconds
50387.076 2 seconds
50387.0968 1 seconds
50387.1176 1 seconds
50388.1176 2 seconds
50390.1176 4 seconds
50394.1176 8 seconds
50402.1176 16 seconds
50418.1176 32 seconds
50450.1176 60 seconds
```

5 Experiment Results

The setup required for these experiments uses a network configured at 10 Mbps, 10 ms propagation delay, a queue size of 100 packets, and a loss rate of 0%. Using the *internet-architecture.pdf* file, the experiment was re-run with the congestion window sizes of: 1000, 2000, 5000, 10000, 15000, and 20000 bytes. Throughput is calculated by dividing total bits sent by total time to send the file. Average queueing delay was measured by taking the average delay for each packet received.

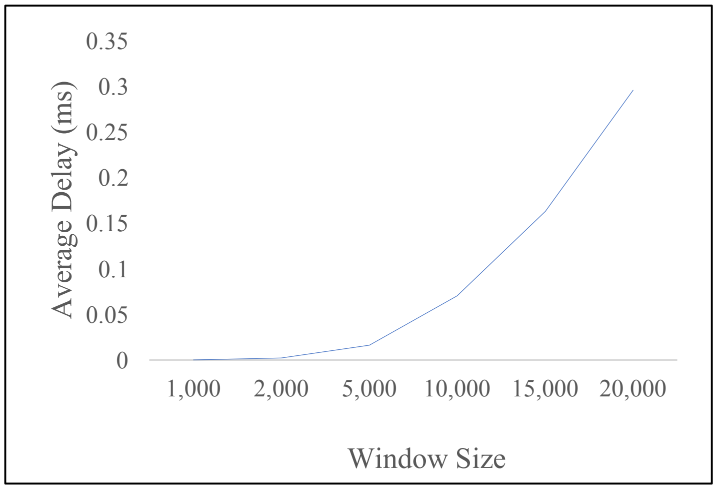
Below are my results for throughput (in bps) as a function of the window size:



These results align well with the anticipated results. Since there is no loss in the scenario, as the window size gets bigger, the program can output more data at one time. As a result, the time spent on waiting for ACKs and sending the next packet is greatly reduced. It appears that the graph itself shows linear growth as the window size increases by 5,000 bytes each time. Eventually the propagation delay would bound the ability of data to be sent as quickly as possible and this increase in throughput would level off.

(see next page)

Below is a graph of my results for the average queueing delay as a function of window size:



This graph depicts the average queueing delay as a function of the window size. As I noted after the previous graph, as the window size increases the number of bytes that are put onto the link per a given segment of time, increases significantly. As a result, the queue time begins to increase because of the growing number of packets being put onto the link at any given moment. My results coincide with these anticipated results. The average queueing delay increases substantially with the increase in window size and overall throughput.