Question 2 Part 4:

A fair amount of research went into the making of this algorithm, which henceforth shall be referred to as TCP Almost Vegas. TCP Almost Vegas (bit of a mouthful) was made using the reference material listed below[1][2][3] We chose TCP Vegas to base TCP Almost Vegas on because our algorithm only must compete with itself. TCP Vegas is a very good at using up available bandwidth and it is very fair.

There are 3 major downsides to TCP Vegas. In "Host-to-Host Congestion Control for TCP," Afanasyev claims that TCP Vegas is unable to get a fair share when competing with aggressive congestion control algorithms, it underestimates available network resources in some environments (such as multipath routing), and it has a bias to new streams. Additionally, he claims, "TCP Vegas has the amazing property of rate stabilization in a steady state, which can significantly improve overall throughput." None of the downsides seem to apply to our situation. So, Vegas it was.

We read "TCP Vegas: end to end congestion avoidance on a global Internet" to get a better understanding of how TCP Vegas ran and used the webpage titled 'TCP Vegas'[3] to sure up any parts we didn't understand. [Aside] A lot of these papers seem to be under the assumption that you understand exactly how TCP Tahoe and Reno work. It is very frustrating.

Initially, when writing out this program we tried to implement TCP Vegas exactly. It did not go well. I think the total run time for sending the file while competing against itself was around 88 seconds. This was awful in comparison to TCP Tahoe. After a few tweaks and a lot of trial and error we settled on this version of TCP Almost Vegas.

TCP Almost Vegas can be broken up into 3 key pieces:

1. Congestion Control
2. Handling Dup Ack
3. Timeout

---

[1] A. Afanasyev, N. Tilley, P. Reiher and L. Kleinrock, "Host-to-Host Congestion Control for TCP," in *IEEE Communications Surveys & Tutorials*, vol. 12, no. 3, pp. 304-342, Third Quarter 2010, doi: 10.1109/SURV.2010.042710.00114.

[2] L. Brakmo and L. Peterson, "TCP Vegas: end to end congestion avoidance on a global Internet," IEEE J. Sel. Areas Commun., vol. 13, no. 8, pp. 1465–1480, October 1995.

[3] *TCP Vegas*. CS558a Syllabus & Progress. (n.d.). Retrieved May 26, 2022, from http://www.mathcs.emory.edu/~cheung/Courses/558/Syllabus/02-transport/vegas.html

Congestion Control

Congestion Control has two phases: slow start and congestion control.

Slow start is almost the same as Reno and Tahoe. Window size is increased every other acknowledgment received. This helps to ensure that the window doesn't grow too fast, giving the algorithm time to find hopefully find an accurate minimum RTT. TCP Almost Vegas leaves slow start when $\Delta < 3$ where the value of delta is given by equation 1.

$$\Delta = cwnd \cdot \frac{(RTT - RTT_{min})}{RTT} \quad (1)$$

While in congestion control, TCP Almost Vegas samples the RTT of each packet acknowledged. It uses the current RTT to update RTT_{min} and Δ. If $\Delta > 4$, $cwnd$ is decreased by 1. If $\Delta < 2$, $cwnd$ is increased by 1. Else, $cwnd$ does not change.

Additionally, the window size decreases by 1 in the case of 3 duplicate acknowledgements in a row and timeout.

Handling Duplicate Acknowledgement

There are 3 things that can happen if a duplicate acknowledgment is received.

First, if this is the first or second duplicate acknowledgement received in row, nothing happens. This is changed from TCP Vegas. TCP Vegas, on the first two duplicate acknowledgements, checks to see if the delay of the packet the receiver is expecting is above the coarse grain delay time. We tried to implement this using the per packet delay, but this network profile reorders packets too much and we were sending to many packets onto the network in response. We decided to do nothing here as in Reno/Tahoe would be best.

Second, if 3 duplicate acknowledgements have been received in row. Here, just like Tahoe/Reno/Vegas we resend the packet the receiver is expecting.

Lastly, if more than 3 duplicate acknowledgements have been received in a row, we do nothing. The philosophy behind this is that it is not good to send a lot of extra packets. Say for example your window size is 10 and the second packet is lost. The receiver is going to send 9 acknowledgements for the first packet in the window. At 3 packets we are going to resend packet 2, but then after that how many times should we resend packet 2. We believe 0 after that. Let the stream of duplicate acknowledgements be processed, if the problem isn't fixed after that, a timeout will occur, and the situation will be handled that way. If you are sending the same packet repeatedly, the stream of duplicate acknowledgements in the buffer seems to stay large. The goal is keeping this stream of duplicate acknowledgements as small as possible.

Timeout:

Timeout value is calculated in the same as given by the textbook. See equation 4.

$$estimatedRTT = .875 \cdot estimatedRTT + .125 \cdot sampleRTT \quad (2)$$

$$devRTT = .75 \cdot devRTT + .75|sampleRTT - estimatedRTT| \quad (3)$$

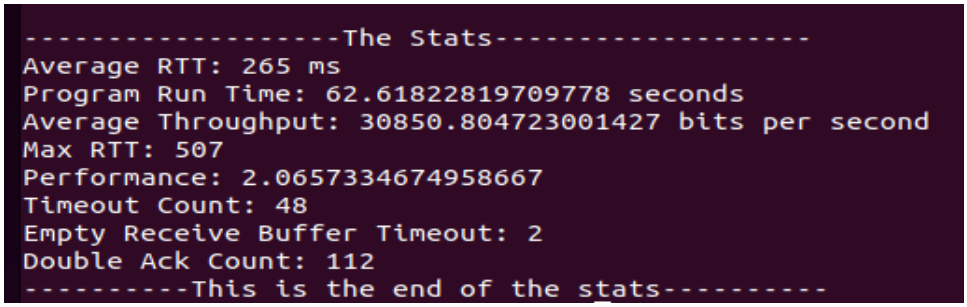
$$TimeoutInterval = estimatedRTT + 4 \cdot devRTT \quad (4)$$

Timeout can occur 2 ways:

First, is if the recv buffer is empty. The timeout for the buffer is set to timeoutInterval before trying to recvfrom(). This is very slow, but it happens very infrequently.

Second, the delay of the oldest unacknowledged packet is checked against the value of timeoutInterval. If it is greater, than a timeout has occurred. In this case we resend the packet. This is very proactive as the algorithm is usually never waiting at the buffer for a packet to arrive.

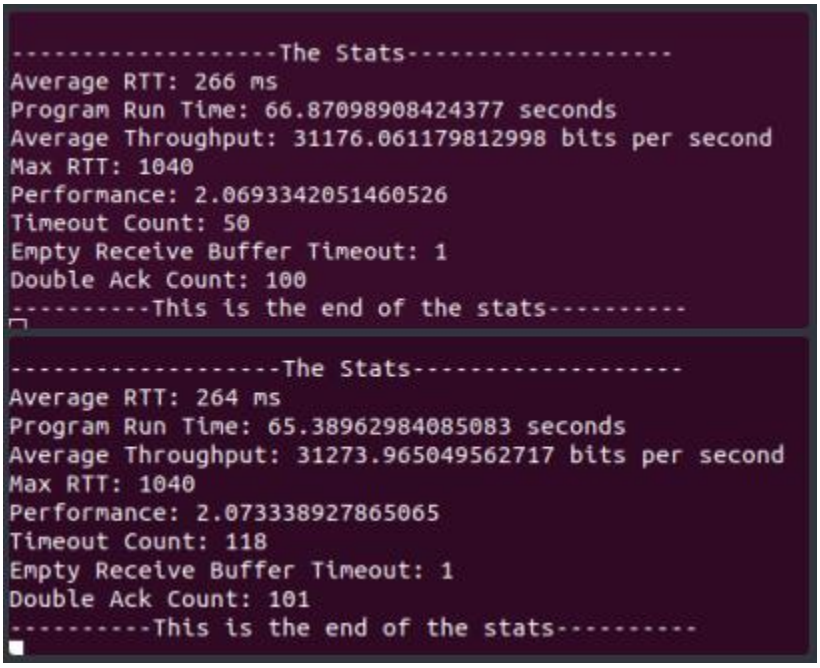
As you can see from the image on the right, we have recorded 2 the two types of timeout. Empty Receive Buffer Timeout is the first case where the program blocks at the buffer.



Timeout counts the number of times that algorithm resends a packet based on the per packet delay.

Stats and Stuff

Below are the recorded statistics of the latest run of TCP Almost Vegas competing against itself.



Average RTT and Average Throughput are poor compared to the metrics provided by the TAs. Average RTT is higher, but it is quite low compared to some implementations we tried. We believe TCP Almost Vegas does a good job in using available network resources. As such one can imagine there are often packets sitting in buffers waiting to be received. The base RTT for the network profile is $200\ ms\ \pm 20\ ms$ It is difficult to keeping RTT down while keeping throughput up.

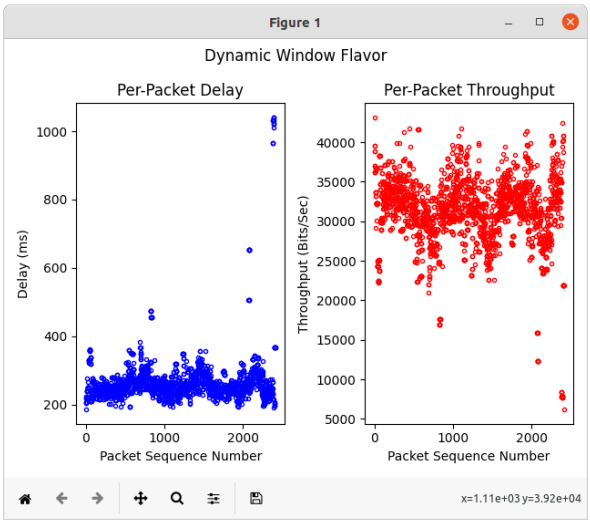
On the topic of throughput, let's talk about that. The throughput mentioned above is a gross miss representation of the actual throughput. If this throughput is to be believed, then it would take 19272000 / 31273 = 616 seconds to send this file. That is not accurate at all. TCP Almost Vegas took 65 seconds for the file to send. With the network profile we sent 19272000 bits in 65 seconds. The throughput is 296492 bits per second.

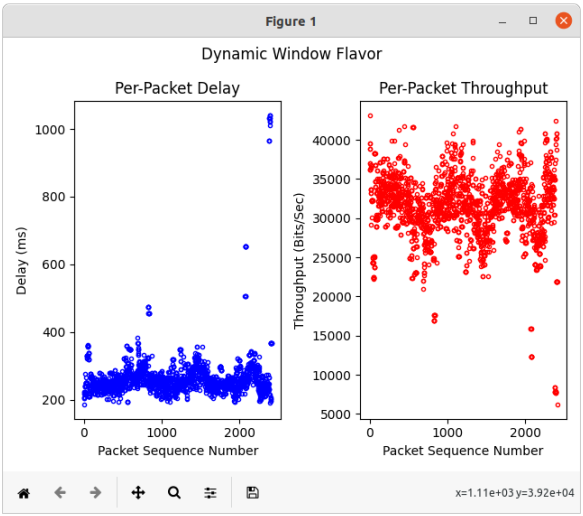
Performance (Using average throughput) = $\log_{10}\left(\frac{31176}{266}\right) = 2.07$

Performance(Using actual throughput) = $\log_{10}\left(\frac{296492}{266}\right) = 3.05$

The rest of the metrics was used to assist in troubleshooting.

Now for some graphs:





As you can see, TCP Almost Vegas does a decent job at keeping the RTT a constant level. The throughput seems to be more variable, but it does appear to ossiculate consistently around the average throughput.

If we had more time, we would have liked to graph the actual throughput as a function of time. Maybe graph, the RTT as a function of the window size. It would be interesting to see how much window size affects the RTT. Window Size as a function of time would also be interesting. I bet you would be able to clearly see the network profile with that. Alas, it is late, and sleep has been ever so elusive these past couple of days. Hope you enjoyed this report.