1 Moving averages and retransmission timeouts (20 points)

1.1 Moving averages (5 points)

A higher value of $\alpha$ has a faster convergence rate because it weighs newer samples with higher weights. We can see this by expanding out the equation for an EWMA filter.

\[
\text{ewma} = \alpha \cdot \text{sample} + \alpha \cdot (1 - \alpha) \cdot \text{sample}_{-1} + \alpha \cdot (1 - \alpha)^2 \cdot \text{sample}_{-2} + \ldots
\]

(1)

, where sample is the current RTT (or some other quantity) sample, sample$_{-1}$ is the previous one, sample$_{-2}$ is two samples ago, and so on. Through this equation, we see that a higher $\alpha$ gives much more weight to the recent samples over earlier ones, while lower $\alpha$ values spread out the weight over these samples more evenly. 3 points for implementing the code and running it using two different alpha values. 2 points for explaining using the equation.

1.2 Retransmission timeouts (15 points)

Look at the solution code for the solutions to each of the TODOs. We need a minimum value of the retransmission timeout so that we never retransmit packets too quickly. This is required under the following scenario: a series of RTT samples are all close to the minimum RTT ($RTT_{min}$), i.e., there is no queueing delay. In this scenario, the mean RTT estimate is close to $RTT_{min}$. The RTT variance is close to zero, because all samples are roughly equal to $RTT_{min}$. Hence, the timeout which is the mean RTT estimate added to 4 times the RTT variance estimate roughly equals the mean RTT, which is $RTT_{min}$. Hence, if one packet accidentally incurs an RTT a bit higher than $RTT_{min}$, it will trigger a timeout right away, leading to a spurious retransmission. If this spurious retransmission happens often enough, it could cause a congestion collapse.

We need a maximum value of the retransmission timeout so that we eventually end up retransmitting every packet. Not having a maximum timeout can degrade transport-layer throughput for TCP because if a particular packet is not received (at least eventually), any packets that are received after that cannot be delivered to the application because TCP provides a reliable, in-order bytestream.

10 points for completing all the TODOs in timeout_calculator.py correctly. 3 points for minimum timeout explanation. 2 points for maximum timeout explanation.

2 The Stop-And-Wait protocol (20 points)

1. Anirudhs-Air:asg2_code_sols anirudh$ python3 simulator.py --seed 1
   --host_type StopAndWait --rtt_min 10 --ticks 50
   Namespace(host_type='StopAndWait', loss_ratio=0.0, queue_limit=1000000,
   rtt_min=10, seed=1, ticks=50, window_size=None)
   sent packet @ 0 with sequence number 0
   @ 9 timeout computed to be 100
   rx packet @ 9 with sequence number 0
sent packet @ 10 with sequence number 1 
@ 19 timeout computed to be 100
rx packet @ 19 with sequence number 1
sent packet @ 20 with sequence number 2
@ 29 timeout computed to be 100
rx packet @ 29 with sequence number 2
sent packet @ 30 with sequence number 3
@ 39 timeout computed to be 100
rx packet @ 39 with sequence number 3
sent packet @ 40 with sequence number 4
@ 49 timeout computed to be 100
rx packet @ 49 with sequence number 4
Maximum in order received sequence number 4

(7 points for any output that reflects the stop and wait structure of proceeding sequentially from one sequence number to the next.)

2. Here’s one set of data points comparing the Stop-And-Wait protocol with the 1/RTT equation

<table>
<thead>
<tr>
<th>RTT-min</th>
<th>StopAndWait</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>200</td>
<td>0.0050</td>
<td>0.005</td>
</tr>
<tr>
<td>300</td>
<td>0.0033</td>
<td>0.00333333...</td>
</tr>
<tr>
<td>400</td>
<td>0.0025</td>
<td>0.0025</td>
</tr>
</tbody>
</table>

To calculate throughput, add 1 to the maximum sequence number (because sequence numbers start from zero) and divide by the number of ticks.

3. Check 1: With 1% random loss, the difference in throughput at an $RTT_{min}$ of 100 ticks, with ticks set to 10000, and the random seed set to 1 is given below (for my implementation):

Throughput without loss: 0.0099
Throughput with loss: 0.0098

Check 2: The protocol continues to function correctly despite losses. Here is a snippet of the output from the protocol with a (very high) loss rate of 0.5 to demonstrate that the protocol can handle losses.

```
Anirudhs-Air:asg2˙code˙sols anirudh$ python3 simulator.py --loss˙ratio 0.5 --seed 1
--host_type StopAndWait --rtt˙min 100 --ticks 10000 | grep "with sequence number 1$"
```

sent packet @ 100 with sequence number 1
retx packet @ 397 with sequence number 1
retx packet @ 991 with sequence number 1
rx packet @ 1090 with sequence number 1

When there are losses, the divergence between the equation’s predictions and the actual throughput are greatest when the $RTT_{min}$ is small. For instance, consider the following data points from my implementation of Stop-And-Wait: for $RTT_{min} = 5$, the throughput is 0.1740, while the prediction is 0.2. But, when $RTT_{min} = 40$, the throughput is 0.0247, while the prediction is 0.025. As the $RTT_{min}$ decreases, the retransmission timeout is more likely to be set to MIN_TIMEOUT (instead of mean + 4 * variance) because the computed timeout is lower than MIN_TIMEOUT. This leads to the timeout being forcibly set to MIN_TIMEOUT, overriding the mean plus 4 variance calculation. Because MIN_TIMEOUT is greater than $RTT_{min}$ at small $RTT_{min}$ values, lost packets take longer to detect and retransmit (relative to packets that are not lost) as the $RTT_{min}$ decreases—because you need to wait at least MIN_TIMEOUT time units to detect a lost packet. This leads to a degradation in transport-layer throughput as the $RTT_{min}$ decreases.
3 The sliding window protocol (20 points)

1. Anirudhs-Air:asg2_code_sols anirudh$ python3 simulator.py --seed 1 --host_type SlidingWindow --rtt_min 10 --ticks 50 --window_size 5
   Namespace(host_type='SlidingWindow', loss_ratio=0.0, queue_limit=1000000, rtt_min=10, seed=1, ticks=50, window_size=5)
sent packet @ 0 with sequence number 0
sent packet @ 0 with sequence number 1
sent packet @ 0 with sequence number 2
sent packet @ 0 with sequence number 3
sent packet @ 0 with sequence number 4
sent packet @ 10 with sequence number 5
sent packet @ 11 with sequence number 6
sent packet @ 12 with sequence number 7
sent packet @ 13 with sequence number 8
sent packet @ 14 with sequence number 9
sent packet @ 20 with sequence number 10
sent packet @ 21 with sequence number 11
sent packet @ 22 with sequence number 12
sent packet @ 23 with sequence number 13
sent packet @ 24 with sequence number 14
sent packet @ 30 with sequence number 15
sent packet @ 31 with sequence number 16
sent packet @ 32 with sequence number 17
sent packet @ 33 with sequence number 18
sent packet @ 34 with sequence number 19
sent packet @ 40 with sequence number 20
sent packet @ 41 with sequence number 21
sent packet @ 42 with sequence number 22
sent packet @ 43 with sequence number 23
sent packet @ 44 with sequence number 24
Maximum in order received sequence number 20

(7 points for any output that reflects the sliding window structure of the protocol, i.e., a window worth of packets is sent out at time 0, another window an RTT-min later, and so on.)

2. Here’s one set of data points comparing the sliding window protocol with the W/RTT equation

<table>
<thead>
<tr>
<th>RTT-min</th>
<th>SlidingWindow (W=10)</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0989</td>
<td>0.1</td>
</tr>
<tr>
<td>200</td>
<td>0.0489</td>
<td>0.05</td>
</tr>
<tr>
<td>300</td>
<td>0.0329</td>
<td>0.03333333...</td>
</tr>
<tr>
<td>400</td>
<td>0.0239</td>
<td>0.025</td>
</tr>
</tbody>
</table>

To calculate throughput, add 1 to the maximum sequence number (because sequence numbers start from zero) and divide by the number of ticks.

3. Check 1: With 1% random loss, here is the difference in throughput at an $RTT_{min}$ of 100 ticks, a window size of 10, 10000 ticks, and the random seed set to 1.

Throughput with loss: 0.0982
Throughput without loss: 0.099

Check 2: The protocol continues to function correctly despite losses. Here is a snippet of the output from the protocol with a (very high) loss rate of 0.25 to demonstrate that the protocol can handle losses.
When there are losses, the divergence between the equation’s predictions and the actual throughput are greatest when the $RTT_{\text{min}}$ and the window size are both small. The reason is similar to the reason for the stop and wait protocol: the retransmission timeout is more likely to be set to MIN_TIMEOUT for small values of $RTT_{\text{min}}$. Further, the degradation of throughput due to a delay in retransmission is most acutely felt when the window size is small, i.e., when the sliding window protocol is closest to the Stop-And-Wait protocol. For larger window sizes, there are typically other packets within the window that are being transmitted (and using up the link capacity).

4 Congestion collapse (20 points)

**Demonstrating congestion collapse:** The following settings demonstrate a congestion collapse: loss_ratio=0.0, host_type=SlidingWindow, rtt_min 100, and ticks=10000 as window size (offered load) is varied from 25 to 1000. Throughput increases until the window size hits the Bandwidth-Delay Product (100, because link capacity is fixed at 1), and then decreases. I have reported throughput values below. Other settings that demonstrate congestion collapse are also acceptable as solutions.

<table>
<thead>
<tr>
<th>Window size</th>
<th>Transport-layer throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.2475</td>
</tr>
<tr>
<td>50</td>
<td>0.4936</td>
</tr>
<tr>
<td>100</td>
<td>0.9730</td>
</tr>
<tr>
<td>150</td>
<td>0.7736</td>
</tr>
<tr>
<td>200</td>
<td>0.7770</td>
</tr>
<tr>
<td>500</td>
<td>0.7636</td>
</tr>
<tr>
<td>1000</td>
<td>0.6952</td>
</tr>
</tbody>
</table>

**Explaining congestion collapse:** In the final row of the previous table, the link delivered (equivalently, the receiver received) 9901 packets. Of this only 6952 were unique, and the remaining 2949 packets were duplicates of these 6952 packets.

5 AIMD (20 points)

1. 5 points for answering all TODOs in the assignment.

2. Decreasing before we wait for an RTT has the net effect of decreasing too fast. When a packet is dropped due to the fact that the queue has overflown, it is quite likely that the same fate will be suffered by packets that arrive back-to-back. Decreasing the congestion window by 1/2 for each packet is too drastic because it doesn’t give the end host enough time to react to the first drop, and it soon drives the congestion window to 1. The reason we wait an RTT specifically is that an RTT is how long it takes for feedback to get back to the sender and the sender’s new action (cutting the window in half) to be reflected at the router.

3. The graph above was generated using my Aimd code using the command line arguments supplied below.

   python3 simulator.py --loss_ratio 0.0 --seed 1 --host_type Aimd --rtt_min 20 --ticks 10000 --queue_limit 10

   Here, the BDP is 20 ($1 \times 20$), while the queue limit is 10. In my graph, the period of the waveform is 754 ticks or approximately 38 $RTT_{\text{min}}$ units. Any graph that demonstrates the sawtooth pattern will be accepted as a solution. If you plot the window size over time and explain it, but it does not demonstrate a sawtooth pattern, you’ll lose 3 points out of 5.
4. Using the same settings as the previous question, but varying the queue limit alone, here’s what I get using my Aimd implementation.

<table>
<thead>
<tr>
<th>Queue limit</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 (1/2 BDP)</td>
<td>0.7718</td>
</tr>
<tr>
<td>20 (1 BDP)</td>
<td>0.9022</td>
</tr>
<tr>
<td>30 (1.5 BDP)</td>
<td>0.9807</td>
</tr>
<tr>
<td>40 (2 BDP)</td>
<td>0.9897</td>
</tr>
<tr>
<td>50 (2.5 BDP)</td>
<td>0.9912</td>
</tr>
<tr>
<td>60 (3 BDP)</td>
<td>0.9912</td>
</tr>
</tbody>
</table>

The important takeaway is that the transport-layer throughput (maximum number of packets received in order at the receiver) goes up as the queue limit goes up. Your numbers will probably be different from mine, and we’ll still give you points so long as you can demonstrate an increase in throughput with increasing queue limits.

5. The purpose of having a non-zero queue limit is that it serves as a buffer for Aimd when it cuts the window in half. When Aimd cuts its window in half, the arrival rate of packets coming into the queue is diminished—until Aimd ramps up its window through congestion avoidance to the BDP or higher. During this period where the window is not yet at the BDP (hence the arrival rate \( \frac{W}{R_T} \) is less than the link capacity), we would ideally still want to keep the link occupied with gainful work, even if the arrival rate is less than the link capacity. The queue limit allows us to do this by buffering enough packets to keep the link occupied while Aimd is ramping up again. The question of how large your queue limit needs to be to provide high link utilization has seen much work because it affects how much router memory you need. A good example of a paper that addresses this question is [http://yuba.stanford.edu/~nickm/papers/sigcomm2004.pdf](http://yuba.stanford.edu/~nickm/papers/sigcomm2004.pdf).