Homework 3 Part 3

# Question 1

We measured the steady-state throughput r(W) for a range of window sizes W=1,2,4,...,1024, with RTT fixed at 0.5 seconds, no packet loss, and link speed set to 1 Gbps. The results show:

* Throughput grows linearly with window size
* The observed rate matches the model:
* At W=1024, measured throughput was ~24 Mbps, matching expected behaviour.

Python Code in Appendix

A graph with a line

AI-generated content may be incorrect.

# Question 2

We measured the steady-state throughput r(RTT)r(RTT)r(RTT) for a fixed window size W=30W = 30W=30 packets, while varying the RTT from 10 ms to 5120 ms. The actual measured RTT values were used to account for emulation and OS-induced deviations.

The results show:

* **Throughput decreases inversely with RTT**, consistent with the theoretical model
* The observed rate closely matches the model:
* These findings validate the relationship r(RTT) ∝ 1/RTT​, though minor deviations exist due to queuing and system overheads in the test environment.

Code in Appendix

A graph of a graph

AI-generated content may be incorrect.

# Question 3

The experiment measured throughput on a local dummy receiver with window size W=8000W = 8000W=8000. With standard 1472-byte packets, the peak throughput reached approximately **845 Mbps**, remaining below the 1 Gbps threshold. In contrast, using 9KB jumbo packets resulted in a peak throughput of approximately **4.7 Gbps**. This difference illustrates how larger packet sizes reduce per-packet processing overhead and significantly improve achievable throughput, especially on high-speed links. The test system used an Intel Core i5-9400 CPU (6 cores @ 2.90 GHz) with 16 GB RAM.

# Question 4

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

In both the loss-free and the reverse-path loss (10%) scenarios, the protocol achieves nearly the same throughput — around 9390 Kbps — because the sender can tolerate some ACK loss without stalling. When an ACK is lost, it doesn’t always require a timeout or retransmission; a later ACK (for a higher sequence number) can still acknowledge earlier packets and advance the window. This cumulative ACK behavior allows the sender to continue transmitting without waiting for every single ACK to arrive. Additionally, with a large window size of 300 packets, the sender has enough in-flight packets to keep the pipeline full, maintaining high throughput even when a few ACKs are dropped. Since the RTT estimate and retransmission timeout (RTO) adapt over time, occasional ACK loss has little impact on overall performance.

# Question 5

The receiver uses **TCP flow control** to advertise its current receive window in each ACK. This window is based on the remaining space in the receiver's buffer and limits how much data the sender can transmit without acknowledgment. This technique ensures the sender does not overwhelm the receiver. The advertised window has a fixed **upper bound** determined by the size of the receiver’s buffer. Without window scaling, this upper bound is 65,535 bytes (64 KB). With TCP Window Scaling, this can be extended to 1 GB (1,073,741,824 bytes).

**65,535 bytes (64 KB) — Default Maximum Without Scaling**

* **TCP headers have a 16-bit field for the Window Size in the TCP header.**
* **A 16-bit field can hold values from 0 to 216−1=65,5352^{16} - 1 = 65,535216−1=65,535.**
* **So, without window scaling, the max advertised window is 65,535 bytes (just under 64 KB).**

**This is defined in RFC 793 (TCP specification).**

**1,073,741,824 bytes (1 GB) — With Window Scaling**

* **In RFC 1323, TCP introduced Window Scaling to support high-bandwidth networks.**
* **Window scaling adds a scale factor (shift count) from 0 to 14, applied to the 16-bit window.**
* **This is negotiated during the TCP 3-way handshake using the Window Scale option.**

# Appendix

Question 1

import matplotlib.pyplot as plt  
import numpy as np  
  
# Data  
W = np.array([1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024])  
r = np.array([0.023, 0.047, 0.093, 0.187, 0.374, 0.748, 1.493, 2.992, 5.991, 12.859, 24.020])  
  
# Theoretical throughput (MSS = 1472 bytes, RTT = 0.5s)  
MSS = 1472  
RTT = 0.5  
theoretical = (W \* MSS \* 8) / RTT / 1e6  
  
# Plot  
plt.figure(figsize=(8, 5))  
plt.semilogx(W, r, 'o-', label='Measured r(W)')  
plt.semilogx(W, theoretical, 'x--', label='Theoretical r(W)', alpha=0.7)  
plt.xlabel('Window Size W (packets)')  
plt.ylabel('Throughput r(W) [Mbps]')  
plt.title('Throughput vs Window Size')  
plt.grid(True, which='both', linestyle='--', alpha=0.5)  
plt.legend()  
plt.tight\_layout()  
plt.savefig('throughput\_plot.png') # <- Save the plot  
plt.show()

Question 2

import matplotlib.pyplot as plt

import numpy as np

# Measured values

RTT = np.array([0.011, 0.021, 0.041, 0.080, 0.162, 0.321, 0.642, 1.281, 2.561, 5.122]) # in seconds

r\_measured = np.array([31.841, 16.582, 8.604, 4.390, 2.106, 1.229, 0.527, 0.351, 0.176, 0.091]) # in Mbps

# Theoretical model: r(RTT) = (W \* MSS \* 8) / RTT

W = 30

MSS = 1472 # bytes

r\_theory = (W \* MSS \* 8) / RTT / 1e6 # Mbps

# Plot

plt.figure(figsize=(8, 5))

plt.semilogx(RTT, r\_measured, 'o-', label='Measured r(RTT)')

plt.semilogx(RTT, r\_theory, 'x--', label='Theoretical r(RTT)', alpha=0.7)

plt.xlabel('RTT (seconds)')

plt.ylabel('Throughput r(RTT) [Mbps]')

plt.title('Throughput vs RTT at W = 30 (log-scale x-axis)')

plt.grid(True, which='both', linestyle='--', alpha=0.5)

plt.legend()

plt.tight\_layout()

plt.show()