Ahmed Irtija (920742338)
Python: sender_stop_and_wait.py, sender_fixed_sliding_window.py, sender_tahoe.py, sender_reno.py, sender_custom.py

Team Zubair

Protocol Name - Zubair Protocol

Stop and Wait (sender_stop_and_wait.py)

In our stop and wait file, we demonstrate a simple file sending protocol using UDP sockets, focusing on sending data from a file ('file.mp3') and calculating performance metrics like throughput and average delay. We defined packet size, reserved a portion for the sequence ID, and calculated the message size. The 'finack' function is designed for the end of the transmission process, managing final messages and acknowledgments. We then read and store the file's data, then create a UDP socket, bind it to a local address, and set a timeout to handle potential packet loss.

During transmission, we enter a loop where each iteration constructs and sends a packet comprising a sequence ID and a segment of the file's data. It records the time taken for each packet to be acknowledged, calculating the delay for each. If an acknowledgment is not received within the timeout, we just resend the packet. Once all the data is transmitted, the 'finack' function is called to conclude the session. The total throughput is calculated as the total data transmitted over the total time taken, and we compute the average delay per packet. These metrics, including a custom metric combining throughput and delay.

Stop and Wait Trials	Throughput (bps)	Avg Delay (s)	Performance
1	10013.16	0.10	98337.82
2	9928	0.10	96678.59
3	10044.88	0.11	98902.65
4	9923.65	0.09	95591.37
5	10145.89	0.09	98199.61
6	9937.59	0.09	95716.77
7	10391.77	0.11	99723.34
8	10463.66	0.11	97609.24
9	10102.76	0.10	97307.40
10	10391.77	0.11	99723.34

Average	10,134.31 bps	0.101 seconds	97,779.01
Standard Deviation	197.61 bps	0.0083 seconds	1,407.60

Sliding Window (sender_fixed_sliding_window.py)

In the sliding window, we implemented the same process as the previous one, focusing on sending data packets from a file ('file.mp3') to a receiver and measuring performance metrics like throughput and average delay per packet, however here we send multiple files at a time. We began by defining constants for packet size, sequence ID size, and the window size, which determines how many packets are sent before waiting for an acknowledgment. It reads the data from 'file.mp3' and stores it in memory. Two key functions, `resend` and `finack`, manage the sending of data packets and the final termination of the session, respectively. `resend` constructs and sends a packet with a sequence ID and a slice of the file data, while `finack` handles the exchange of final messages and acknowledgments with the receiver.

We also created a UDP socket, bonded it to a local address, and set a timeout for handling packet loss. It then enters a loop to transmit the data, sending packets within a specified window size and tracking the total packets sent and their respective delays. If an acknowledgment is received, we update the sequence ID and window size for subsequent transmissions. In case of a timeout or repeated acknowledgments for the same packet, we resend the packet, assuming it was lost or delayed. After all data is sent, the script calculates the throughput (total data sent over total time), average delay per packet and the metric.

Sliding Window Trials	Throughput (bps)	Avg Delay (s)	Performance
1	75419.61	1.3	57833.81
2	78232.32	1.29	60471.06
3	82269.62	1.22	67488.94
4	81393.27	1.25	63793.99
5	78691.83	1.24	63733.05
6	81244.23	1.23	63786.27
7	75946.20	1.28	67381.23
8	80119.37	1.25	64500.02
9	76146.83	1.26	60954.59
10	82557.35	1.20	61807.35

Average	79202.06	1.25	63175.03
Standard	2706.45	0.03	3014.16

Deviation			

TCP Tahoe (sender_tahoe.py)

Our TCP Tahoe congestion control mechanism uses a UDP socket, focusing on efficient data transmission and calculation of network performance metrics. TCP Tahoe itself is a congestion control algorithm that adjusts the rate of data transmission based on network feedback. For this reason, our implementation sets up the packet size, reserves bytes for the sequence ID, and calculates the message size. It reads data from 'file.mp3' and uses functions `resend` and `finack` for packet retransmission and to finalize the communication session, respectively.

We created a UDP socket, bound it to a local port, and set a timeout for simulating packet loss detection. It uses a loop to control data transmission, guided by Tahoe's congestion control strategies. The window size, which dictates the number of packets sent before waiting for an acknowledgment, starts small and increases as acknowledgments are received, indicating a slow start phase. If a packet loss is inferred from a timeout or receiving duplicate acknowledgments, the window size is reduced, and the slow start threshold is adjusted, which is characteristic of the Tahoe algorithm. We also calculated performance metrics such as throughput (total data sent over total time), average delay per packet and the metric.

TCP Tahoe Trials	Throughput (bps)	Avg Delay (s)	Performance
1	261583.32	6.32	41416.71
2	270268.94	6.33	42695.19
3	271987.38	6.34	42233.87
4	266922.19	6.33	42887.50
5	271707.94	6.38	42363.64
6	261774.12	6.35	41719.02
7	269598.82	6.36	41874.06
8	262150.30	6.30	42395.26
9	274170.03	6.36	41120.45
10	267827.72	6.36	42333.53

Average	267799.08	6.34	42103.92
Standard Deviation	4599.08	0.02	560.36

TCP Reno (sender_tahoe.py)

Our TCP Reno congestion control algorithm, implemented using a UDP socket to transmit data from a file ('file.mp3') and calculate network performance metrics. TCP Reno is a lot better than Tahoe, in part because it introduces the 'Fast Recovery' algorithm. For this reason, we defined packet size, reserved space for a sequence ID, and set the message size. It reads and stores the file data, then employs functions like `resend` and `finack` for data transmission and closing the communication session.

As with the others, we created a UDP socket, bound it to a local port, and set a timeout to simulate packet loss scenarios. In the main loop, it follows the Reno congestion control strategy: starting with a small window size and increasing it as long as acknowledgments are received, signifying the slow start phase. Upon detecting duplicate acknowledgments, indicative of packet loss, Reno differs from Tahoe by reducing the window size to the slow start threshold (instead of reducing it to 1) and then transitioning into congestion avoidance, not slow start. This 'Fast Recovery' mechanism allows Reno to recover more quickly from packet losses than Tahoe. We tracked the total packets sent and their respective delays, adjusting the window size based on network feedback. Once finished, we call on the `finack` function to end the session. Then calculated throughput (total data sent over total time), average delay per packet and the metric.

TCP Reno Trials	Throughput (bps)	Avg Delay (s)	Performance
1	237339.78	6.06	39183.47
2	257607.08	5.84	44094.8
3	225739.4	10.39	21736.91
4	279962.36	5.75	48670.2
5	260238.27	7.76	28092.16
6	232622.95	10.35	34056.70
7	227735.58	6.18	27843.92
8	238925.70	6.68	25451.15
9	241822.65	6.46	24090.50
10	254211.81	8.77	39377.23

Average	245620.56	7.42	33259.70

Standard Deviation	17069.83	1.81	9200.78

TCP Custom [Zubair Protocol] (sender_custom.py)

Our custom sender, named Zubair Protocol, also works to transmit a file, specifically an MP3 file, from a sender to a receiver. We start by defining constants for the packet size, sequence ID size, and the message size. Then we read the entire content of the 'file.mp3' into memory. The resend function is used to send packets to the receiver at a specific address ('localhost', 5001) and returns the time when the packet was sent. The finack function handles the closing of the communication session by sending a final message to the receiver and waiting for an acknowledgment and a 'fin' message.

The core of our program creates a UDP socket and sets its timeout. It initializes variables for packet tracking, total delay, and sequence IDs. Our congestion control mechanism is similar to the TCP sliding window protocol and slow start algorithm. Packets are sent in a window, and the window size adjusts based on network conditions. The script detects timeouts and duplicate acknowledgments to adjust the window size and slow-start threshold accordingly.

After receiving an acknowledgment for a packet, the script updates the sequence ID, adjusts the window size (either linearly or exponentially, based on the slow-start threshold), and computes the new window range for sending packets. We also keep track of the total delay for each packet and the total number of packets sent.

After all packets are sent, the finack function is called to close the session. The last thing we do is calculate the throughput, average delay per packet, and the metric which is the ratio of throughput to average delay, providing a measure of efficiency.

TCP Custom Trials	Throughput (bps)	Avg Delay (s)	Performance
1	459928.18	8.29	55450.87
2	458393.58	7.82	58626.26
3	529299.43	3.68	143717.09
4	458781.27	7.79	58872.8
5	497538.32	5.80	120220.59
6	476946.62	7.93	121177.85
7	475165.32	5.42	58747.58
8	452548.85	6.25	81468.33
9	464727.00	4.86	90547.13
10	512879.50	6.04	55969.97

Average	478620.81	6.39	84479.85

Standard Deviation	26118.41	1.53	33033.40

Submission Page

I certify that all submitted work is my own work. I have completed all of the assignments on my own without assistance from others except as indicated by appropriate citation. I have read and understand the university policy on plagiarism and academic dishonesty. I further understand that official sanctions will be imposed if there is any evidence of academic dishonesty in this work. I certify that the above statements are true.

Team Member 1:

Ahmed Irtija		12/6/23
Full Name (Printed)	Signature	Date