Student 1: Chris Nguyen (921229957)
Student 2: Andre Vojtenyi (919403594)

Team Name: Miami 67er's

Python files:

- protocol_stats.py
- sender_fixed_sliding_window.py
- sender_ml_classifier.py
- sender_reno.py
- sender_stop_and_wait.py
- sender_tahoe.py
- pantheon_data_collection.py
- pantheon_data_preprocess.py
- model.py

Stop and Wait

To implement Stop-and-Wait, we designed a sender that reads a payload file, splits it into MSS-sized chunks, and sends the chunks as individual UDP packets to a receiver. Each packet is sent sequentially, and the sender waits for an acknowledgement from the receiver before moving to the next packet, timing the interval between sending and receiving each ACK to track delays. If an ACK is not received within a fixed timeout, the sender transmits the packet up to 5 times. After all packets are sent, an end-of-file packet signals the transfer's completion and acknowledgement. During transmission, the sender records delays per packet to calculate metrics: throughput, average delay, and average jitter.

Run	Throughput (bytes/s)	Avg Delay (s)	Avg Jitter (s)	Score
1	6916.7456166	0.1252654	0.0032775	4856.3759529
2	7326.5278396	0.1190395	0.0051026	3233.9509907
3	7793.6016608	0.1345394	0.0039352	4072.1876596
4	7797.4453662	0.1089885	0.0049641	3343.0802613
5	7166.2268511	0.1198366	0.0034363	4657.5479269
6	7004.8859287	0.1228598	0.0040974	3946.0550054
7	7031.4239718	0.1216384	0.0047392	3453.1277119
8	7265.1363770	0.1174599	0.0040247	4025.2370894
9	7268.1474938	0.1179414	0.0038142	4229.7391447

10	7276.7957630	0.1177941	0.0052769	3140.0033756
Average	7149.384	0.120536	0.004267	3895.731
SD	301.1512084	0.0065393	0.0007070	594.0074835

Fixed Sliding Window

To implement our fixed sliding window, we created a sender that reads a payload creating MSS-sized chunks, constructing each chunk as a packet with a sequence number. The sender has a fixed window size of 100 that can be in-flight and transmits all packets in the current window to the receiver. For each sent packet, the sender records the time to measure delays. It then waits for acknowledgements from the receiver and marks the corresponding packets upon receiving an ACK and updates the base of the window to allow new packets to be sent. If an ACK is not received within a fixed timeout, all unacknowledged packets in the window are retransmitted. After all packets are sent and acknowledged, the sender sends a EOF packet indicating end of transfer and waits for the acknowledgement.

Run	Throughput (bytes/s)	Avg Delay (s)	Avg Jitter (s)	Score
1	86365.9821651	0.4358238	0.0013903	10869.1819877
2	78002.8487925	0.4676507	0.0028041	5424.1301693
3	75917.7347883	0.4916577	0.0030955	4916.9189382
4	75159.9925763	0.4775255	0.0024224	6265.6095897
5	92057.9442681	0.3662451	0.0013502	11205.0041653
6	70298.0908846	0.5149054	0.0028323	5364.0488978
7	76842.3082031	0.4645461	0.0020507	7389.8464322
8	74444.6610137	0.4832388	0.0028638	5310.2803791
9	78942.5605268	0.4361074	0.0017682	8563.5386854
10	84214.7935395	0.4158211	0.0013480	11211.8221086
Average	79224.692	0.455352	0.002193	7652.038
SD	6479.2866980	0.0429784	0.0006967	2616.8382329

TCP Tahoe

To implement Tahoe, we designed a sender that would send a packet and wait for an acknowledgement from the receiver that the packet was successfully received. Each successful acknowledgement would result in our sender increasing the sending window exponentially while the program was still in slow start, and then linearly once the program reached congestion control. If the sender experienced a timeout (it doesn't receive an acknowledgement within an allotted time frame) or a triple acknowledgment (receiving three acks of the same packet), the slow start threshold would be cut in half, and the current window size would be reset back to 1, until finally trying to retransmit the earliest unacknowledged packet.

Run	Throughput (bytes/s)	Avg Delay (s)	Avg Jitter (s)	Score
1	4145.0042532	0.1676767	0.0094962	1788.8013551
2	2624.8649459	0.1032305	0.0092766	1956.7771925
3	3002.3459342	0.2103435	0.0140132	1237.4804258
4	2221.7430458	0.2012348	0.0239504	801.1219808
5	580.9267752	0.0872355	0.0243654	1020.2829357
6	1032.3598041	0.0929162	0.0139976	1450.2330767
7	4511.2279025	0.4147221	0.0159788	1023.5798153
8	2022.7456011	0.4612205	0.0255716	663.4633428
9	359.0659019	0.0978250	0.0429830	712.3269659
10	5645.7085557	0.3696113	0.0123212	1312.4626093
Average	2614.599	0.220602	0.019195	1196.653
SD	1748.2937409	0.1429862	0.0103267	440.2245000

TCP Reno

To implement Reno, we designed a sender that acts very similarly to our implementation of Tahoe. Reno behaves similarly on a timeout, but when it experiences a triple acknowledgement, it only halves the congestion window instead of dropping it all the way to 1, and then instantly starts trying to retransmit the earliest unacknowledged packet.

Run	Throughput (bytes/s)	Avg Delay (s)	Avg Jitter (s)	Score
1	49586.7859969	0.3093125	0.0277325	654.0770036

2	37554.3858767	0.2799875	0.0220048	806.7278210
3	83523.5588274	0.3109275	0.0205182	843.6497432
4	64471.1552495	0.5031979	0.0265784	633.9549352
5	60550.8195746	0.6114246	0.0227396	716.9179874
6	53811.3347244	0.1230520	0.0184086	1099.3050389
7	29842.5995726	1.3086654	0.0262297	598.6821036
8	85984.8568073	0.2137420	0.0158068	1112.7313695
9	40940.7586950	1.0155453	0.0305131	526.1054909
10	47701.0596403	0.6115449	0.0267953	617.0730586
Average	55396.731	0.528740	0.023733	760.922
SD	18593.5832309	0.3772538	0.0046086	205.1668435

Custom Protocol - CWND Classifier

To implement the custom congestion control, we designed a model-driven sender that adjusts the congestion window (CWND) based on real-time network measurements. The sender collects three metrics: loss, delay, and throughput. The values were fed into a machine learning classification model via a multi-class logistic regression model solving for the formula:

$$Cwnd = \alpha*loss + \beta*delay + \gamma*throughput + bias$$

Each label was given an output for CWND action: decrease, hold, increase. The model gives a probability of each action with a linear combination of the metrics with learned coefficients applying a softmax to them.

We collected data from an online dataset called Pantheon, which measured throughput, delay, and packet loss from multiple congestion control algorithms. We encoded feature labels based on changes in the metrics: Decrease if loss or delay increased, or a timeout occurred; Increase if throughput increased while delay and loss did not increase significantly; Hold otherwise. The dataset was then scaled and encoded, and the processed data was fitted to a logistic regression model.

This Python script implements a simplified custom reliable transport protocol over UDP, inspired by TCP Reno, to send data from a sender to a receiver while dynamically adjusting its congestion window based on network conditions. The sender reads a payload file, splits it into fixed-size chunks, and transmits them to a specified host and port, tracking the sequence

number of each packet. For each received acknowledgment (ACK), it calculates round-trip delay and throughput, keeps track of duplicate ACKs, and adapts the congestion window using a logistic regression model that decides whether to increase, decrease, or hold the window based on loss, delay, and throughput metrics. It also handles timeouts and implements fast retransmit when three duplicate ACKs are received, mimicking TCP Reno's congestion control behavior. After all data is sent, it transmits an end-of-file packet and reports metrics including total throughput, average delay, jitter, and a combined performance metric.

Run	Throughput (bytes/s)	Avg Delay (s)	Avg Jitter (s)	Score
1	3110.3954627	0.1236642	0.0039295	4100.9109344
2	1771.6020437	0.2145506	0.0057508	2772.6122375
3	1165.2585541	0.1287821	0.0008359	18218.4259806
4	1993.7817465	0.6337517	0.0140062	1127.1858651
5	1268.6139565	0.2655381	0.0065426	2426.0577282
6	1598.8763665	0.3778821	0.0090213	1756.5998481
7	1643.4572563	0.1272424	0.0008395	18143.7092133
8	1318.0131434	0.1529757	0.0008914	17057.2689675
9	1545.5084326	0.3826545	0.0095005	1671.6167884
10	1990.0564674	0.3041146	0.0081834	0.0081834
Average	1740.556	0.271116	0.00595	6922.345
SD	558.1299649	0.1620432	0.0044003	7558.7109176