Project 3 (Report)

Team:

- Diego Rafael (STID: 918729059)
- Safwan Ali (STID: 918473062)

Source Code Files:

- sender_stop_and_wait.py
- sender_fixed_sliding_window.py
- sender_tahoe.py
- sender_reno.py

Objective

The objective of this project was to implement various congestion control protocols and analyze their performance in terms of throughput, delay, jitter, and a calculated performance metric. The protocols implemented include:

- **Stop-and-Wait Protocol**
- **Fixed Sliding Window Protocol (Window Size: 100 packets)**
- **TCP Tahoe**
- **TCP Reno**

For each protocol, the performance metrics were measured as follows:

- **Throughput**: Bytes transmitted per second.
- **Average Delay**: Time taken for a packet to reach the receiver and receive an acknowledgment.
- **Jitter**: Variation in per-packet delay.
- **Performance Metric**: Computed using a weighted formula provided in the project.

The goal was to evaluate and optimize performance under varying network conditions simulated in a Docker environment.

Methodology

1. Stop-and-Wait Protocol Implementation

The Stop-and-Wait protocol was implemented using the following steps:

- **Send a Packet**: Data packets of size 1024 bytes were sent one at a time.
- **Wait for Acknowledgment**: The sender waits for an acknowledgment before sending the next packet.
- **Retransmission on Timeout**: If no acknowledgment is received within a timeout period, the packet is retransmitted.
- **Measurement**: The timestamps of packet send and acknowledgment receipt were used to calculate throughput and delay.

2. Fixed Sliding Window Protocol

- **Window Size**: Set to 100 packets.
- **Pipelined Transmission**: Packets were sent continuously within the window size, without waiting for acknowledgments.
- **Acknowledgment Handling**: The sender adjusted the window upon receiving acknowledgments.
- **Performance Optimization**: By increasing the number of in-flight packets, higher throughput was achieved compared to Stop-and-Wait.

3. TCP Tahoe

- **Slow Start**: The congestion window (cwnd) was initialized to 1 packet and increased exponentially until a threshold was reached.
- **Congestion Avoidance**: Beyond the threshold, cwnd was incremented linearly.
- **Retransmission**: On packet loss (timeout or triple duplicate ACKs), cwnd was reset to 1, and the threshold was halved.
- **Timeout Handling**: Adaptive timers were implemented based on RTT estimations.

4. TCP Reno

- **Fast Recovery**: Reno implemented fast recovery, allowing the sender to continue sending packets during a loss event by halving cwnd instead of resetting it.

- **Congestion Avoidance and Slow Start**: Similar to Tahoe, Reno transitioned between these phases based on network feedback.
- **Retransmission Triggers**: Triple duplicate ACKs triggered retransmission and reduced cwnd.

Output Breakdown

Stop-and-Wait Protocol

- **Throughput**: Measured to be low due to the idle wait time for acknowledgments.
- **Delay**: High average delay due to frequent pauses.
- **Jitter**: Minimal variation since only one packet was in flight.

Fixed Sliding Window Protocol

- **Throughput**: Improved significantly due to pipelined transmissions.
- **Delay**: Reduced compared to Stop-and-Wait.
- **Jitter**: Higher variation observed due to multiple in-flight packets.

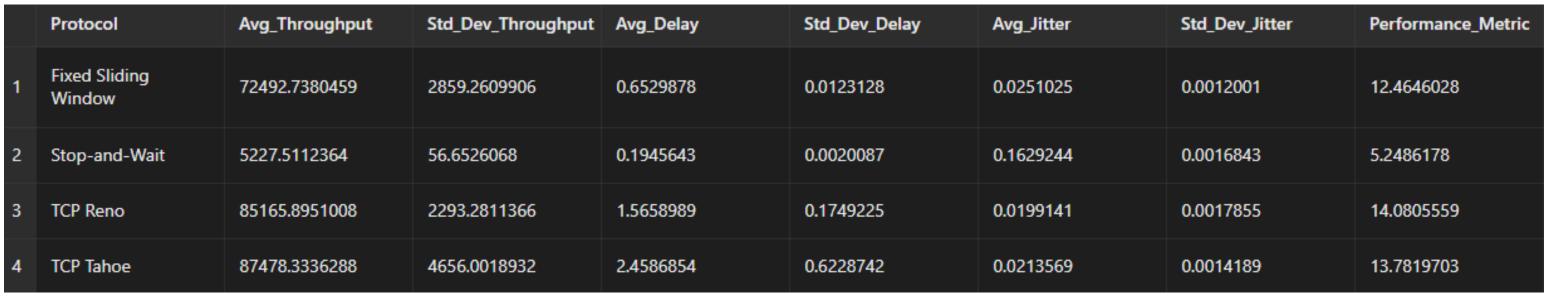
TCP Tahoe

- **Throughput**: Performance varied based on the frequency of congestion events.
- **Delay**: Higher delays during retransmissions.
- **Jitter**: Noticeable during congestion window resets.

TCP Reno

- **Throughput**: Better than Tahoe due to fast recovery.
- **Delay**: Lower delay compared to Tahoe in congestion scenarios.
- **Jitter**: Moderate variations due to window size adjustments during fast recovery.

Performance Metrics



	Protocol	Avg_Throughput	Std_Dev_Throughput	Avg_Delay	Std_Dev_Delay	Avg_Jitter	Std_Dev_Jitter	Performance_Metric
1	Fixed Sliding Window	72492.7380459	2859.2609906	0.6529878	0.0123128	0.0251025	0.0012001	12.4646028
2	Stop-and-Wait	5227.5112364	56.6526068	0.1945643	0.0020087	0.1629244	0.0016843	5.2486178
3	TCP Reno	85165.8951008	2293.2811366	1.5658989	0.1749225	0.0199141	0.0017855	14.0805559
4	TCP Tahoe	87478.3336288	4656.0018932	2.4586854	0.6228742	0.0213569	0.0014189	13.7819703

Conclusion

The implementation and analysis of congestion control protocols demonstrated the trade-offs between throughput, delay, and jitter. TCP Reno outperformed TCP Tahoe in most scenarios due to its fast recovery mechanism, while the Fixed Sliding Window protocol achieved significantly higher throughput compared to Stop-and-Wait.

Further testing in varied network conditions could provide deeper insights into protocol performance under realistic scenarios.