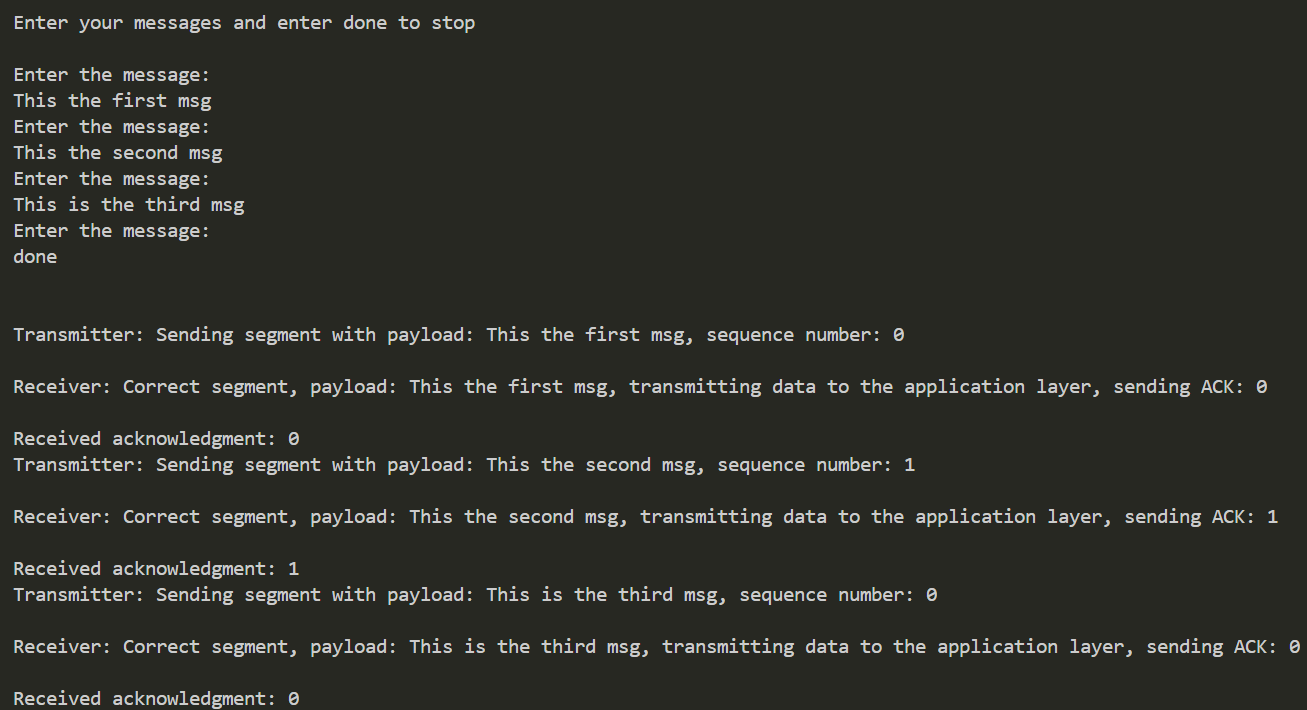
**EGC-123 Computer Networks**

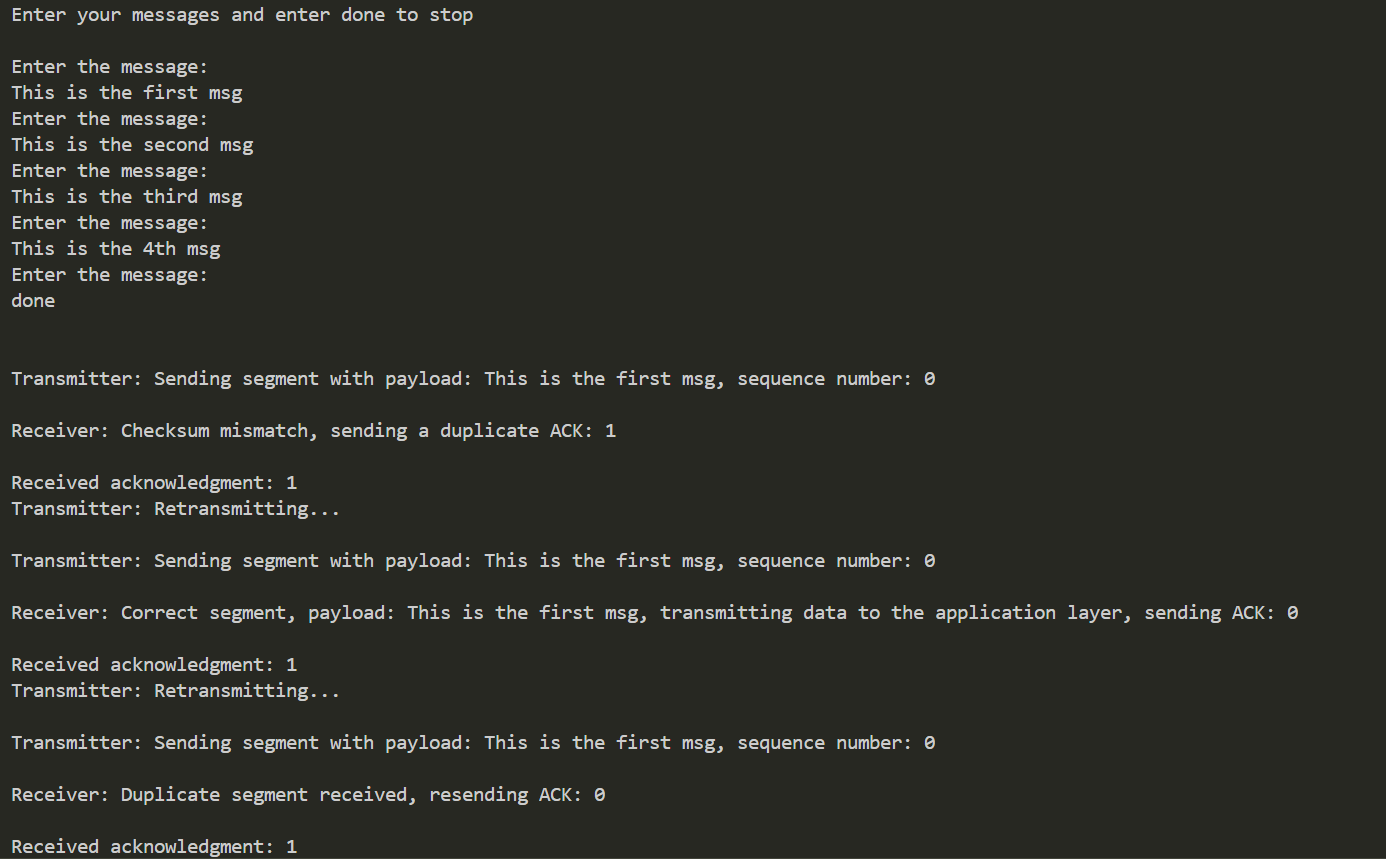
**Report**

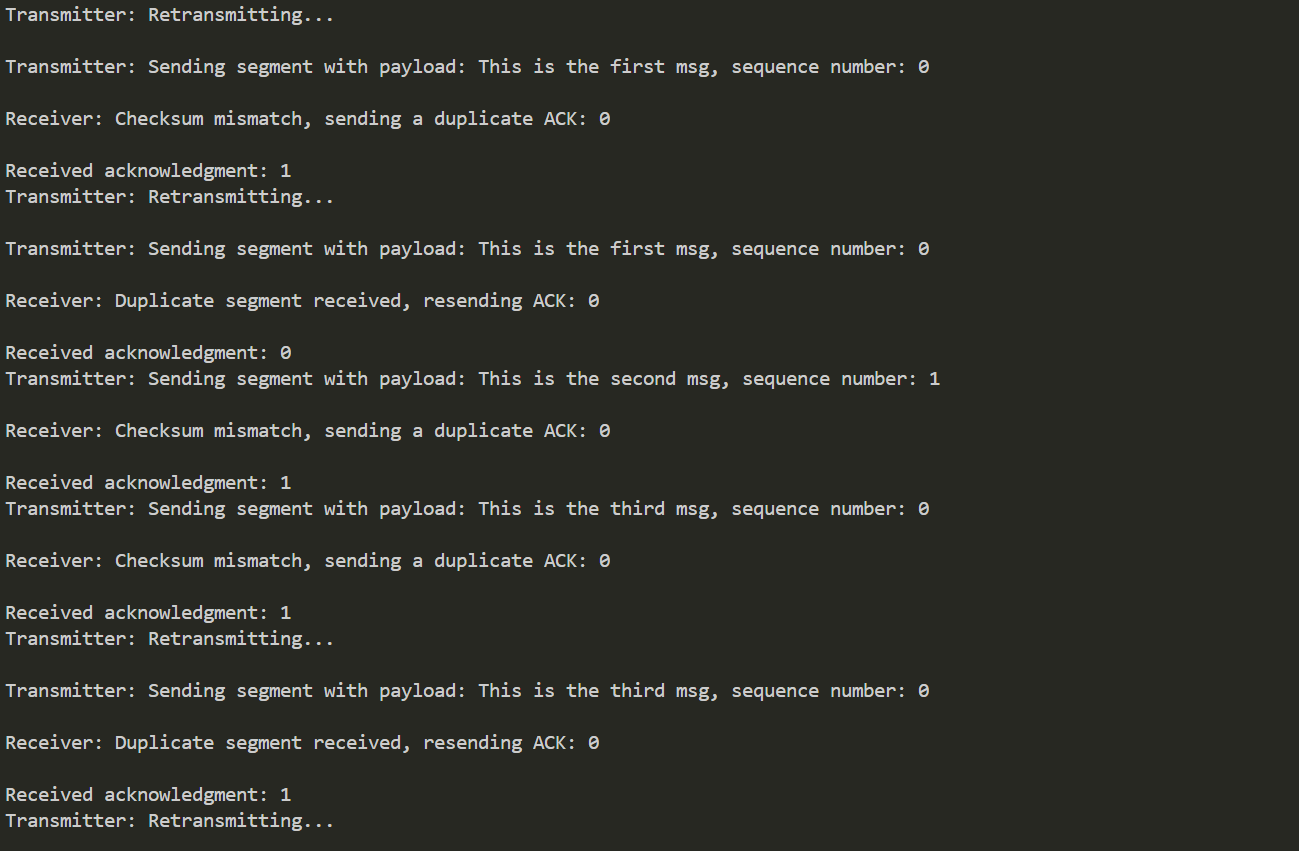
By Rohan Kamath

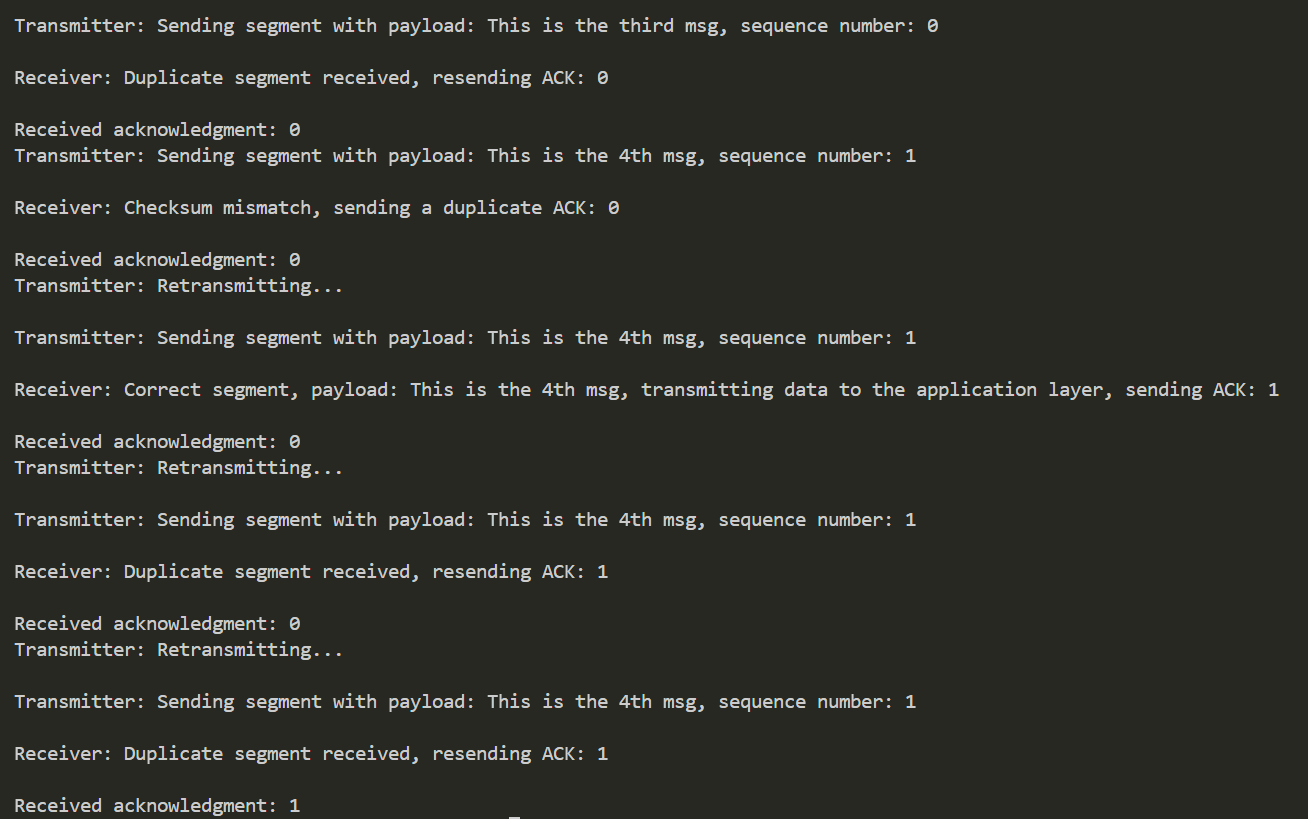
1. RDT 2.2

The code simulates the RDT 2.2 (Reliable Data Transfer) protocol using Python, which ensures reliable communication over an unreliable network channel. It defines a NetworkChannel class to simulate message corruption and acknowledgment errors with configurable error rates. Data to be sent is wrapped in a DataSegment containing a sequence number, payload, and checksum for error detection. The TransmitterNode repeatedly sends segments until it receives the correct acknowledgment, using stop-and-wait logic and alternating sequence numbers (0 and 1). The ReceiverNode validates the checksum and sequence number to detect and discard duplicates or corrupted segments, sending acknowledgments accordingly. The protocol ensures that messages are delivered reliably and in order despite the simulated network unreliability. The script also allows user input for messages and transmits them one by one using this reliable transfer mechanism.

Ideal case – where neither the packet sent doesn’t get corrupted nor ACKs are lost or corrupted



****

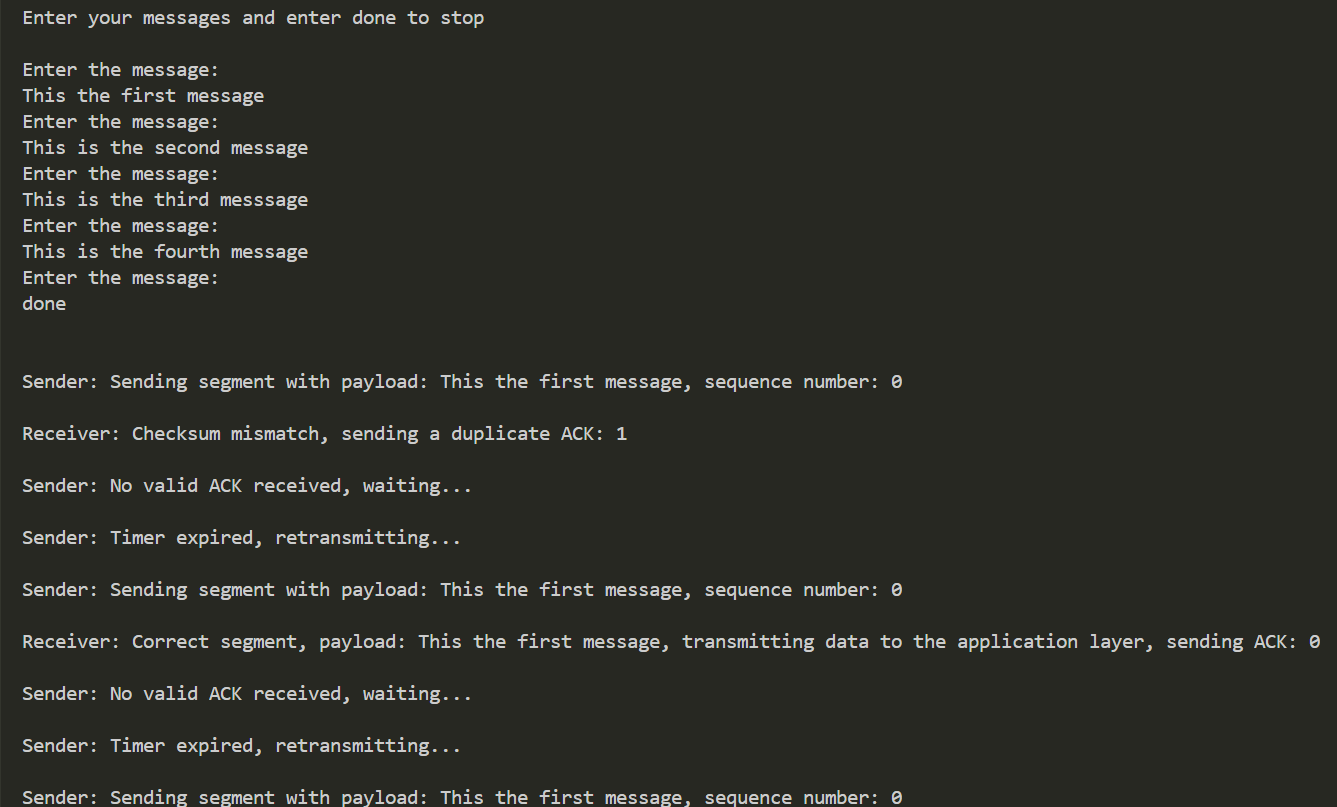
****

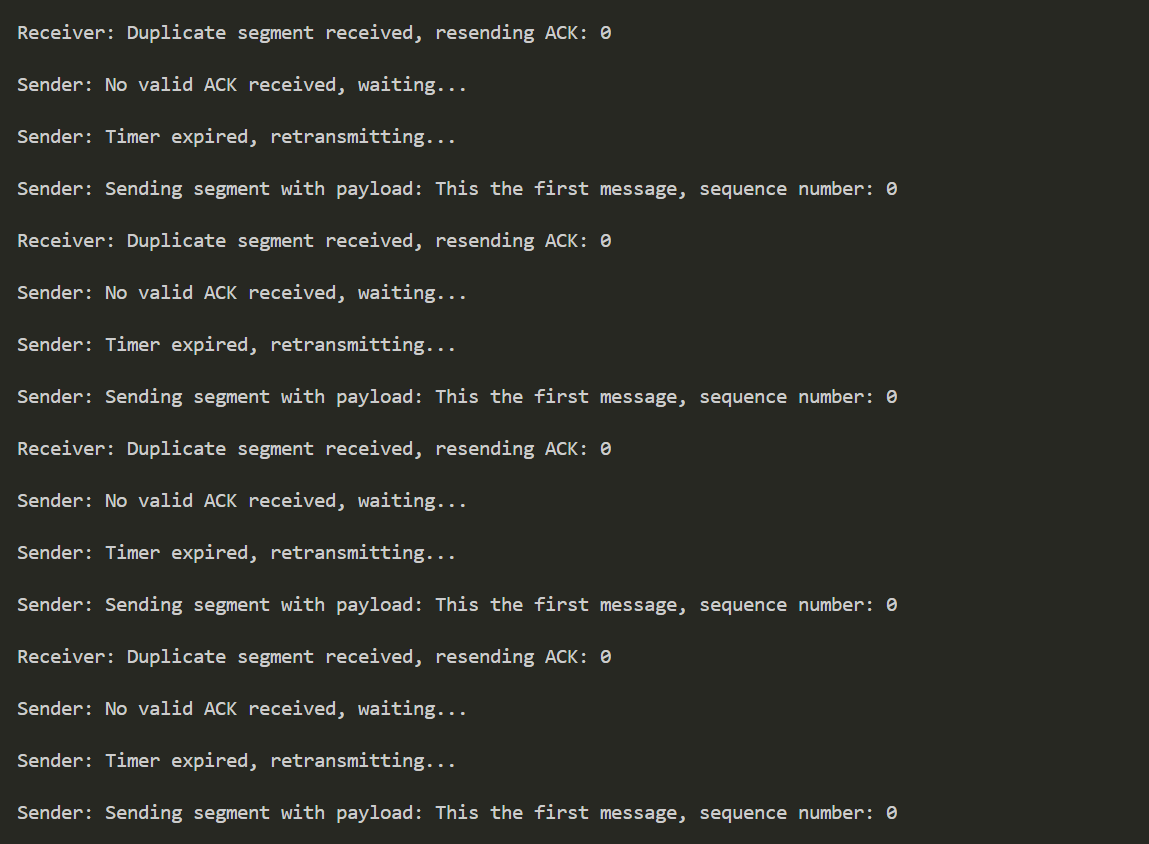
The above three screenshots show the case where the either the packet sent or the ACK received get corrupted

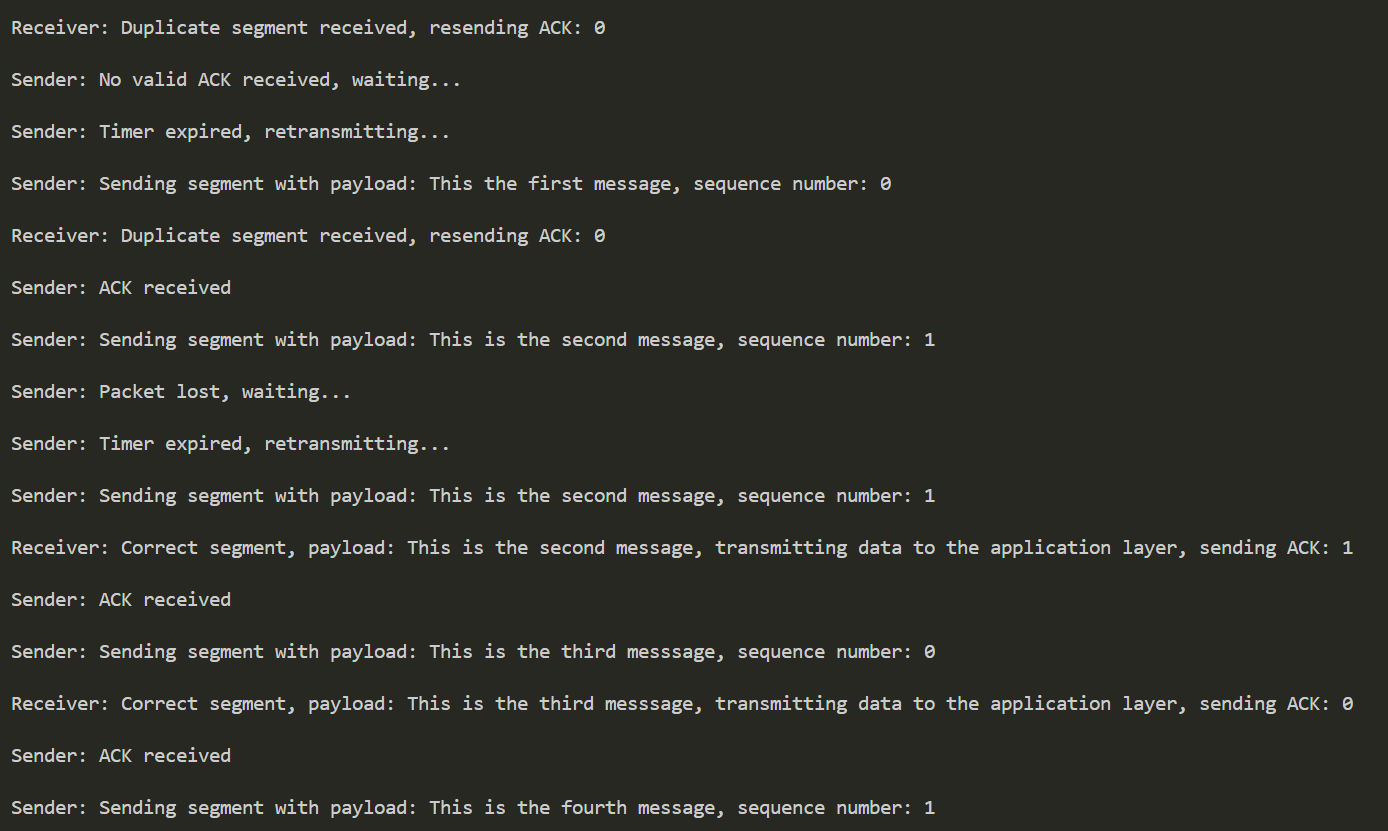
1. RDT 3.0

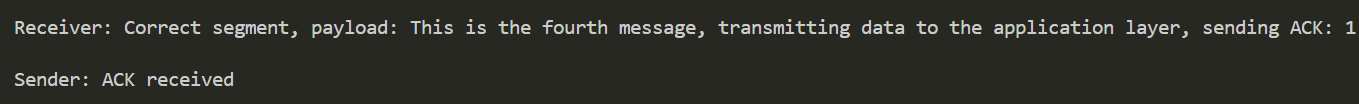
This code implements a simulation of the Reliable Data Transfer (RDT) 3.0 protocol in Python, which is designed to ensure reliable communication over an unreliable network that may introduce packetcorruption or *loss*. The core components are the Sender, Receiver, and NetworkChannel classes. The sender and receiver use alternating bit protocol (ABP), where a 1-bit sequence number (0 or 1) is used to distinguish between new and duplicate packets. The Sender class handles message transmission, waits for acknowledgments (ACKs), and retransmits if an ACK is lost or incorrect. The Receiver class validates each incoming segment using a checksum and sequence number, and it responds with appropriate ACKs to help the sender determine whether retransmission is necessary.

The NetworkChannel class simulates an unreliable medium by randomly corrupting or dropping packets based on configurable error and loss probabilities. A simple integrity check (checksum) is computed using the segment's sequence number and payload to detect errors. Each data packet is encapsulated in a DataSegment that supports serialization and deserialization for transmission. The program accepts multiple user inputs and reliably sends them across the simulated channel, handling potential errors and ensuring each message reaches the receiver without duplication or corruption, following the principles of RDT 3.0.







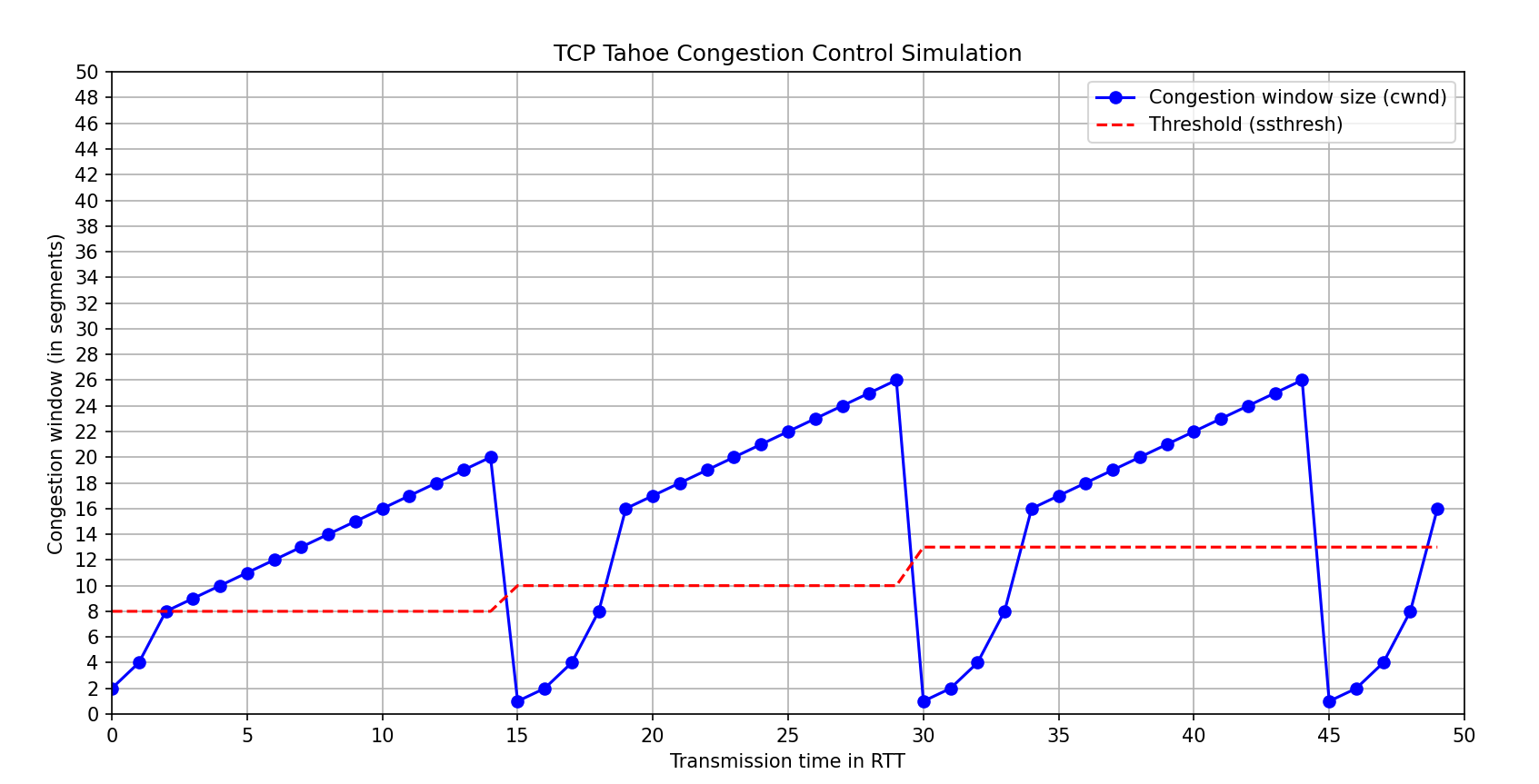
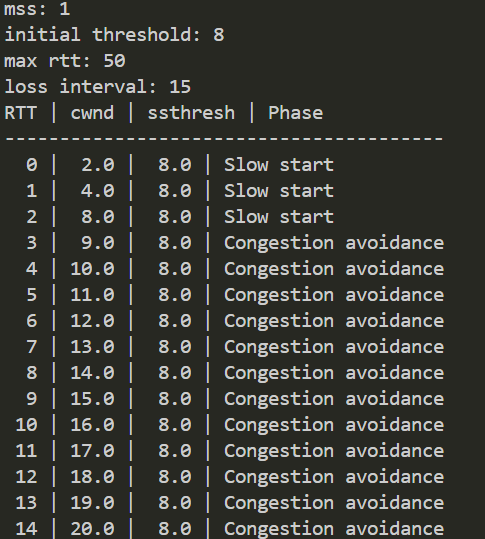


The above screenshots illustrate various network anomalies including corruption of sent packets, corrupted or lost acknowledgments (ACKs), delayed ACKs, and both packet and ACK loss, all of which are effectively handled by the RDT 3.0 protocol to ensure reliable data transfer.

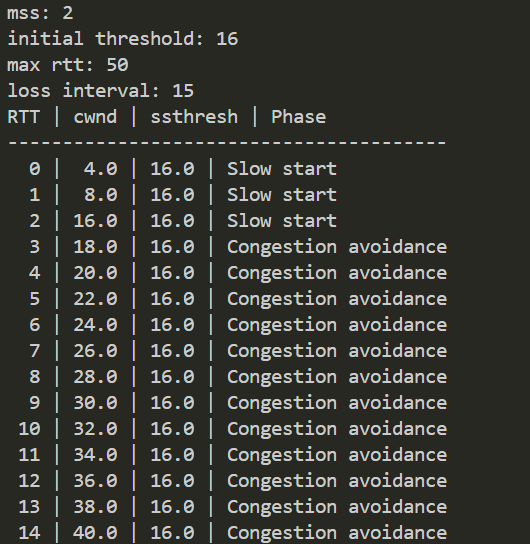
1. TCP TAHOE Congestion Control

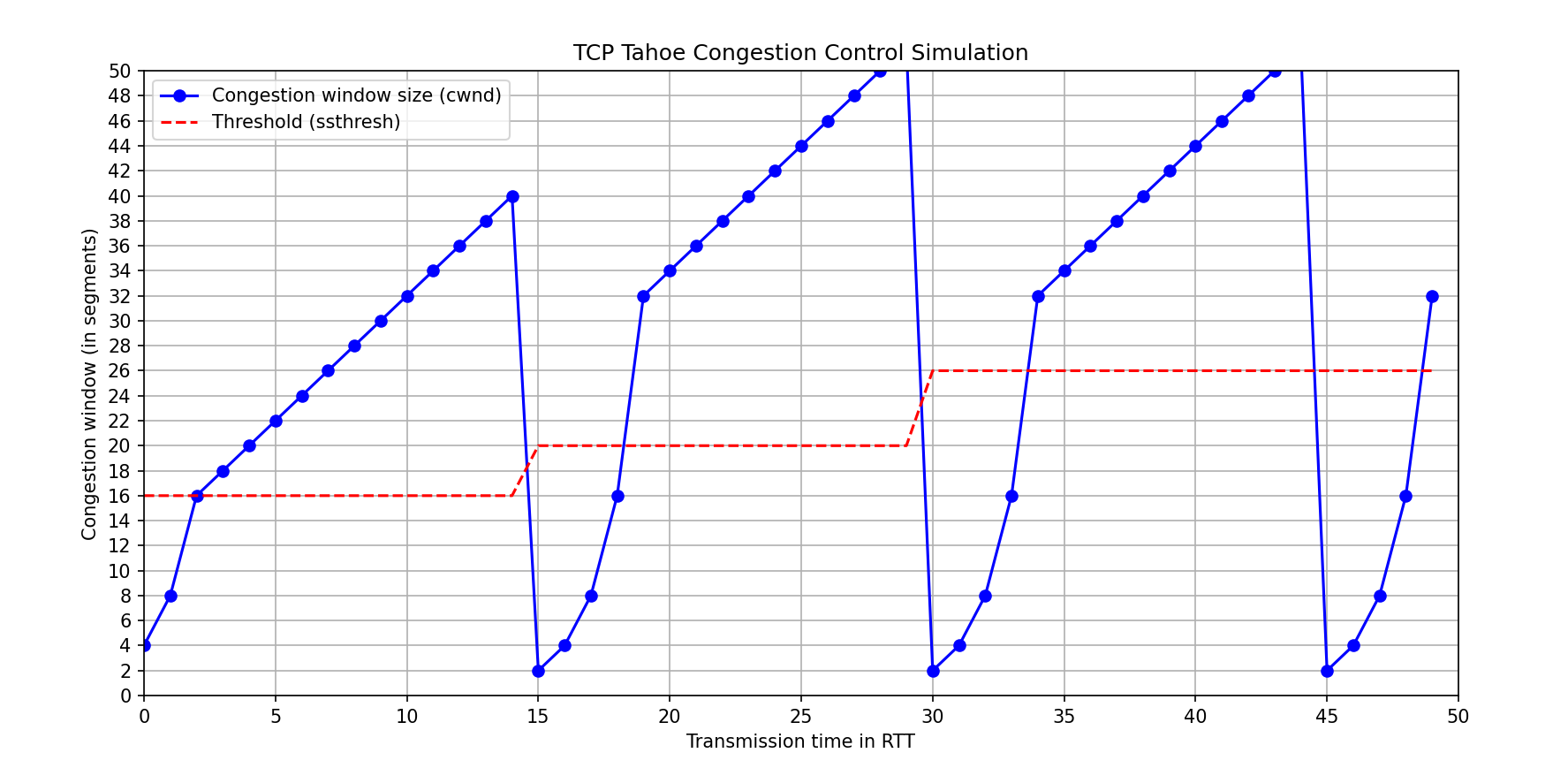
This program simulates the TCP Tahoe congestion control algorithm using Python in an object-oriented approach. It models how TCP adjusts its congestion window (cwnd) in response to network conditions across successive RTTs (Round-Trip Times). The simulation begins with a congestion window equal to the Maximum Segment Size (MSS) and a user-defined slow-start threshold (ssthresh). TCP Tahoe starts in the slow start phase, exponentially increasing cwnd until it reaches the threshold, after which it enters congestion avoidance, where cwnd grows linearly. If a packet loss is detected (simulated periodically or with low probability), Tahoe triggers a multiplicative decrease, halving the threshold and resetting cwnd to the initial MSS — a defining behavior of TCP Tahoe.

The code separates concerns neatly: simulation parameters are initialized via user input, the main logic is handled in the update\_window() and run() methods, and all key metrics (cwnd, ssthresh, RTTs, and phases) are logged for analysis. A plotting function visualizes the dynamics of the congestion window, with vertical lines marking loss events. Additionally, a tabular log output shows how cwnd evolves step-by-step. This structured design and use of clear simulation phases make the implementation both intuitive and illustrative of how TCP Tahoe regulates data flow in response to congestion.



Output for mss=1, intitial threshold=8



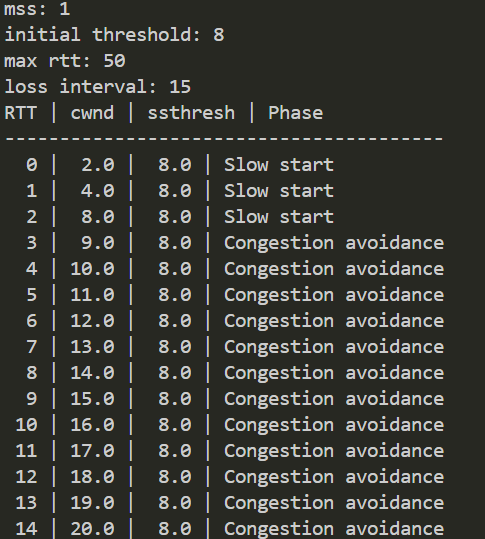


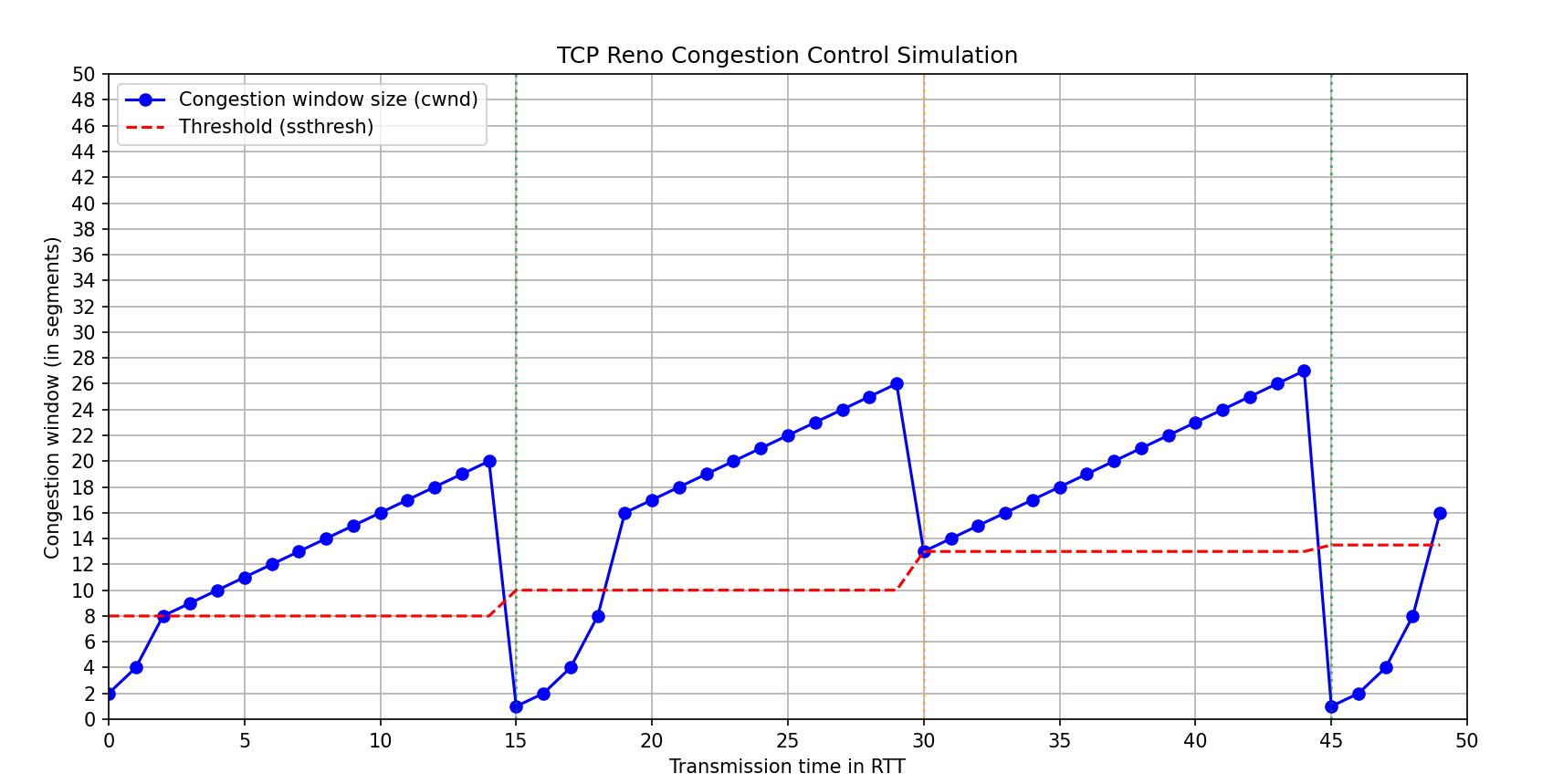
Output for mss=2, intital threshold=16

1. TCP RENO Congestion Control

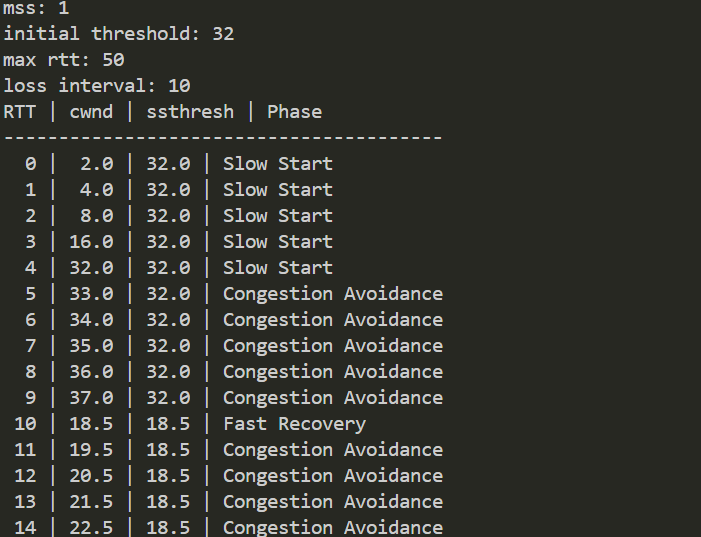
This Python program simulates the TCP Reno congestion control protocol, modeling how the congestion window (cwnd) evolves over time in response to network conditions. The simulation is structured as a class-based implementation, where key parameters like Maximum Segment Size (MSS), initial slow-start threshold (ssthresh), total simulation duration (max\_rtt), and a packet loss interval are provided by the user. At each simulated Round-Trip Time (RTT), the code updates the congestion window according to the Reno protocol's rules: it begins with slow start (exponential growth of cwnd), transitions to congestion avoidance (linear growth), and reacts to simulated packet loss events. These losses may be timeouts or triple duplicate ACKs, determined either at fixed intervals or probabilistically.

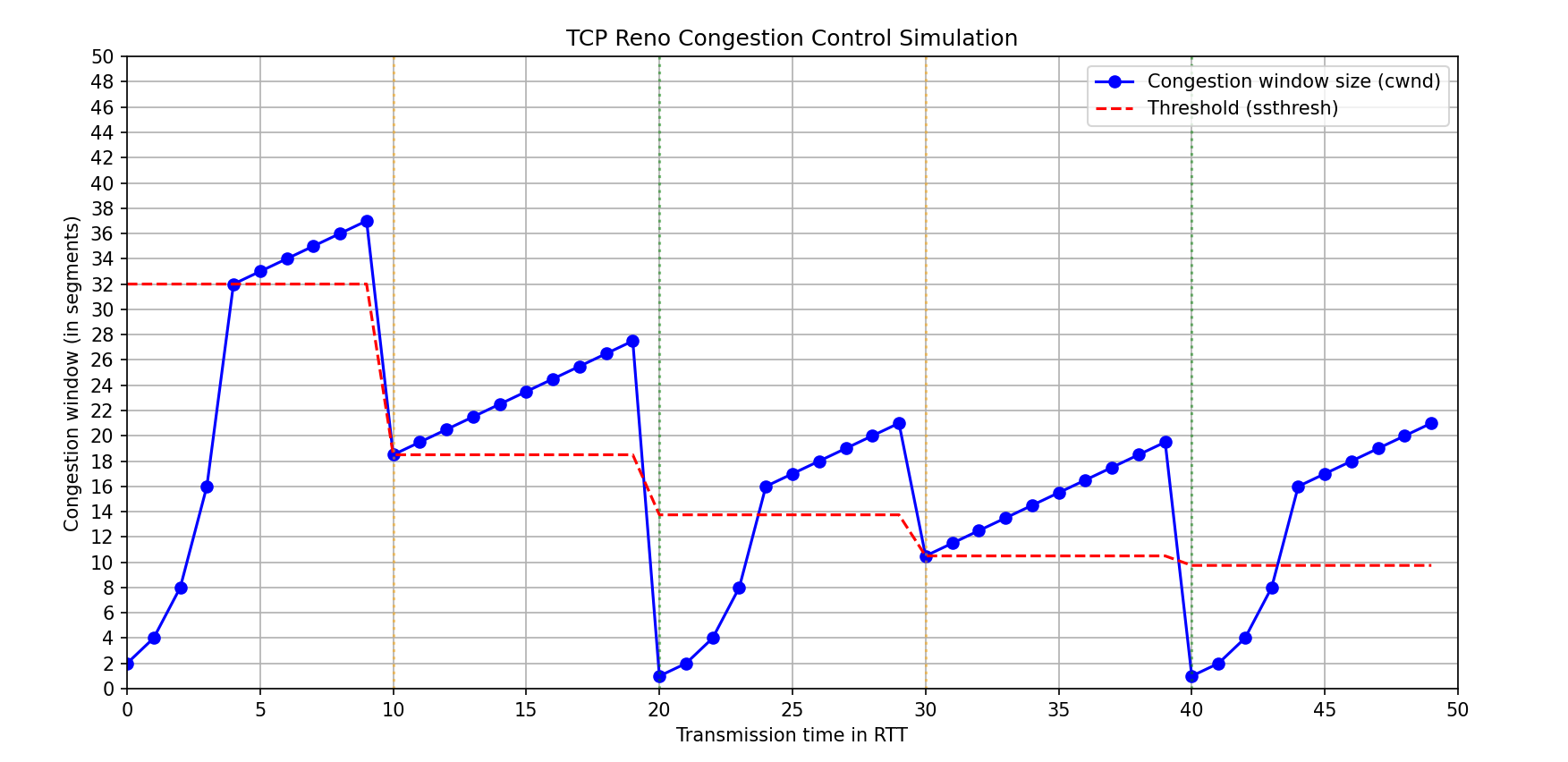
The core logic resides in the update\_window() method. During a loss, TCP Reno distinguishes between timeouts and duplicate ACKs. In case of a timeout, it resets cwnd to MSS and re-enters slow start. However, for triple duplicate ACKs, Reno enters fast recovery, halving cwnd and temporarily increasing it linearly until the loss is recovered. This nuanced behavior helps maintain higher throughput than Tahoe. The simulation logs values of cwnd, ssthresh, RTT count, and congestion phase, which are then used to plot a detailed graph showing the evolution of the congestion window and threshold. Color-coded vertical lines indicate loss events to visually distinguish between phases. The design is modular, with clear separation between simulation, logging, and visualization, making it ideal for educational or experimental analysis of TCP Reno.





Output for mss=1, intitial threshold=8





Output for mss=1, intitial threshold=32

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*