Blake Whitaker 5438770

EEL4781-25FALL0001 Computer Communication Networks Project 1

Go-Back-N Protocol Simulation Report

This project focuses on the implementation and simulation of the Go-Back-N (GBN) Automatic Repeat Request (ARQ) protocol. GBN is a fundamental sliding window protocol designed to facilitate reliable data transmission over unreliable networks. By allowing the sender to transmit multiple packets up to a fixed window size without waiting for individual acknowledgments, GBN significantly improves channel utilization compared to simpler Stop-and-Wait protocols. This project simulates the complete lifecycle of the protocol, including a sender that manages windowing and timeouts, a channel that introduces propagation delay and packet loss, and a receiver that validates packet order and generates cumulative acknowledgments.

Implementation

The simulation was developed using Python and the Tkinter library to create an interactive Graphical User Interface (GUI). The application is built upon an event-driven architecture featuring two primary loops. A logic loop runs once per second to handle discrete protocol events such as timer checks, packet generation, and state updates. Simultaneously, an animation loop runs at 50 frames per second to render the smooth movement of packets between the sender and receiver, providing a clear visual representation of transmission and propagation delays.

The core logic is divided into three classes. The Sender class manages the state of the sliding window, tracking the oldest unacknowledged packet and the next sequence number to be sent. The Receiver class enforces strict in-order delivery, maintaining an expected sequence number, and discarding any packet that does not match. The Channel class simulates the physical network by holding packets in a queue to enforce propagation delay and filtering packets based on user-defined loss configurations.

Timers and Sequence Numbers

Go-Back-N relies on a single logical timer associated with the oldest unacknowledged packet in the window. In my implementation, this is managed via a timestamp variable that resets whenever the window base advances. During every simulation tick, the sender checks if the time elapsed since the timer started exceeds the configured timeout threshold. If this condition is met, a timeout event is triggered, causing the sender to retransmit all currently unacknowledged packets in the window.

The protocol uses a linear sequence number space ranging from zero to the total number of packets minus one. To support retransmission, the sender maintains a buffer of all sent packets. When a timeout occurs, the sender retrieves the original packet data from this buffer for retransmission. On the receiving end, the logic strictly follows the GBN specification by not buffering out-of-order packets; any packet arriving out of sequence is discarded, and the receiver re-sends the acknowledgment for the last correctly received packet.

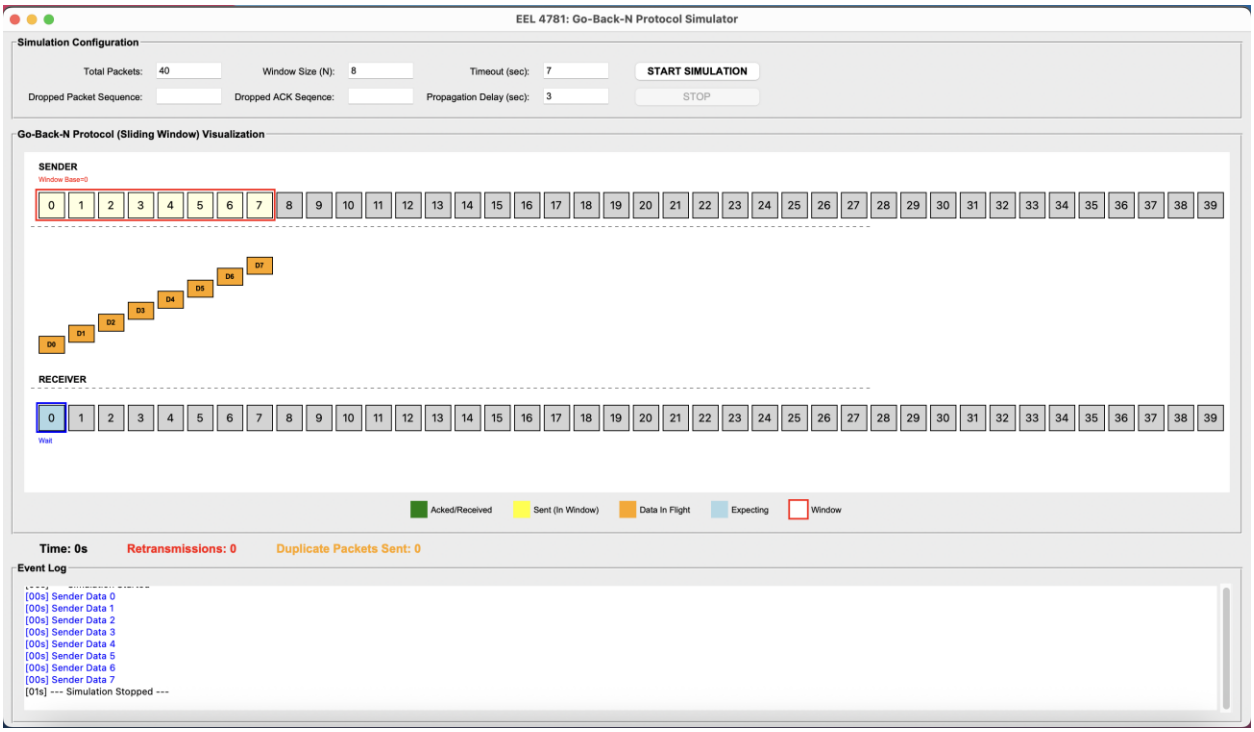
Observations and Simulation Results

I executed the simulation under several network conditions but w/ a constant propagation delay of 3 secs and timeout of 7 secs to prevent premature timeouts from the round trip time being longer than the timeout interval. All the parameters are adjustable in the GUI. The values I used were chosen for clarity in the animations and to match the scenario on display.

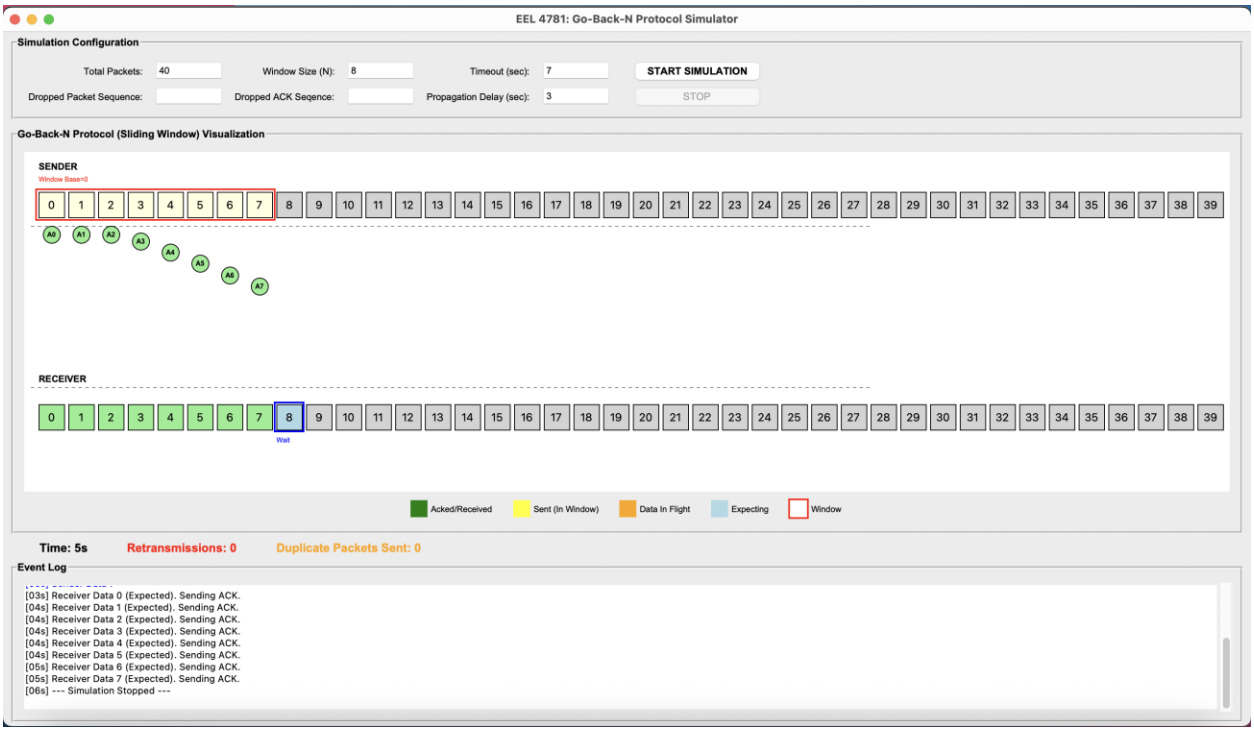
Scenario A: Normal Operation Ideal Conditions

In an ideal scenario with no packet loss, the protocol demonstrated maximum throughput. The visual representation showed a continuous flow of packets, with the sender's window sliding forward smoothly as acknowledgments arrived. The logs confirmed that packets were received in order and the window advanced without interruption.

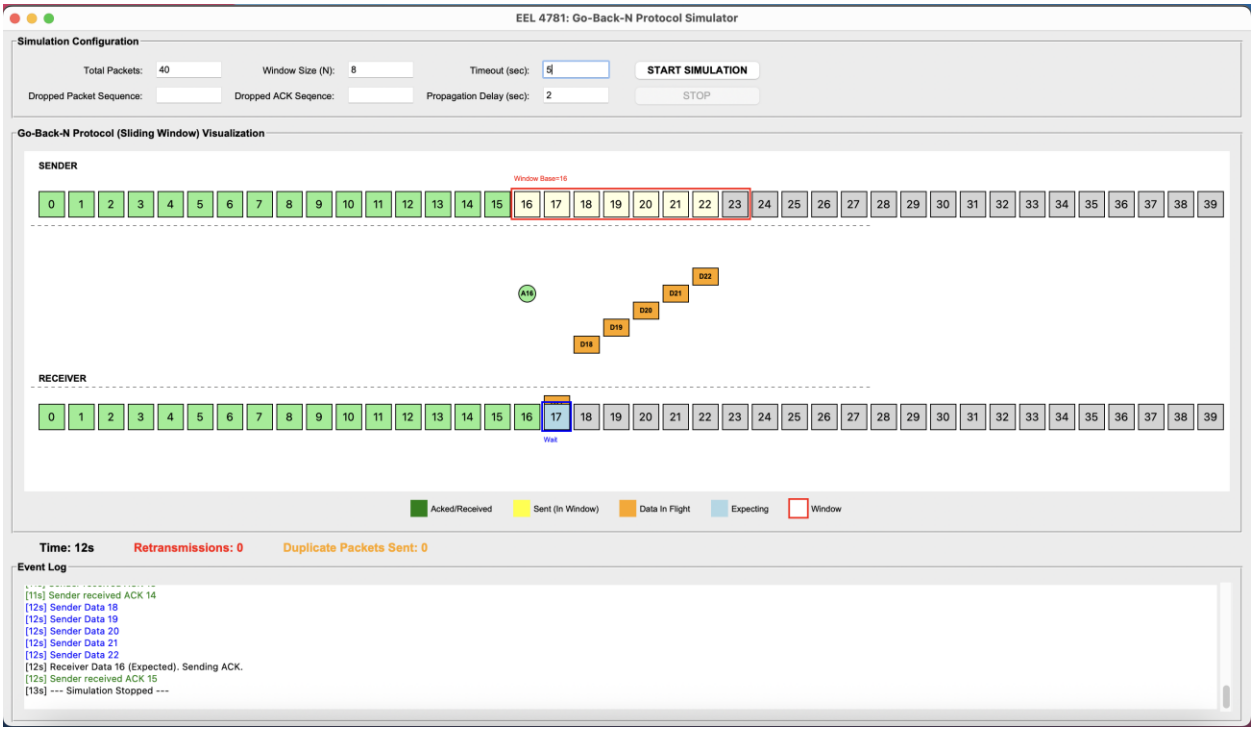
At the start of the visualization w/ ideal conditions, no packets dropped, short propagation delay etc. The “packets” fall vertically one at a time in order to the receiver.



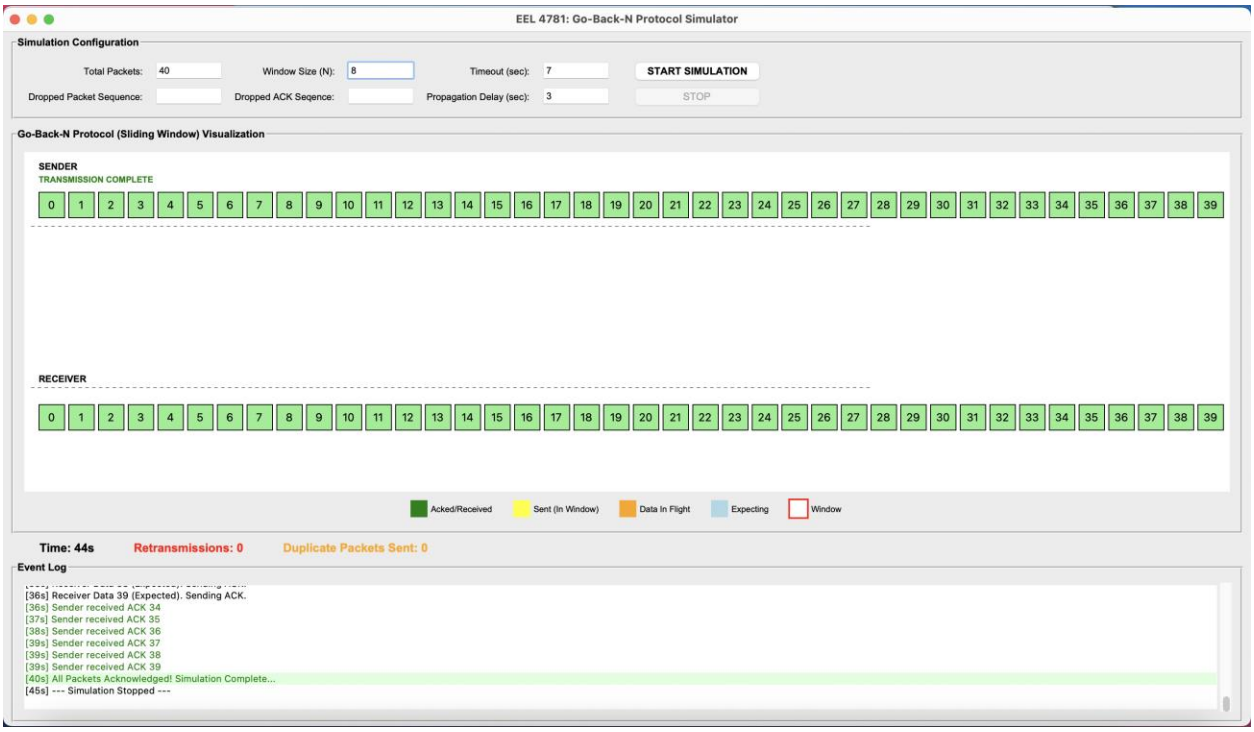
The receiver sends ACK responses to the sender.



Here you can see the “sliding window” about to shift as ACKs arrive.



And finally, all packets and ACKs received, simulation complete. Time = 44s, Retransmissions = 0, Duplicate Packets Sent = 0

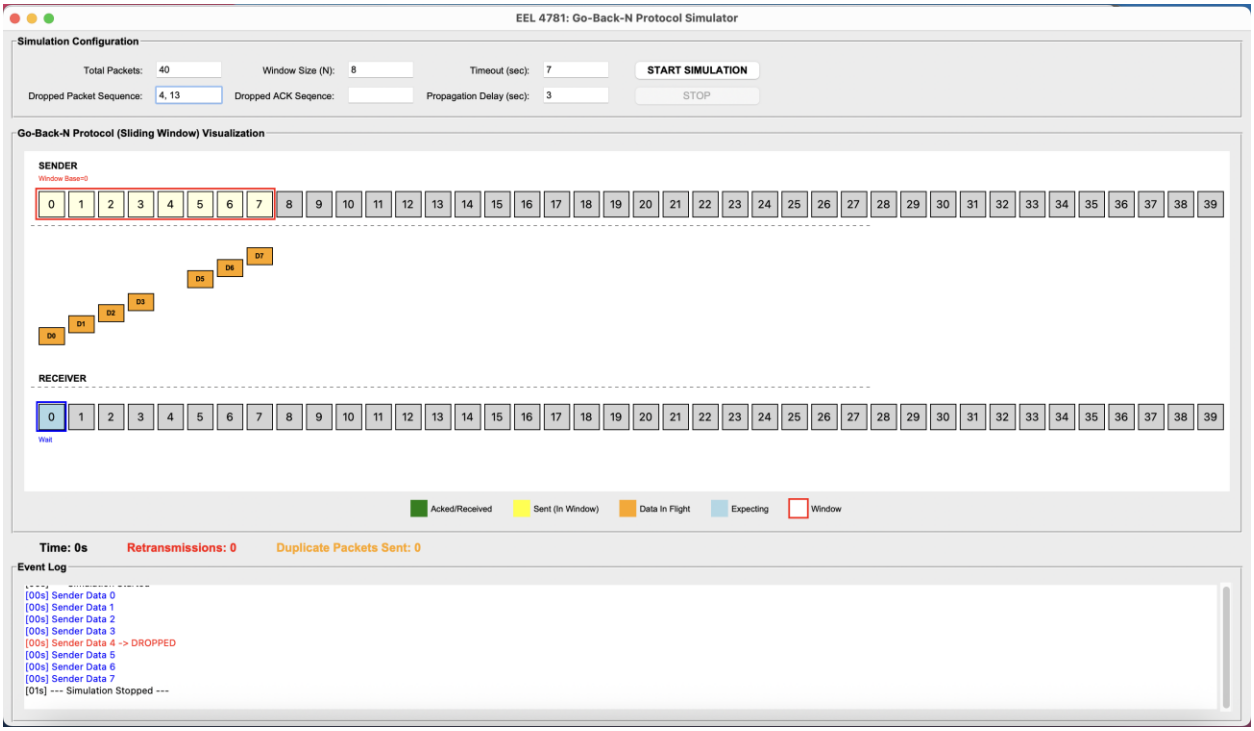


Scenario B: Data Packet Loss

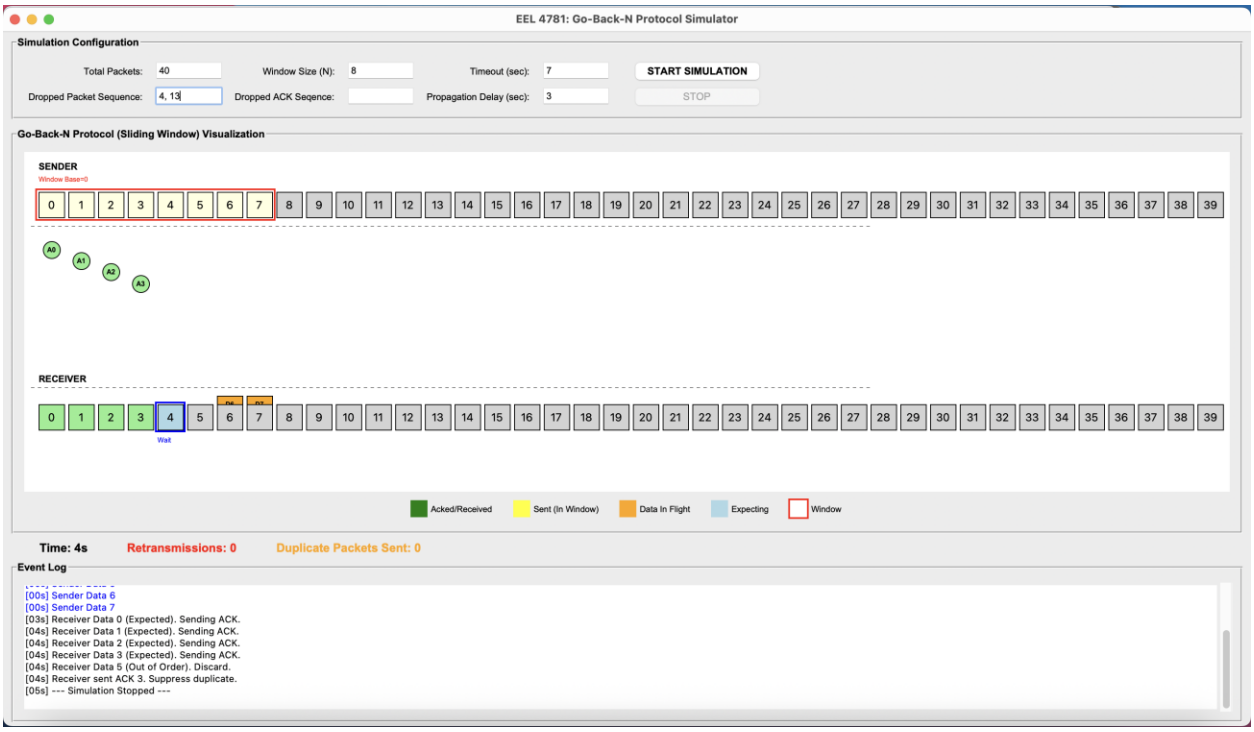
To test error recovery, the sequence I chose (4, 13) was random and is configurable by the user in the “Dropped Packages Sequence” input field. The logs showed that while the receiver successfully acknowledged earlier packets, the loss of a specific packet caused the receiver to discard subsequent valid packets, as they were considered out of order. The receiver repeatedly sent acknowledgments for the last

successfully received packet. Once the sender's timer expired, the simulation correctly showed the retransmission of the entire window, highlighting the bandwidth inefficiency inherent in GBN during lossy conditions.

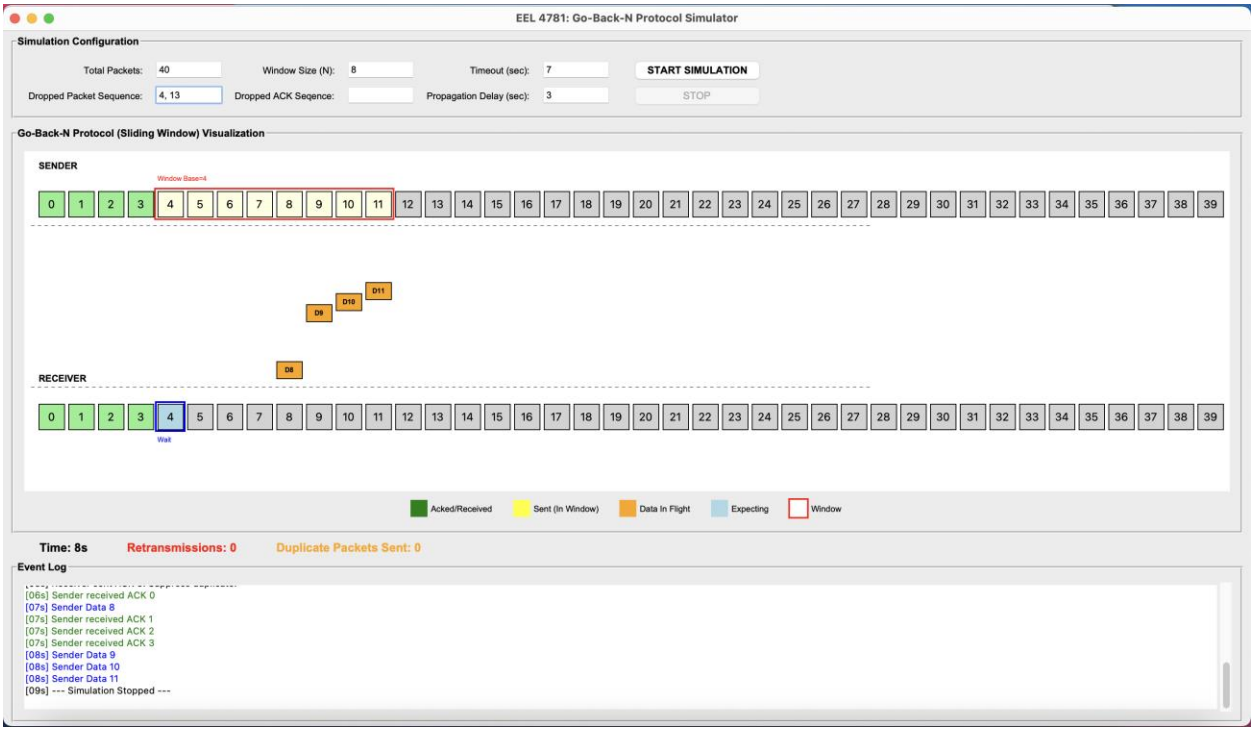
As the simulation begins; packet 4 is not visible due to it being designated as the dropped packet.



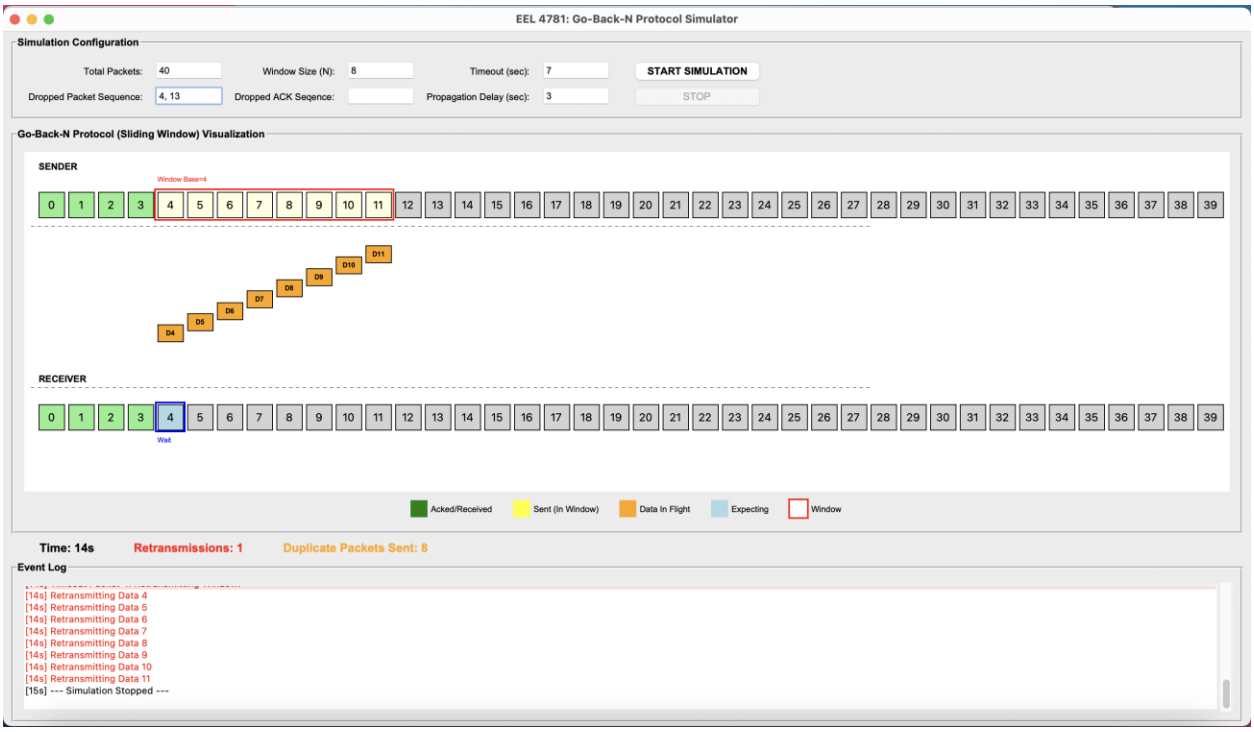
ACKs 0, 1, 2, 3 being sent right before retransmission occurs. Remember that packet 4 was dropped. You can also see out of order packets 6 and 7 arriving



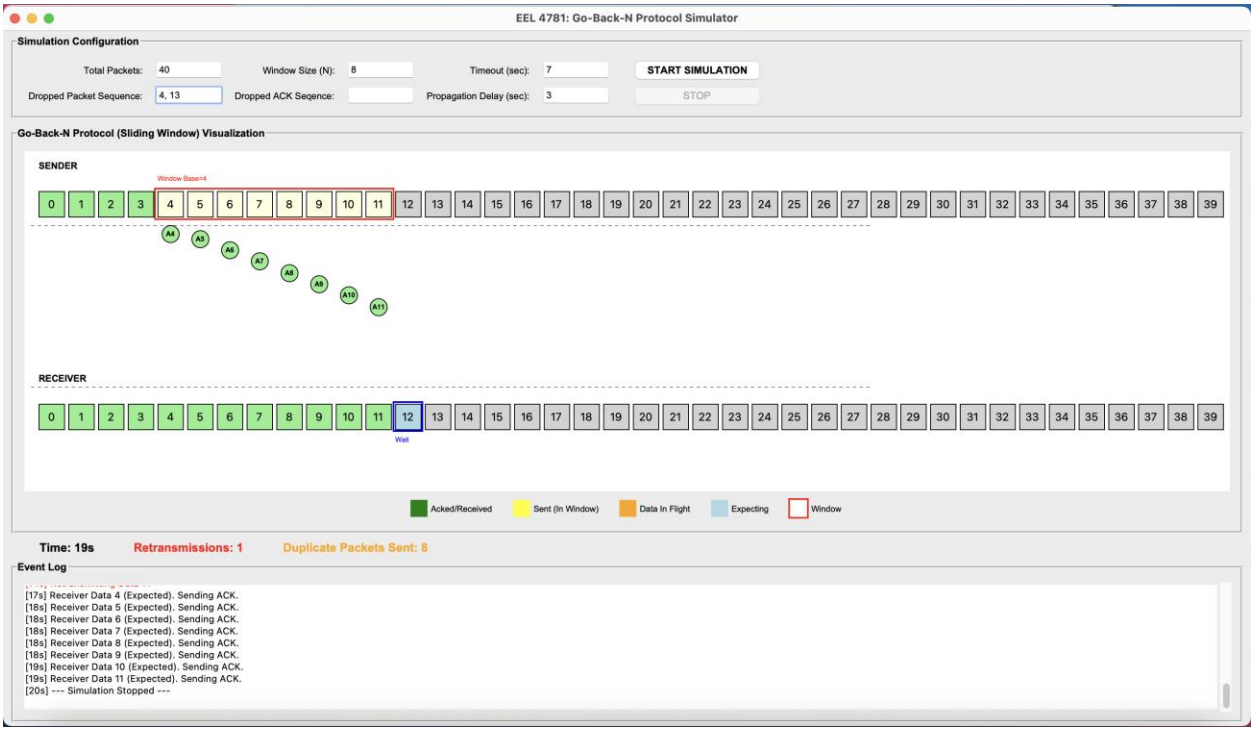
Window shifts, packets 8, 9, 10 and 11 sent.



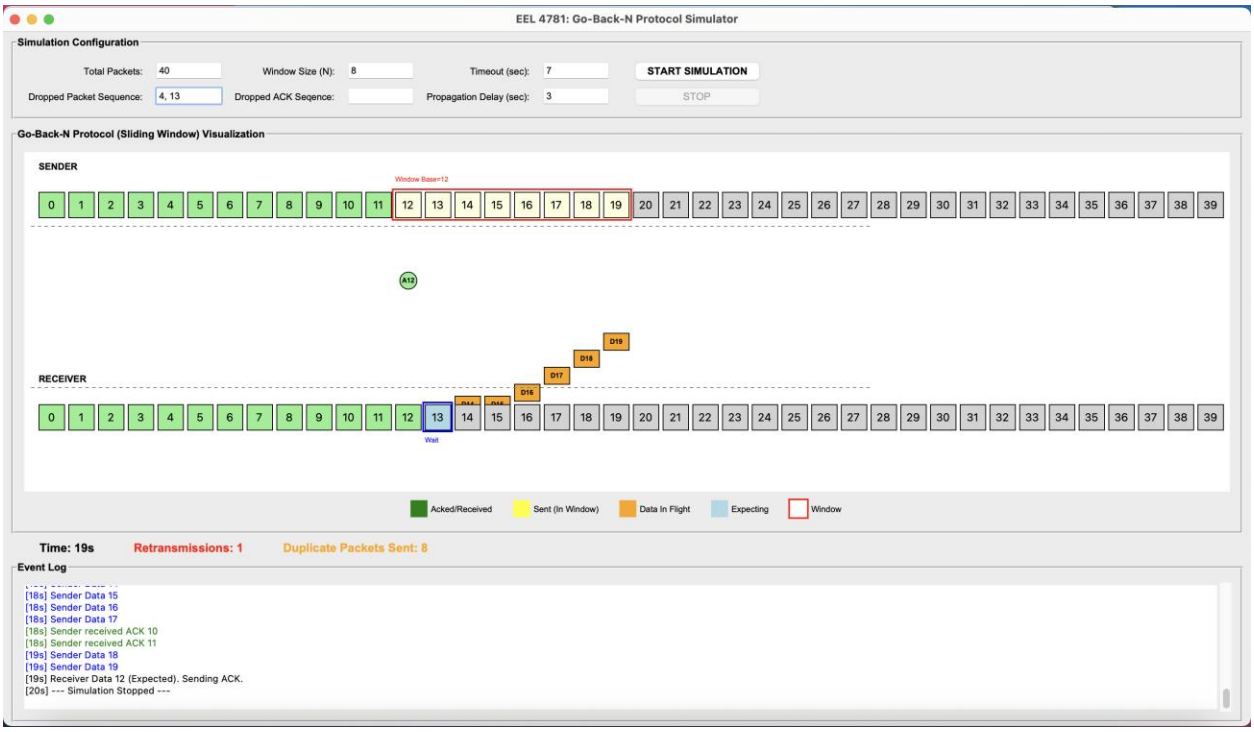
Out of order packets from above, trigger a retransmission of the entire shifted window to retain data integrity.



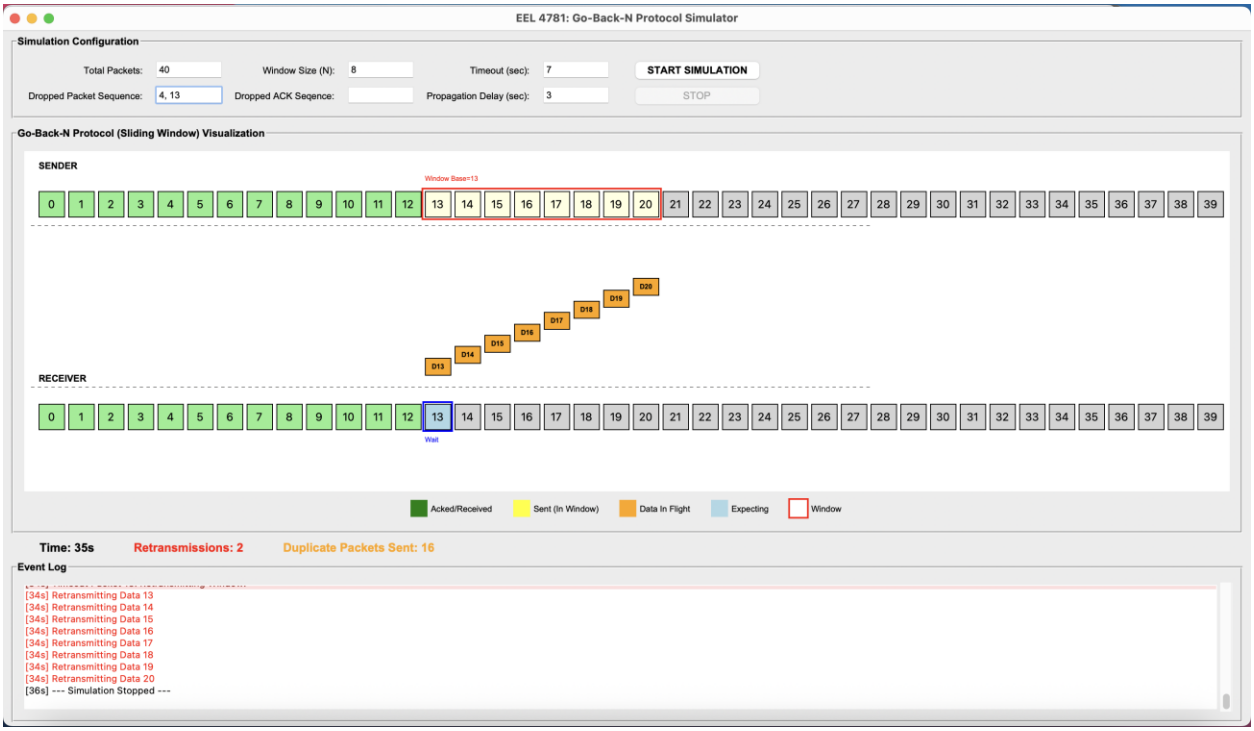
ACKs sent for retransmitted lost packet; transmission resumes normally.



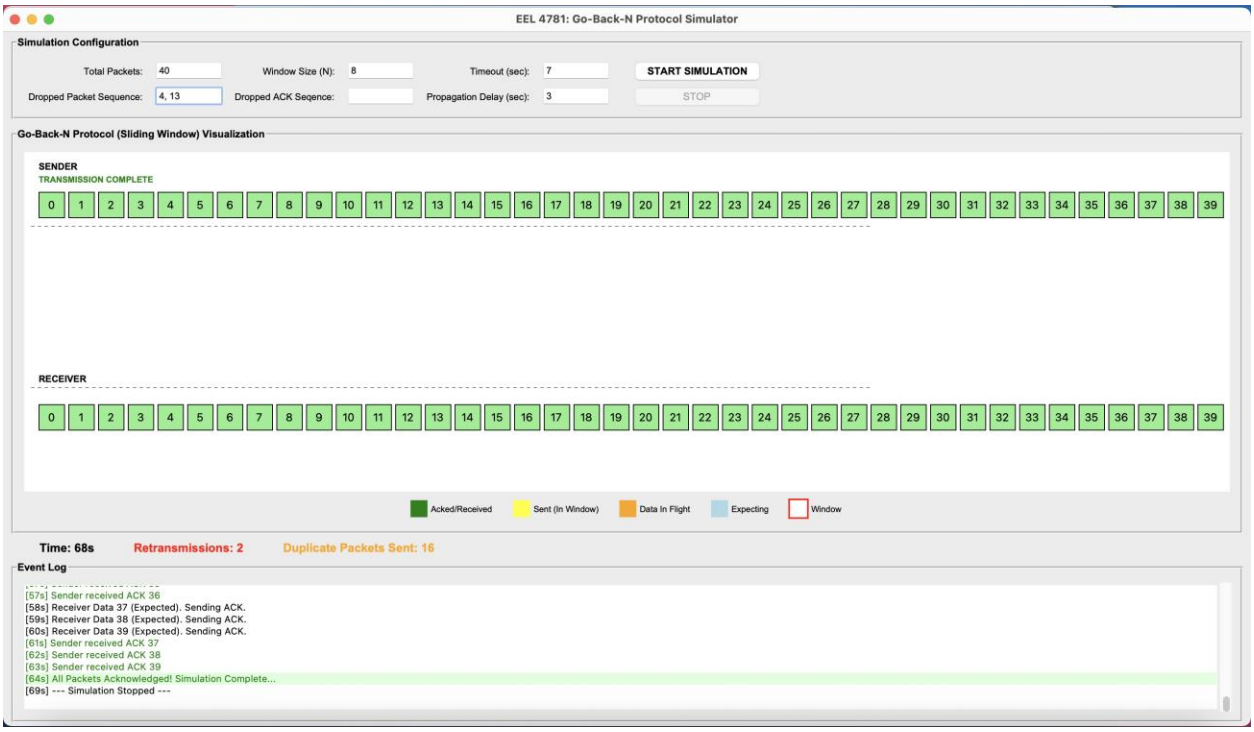
Window shifts, now packet 13 is lost, and the same pattern will repeat.



Out of order packets timeout, retransmission, ACKs sent back to sender. This will occur for any dropped packets during the simulation.



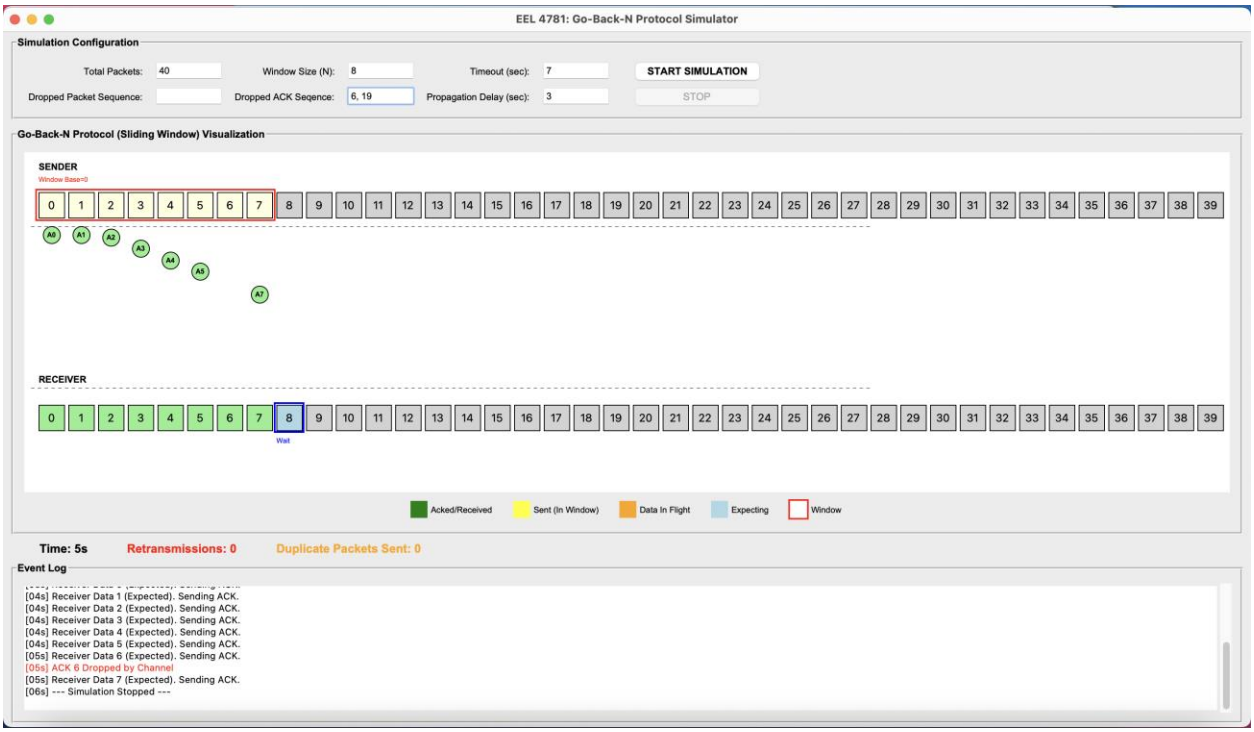
Simulation Complete: Time = 68s, Retransmissions = 2, Duplicate Packets Sent = 16


EEL 4781: Go-Back-N Protocol Simulator
Simulation Configuration
Total Packets: 40
Window Size (N): 8
Timeout (sec): 7
START SIMULATION
Dropped Packet Sequence: 4, 13
Dropped ACK Seqence:
Propagation Delay (sec): 3
STOP
Go-Back-N Protocol (Sliding Window) Visualization
SENDER
TRANSMISSION COMPLETE
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
RECEIVER
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Acked/Received
Sent (In Window)
Data In Flight
Expecting
Window
Time: 68s
Retransmissions: 2
Duplicate Packets Sent: 16
Event Log
[57s] Sender received ACK 36
[58s] Receiver Data 37 (Expected). Sending ACK.
[59s] Receiver Data 38 (Expected). Sending ACK.
[60s] Receiver Data 39 (Expected). Sending ACK.
[61s] Sender received ACK 37
[62s] Sender received ACK 38
[63s] Sender received ACK 39
[64s] All Packets Acknowledged! Simulation Complete...
[69s] --- Simulation Stopped ---

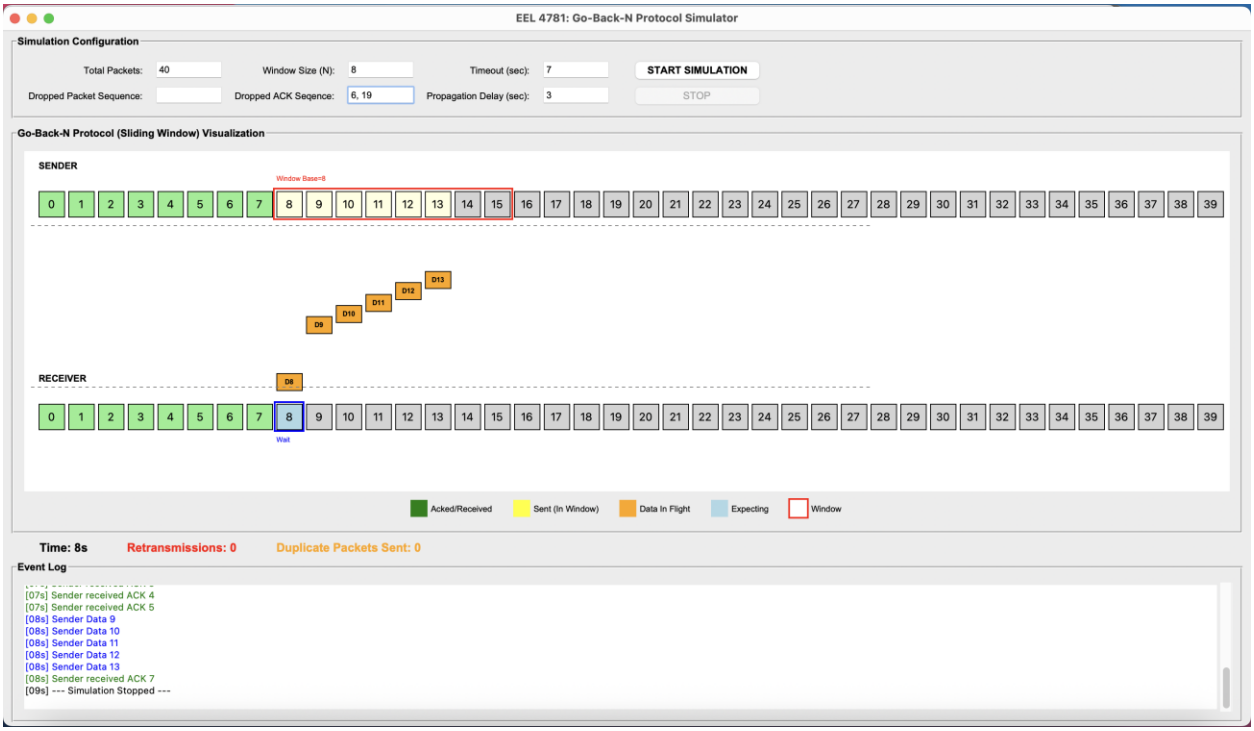
Scenario C: Acknowledgment Loss

I also tested the robustness of the cumulative acknowledgment mechanism by dropping specific ACK packets (6, 19). The simulation showed that even when an intermediate ACK was lost, the arrival of a subsequent ACK allowed the sender to slide the window forward past the missing acknowledgment. This confirmed that the cumulative nature of ACKs in GBN effectively covers lost acknowledgments without triggering unnecessary retransmissions.

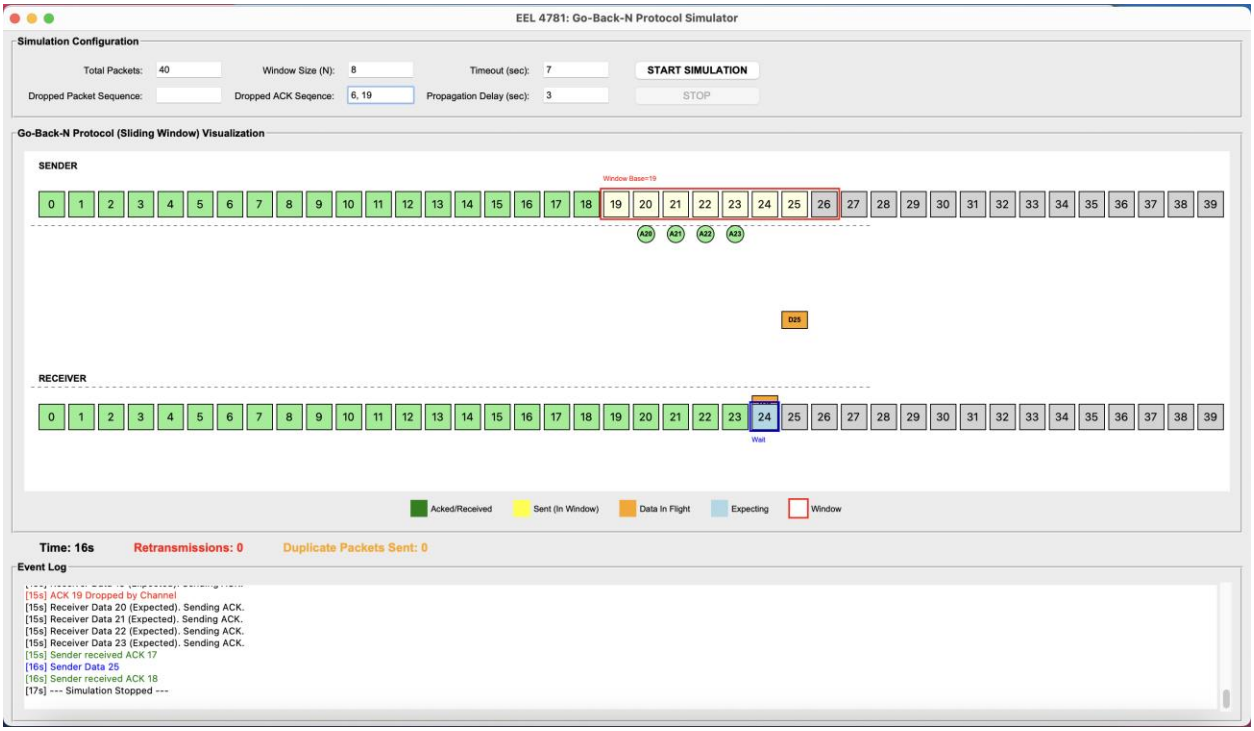
ACK 6 dropped.



Even though ACK 6 is dropped, the window still shifts because ACK 7 was received. This is due to the Cumulative ACK rule feature of GBN. Out of order ACKs do NOT timeout like out of order packets do.

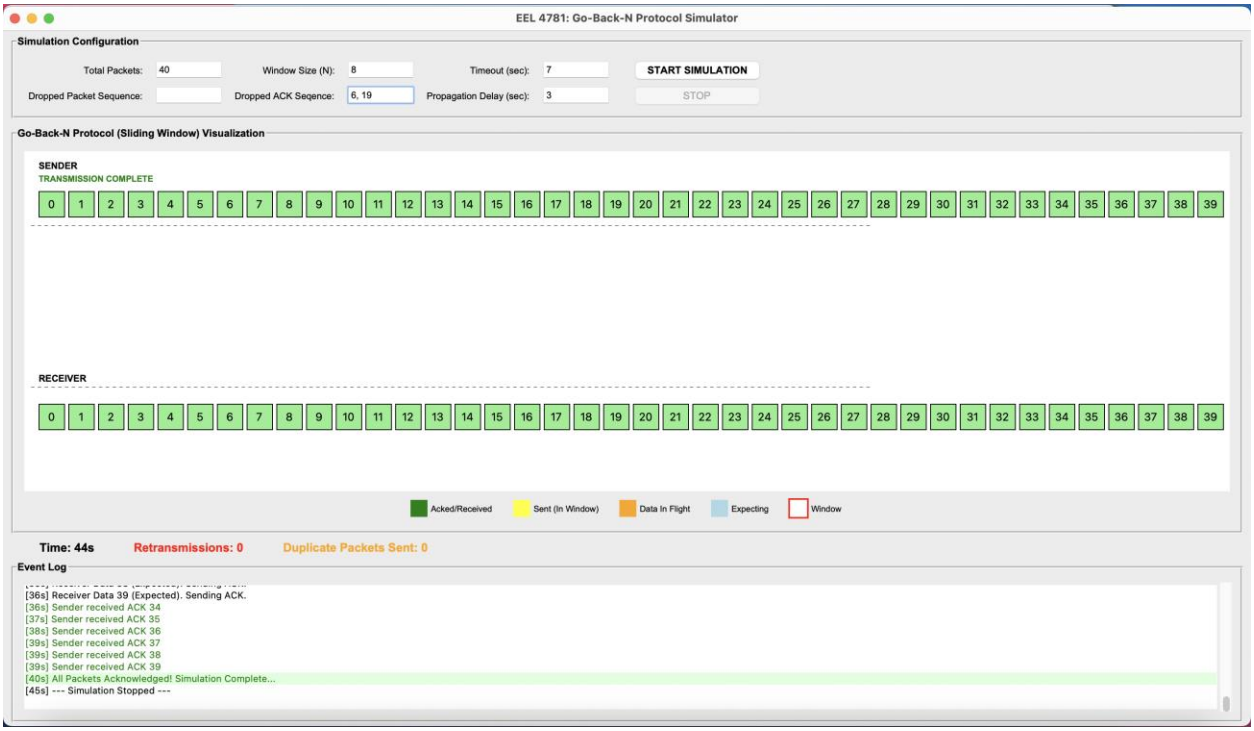


Window about to shift even though ACK 19 was dropped



Window shifts to ACK 23, the most recent ACK received

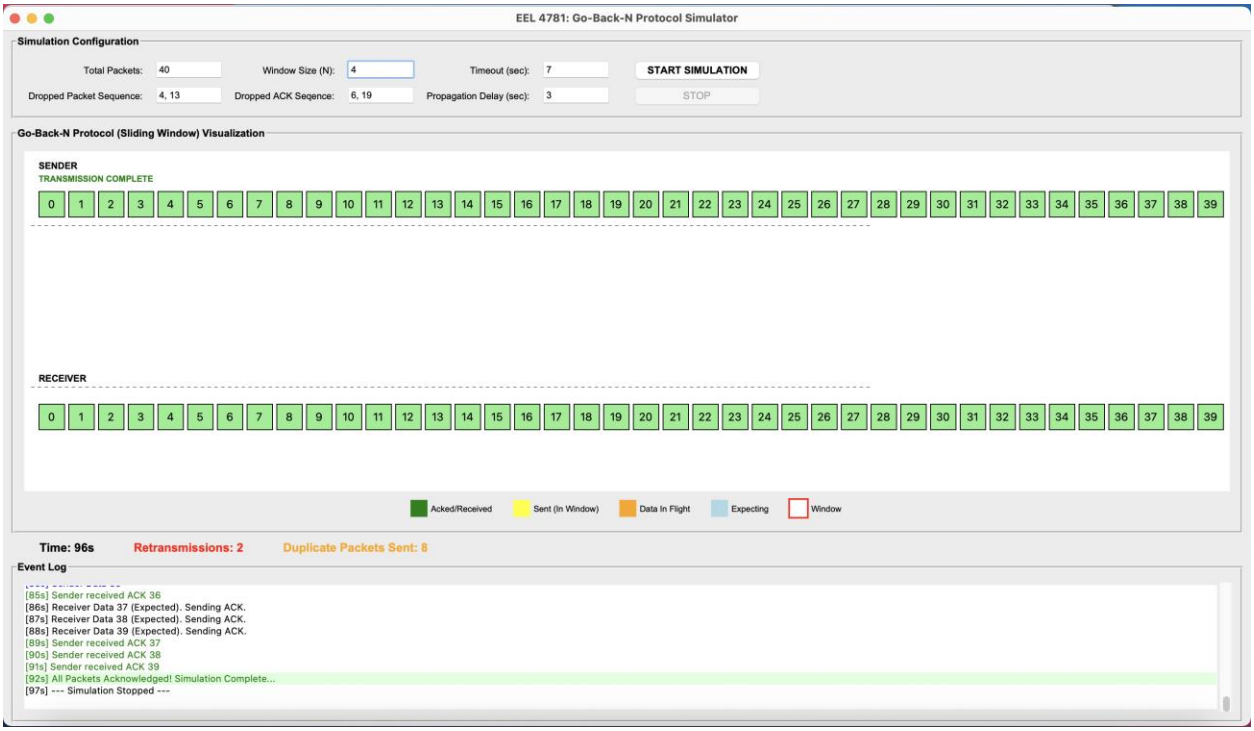
Simulation Complete: Time = 44s, Retransmissions = 0, Duplicate Packets Sent = 0



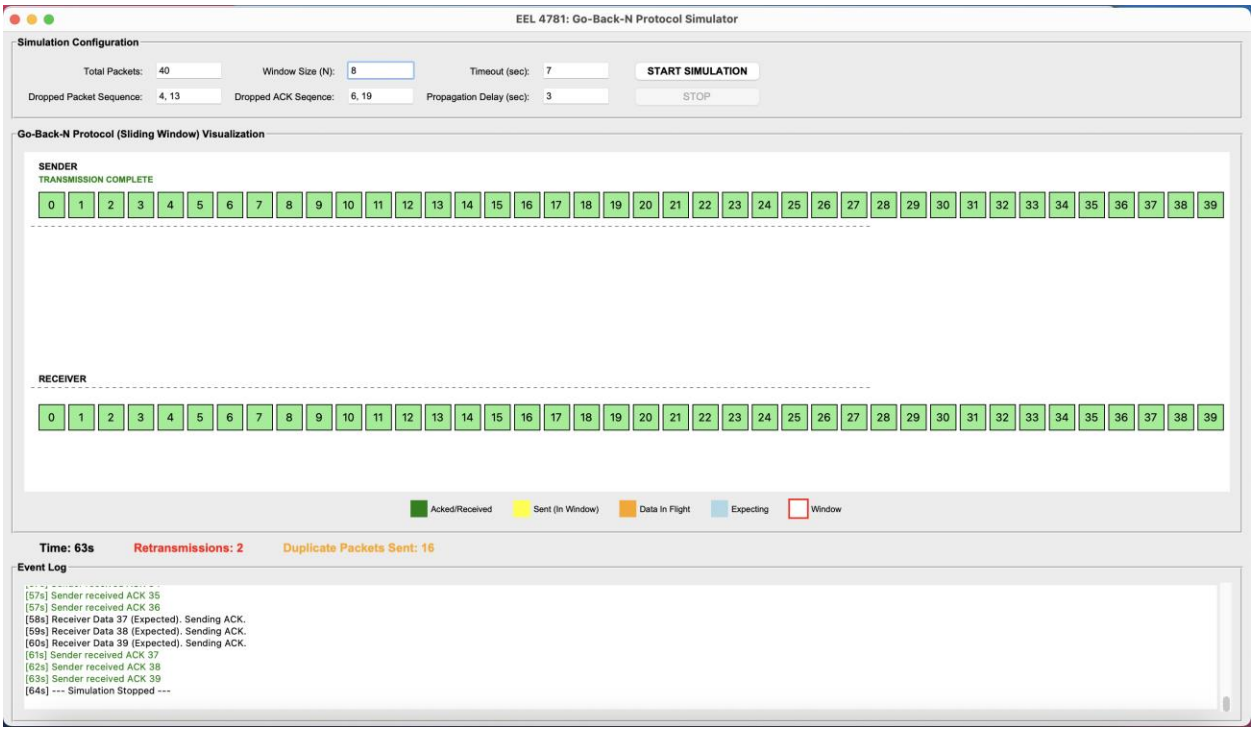
As you can see the dropped ACKs had no effect on the performance; both the ideal conditions simulation and the dropped ACKs simulation finished at the identical time of 44 seconds.

Scenario D: Comparing Varying Window Sizes w/ Combined standardized Packet and ACK loss.

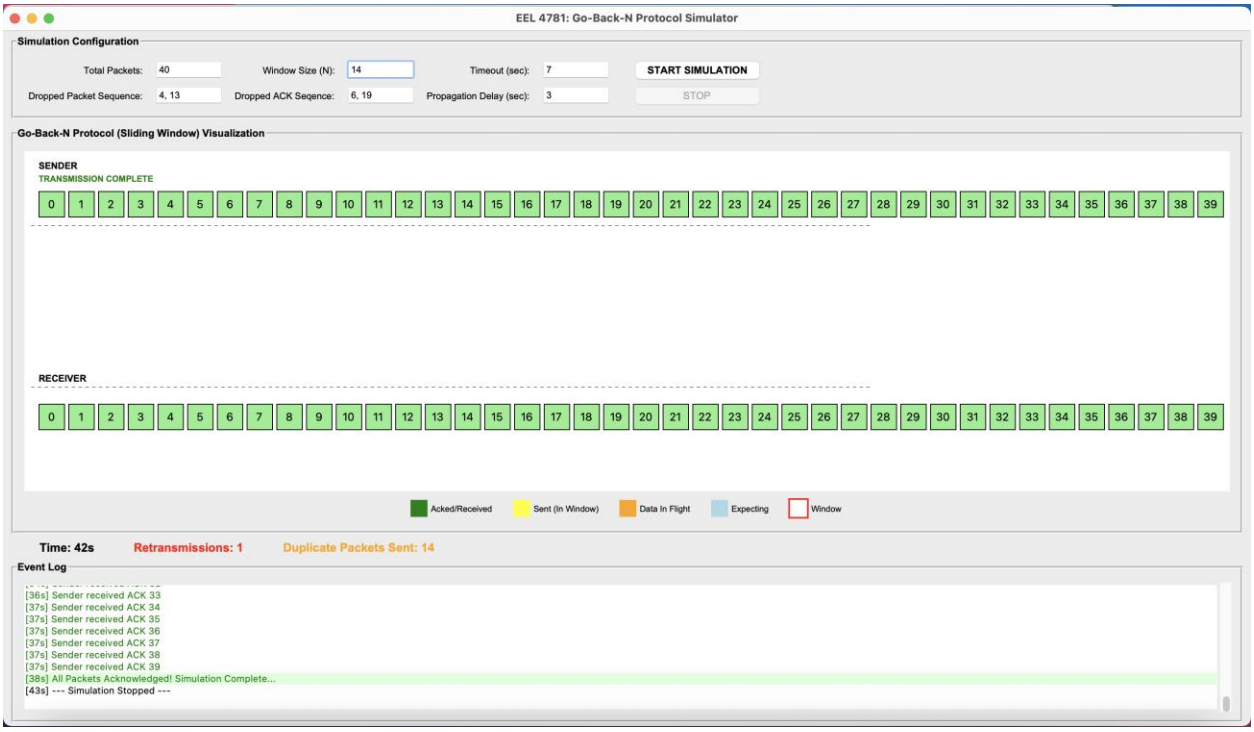
Window Size = 4, Dropped Packages Sequence = 4, 13, Dropped ACKs Sequence = 6, 19. Time = 96s, Retransmissions = 2, Duplicate packets sent = 8



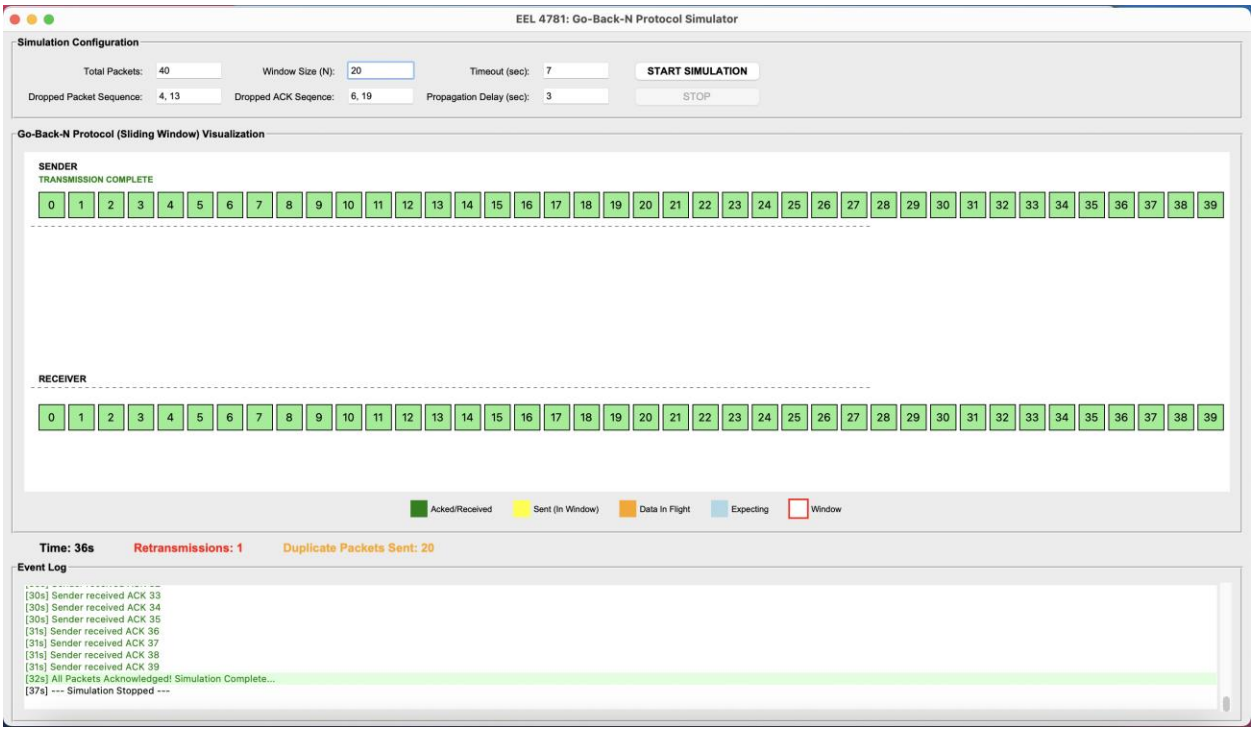
Window Size = 8, Dropped Packages Sequence = 4, 13, Dropped ACKs Sequence = 6, 19, Time = 63s, Retransmissions = 2, Duplicate packets sent = 16



Window Size = 14, Dropped Packages Sequence = 4, 13, Dropped ACKs Sequence = 6, 19. Time = 42s, Retransmissions = 1, Duplicate packets sent = 14



Window Size = 20, Dropped Packages Sequence = 4, 13, Dropped ACKs Sequence = 6, 19. Time = 36s, Retransmissions = 1, Duplicate packets sent = 20



Comparison of varying window sizes w/ constant dropped packets and ACKs:

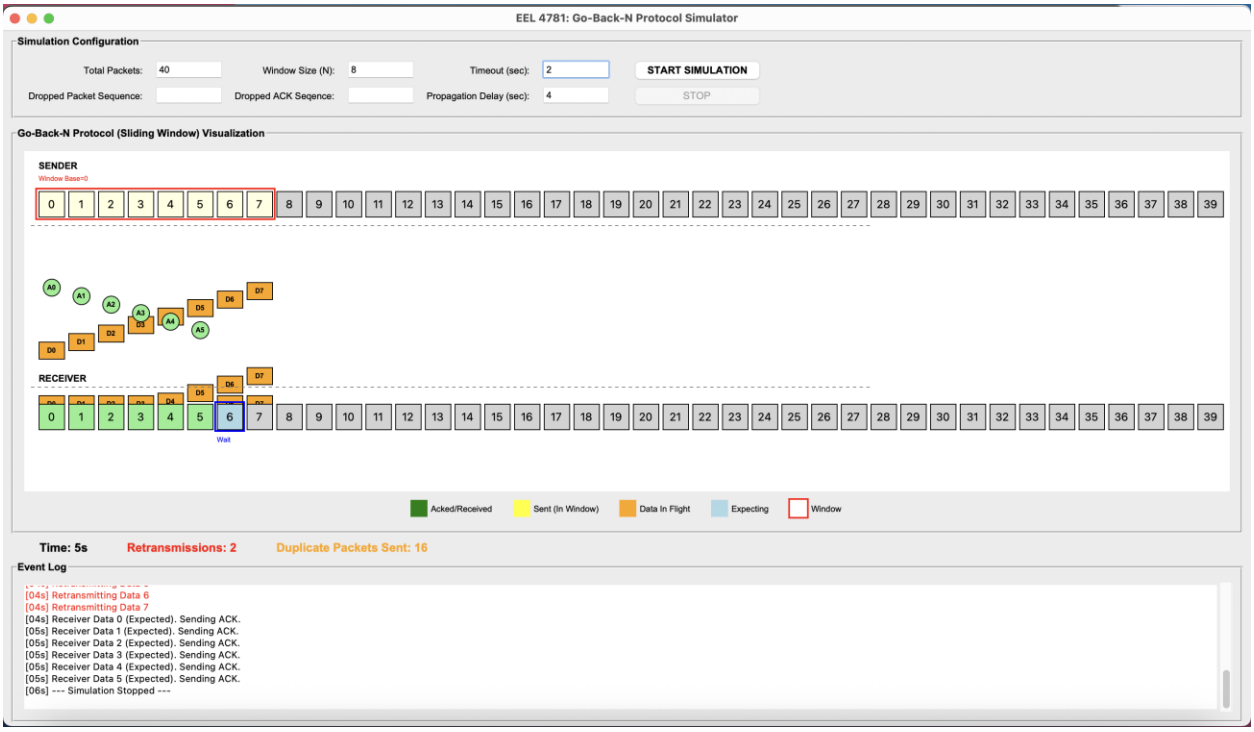
Window Size	Time (seconds)	Retransmissions	Duplicate Packets
4	96	2	8

8	63	2	16
14	42	1	14
20	36	1	20

Varying the window size really highlights the tradeoffs between speed and waste. As the window size decreases, your performance suffers, but you have fewer duplicate packets sent. On the other hand, increasing the window size increase performance significantly but you end up with many wasted duplicate packets. Finding the optimal window size that is large enough to keep the network pipeline full (high throughput) but small enough to limit the amount of unnecessary data repeated after an error (high efficiency).

Scenario E: Premature Timeout (Propagation delay > Timeout)

Here I just wanted to briefly show how having a propagation delay that is higher than the timeout phase causes massive network congestion. The ACKs don't have enough time to be received before the sender retransmits the data resulting in a good amount of waste.



Conclusion

I believe this simulation successfully demonstrates the internal mechanics and behavior of the Go-Back-N protocol. The graphical interface provides an intuitive visualization of the sliding window concept, allowing users to see how the window state reacts to successful transmissions and error conditions in real-time. The

inclusion of a duplicate packet counter further emphasizes the trade-off between the implementation simplicity of Go-Back-N and its potential inefficiency in high-error environments compared to more complex protocols like Selective Repeat. The project confirms that while GBN is robust and reliable, accurate timeout calibration is essential to prevent premature timeouts and unnecessary network congestion.