Ryan Markowitz

EEL 4781

Go Back N Simulation Project

Section 1 - Introduction and Protocol Choice

For this project, I chose to simulate the Go Back N protocol using Python and the PySide6 GUI framework. I picked Go Back N because it fit the animation idea I had from the start. I wanted something I could turn into a clear visual representation, where packets move across the screen, the window slides, and you can actually watch the behavior of the protocol in real time. Go Back N is simple enough to animate cleanly, but it still has all the important mechanics of a reliable data transfer protocol.

Go Back N uses a sliding window on the sender side. The sender is allowed to send up to N unacknowledged packets, and the receiver only accepts packets in order. If the sender does not get an ACK in time for the oldest unacknowledged packet, the sender will retransmit every packet starting from that one. The receiver sends cumulative acknowledgements, so when it sends ACK k, it is confirming it has successfully received all packets up to k. Even though this approach is not the most efficient, it is very easy to understand and it makes it obvious what is happening when something goes wrong.

The goal of my project was to build a graphical simulation that shows how Go Back N works under different conditions. I created a sender and receiver area, animated the packets traveling across the screen, and added settings for window size, propagation delay, and manual packet or ACK loss. When you run the simulation, you can see the sender sliding its window, the receiver sending acknowledgements, and how the protocol reacts to lost packets or long delays. Overall, the simulation lets you watch the entire process play out in a way that is easy to follow.

Section 2 - Implementation Overview

I built the entire simulation in Python using the PySide6 framework. I also used QtDesigner to design the UI layouts and generate the .ui files that my code could import. I kept the project organized by splitting everything into two main folders. The ui folder holds all the translated UI files, and the widget_containers folder holds all the Python widget classes where I added all the logic, signals, and animations.

QtDesigner handled the visual layout for me, which made it much easier to build panels with consistent spacing and responsive layouts. Each UI component was saved as its own widget, and I used a VS Code extension to convert each .ui file into a Python file. After that, the real work happened inside the widget classes. I imported the UI files and wrote the logic that allowed each part of the simulation to communicate with the rest of the system.

After generating the Python versions of the .ui files, I wrote the actual behavior in the widget_containers folder. For each UI file, there is a corresponding Python class that imports it and wires up the logic. The settings widget class connects slider changes and spin boxes to signals so the rest of the application can react when the user changes the configuration. The play and reset widget class handles the Play button toggling between Play and Pause and sends signals when the simulation should start, pause, or reset.

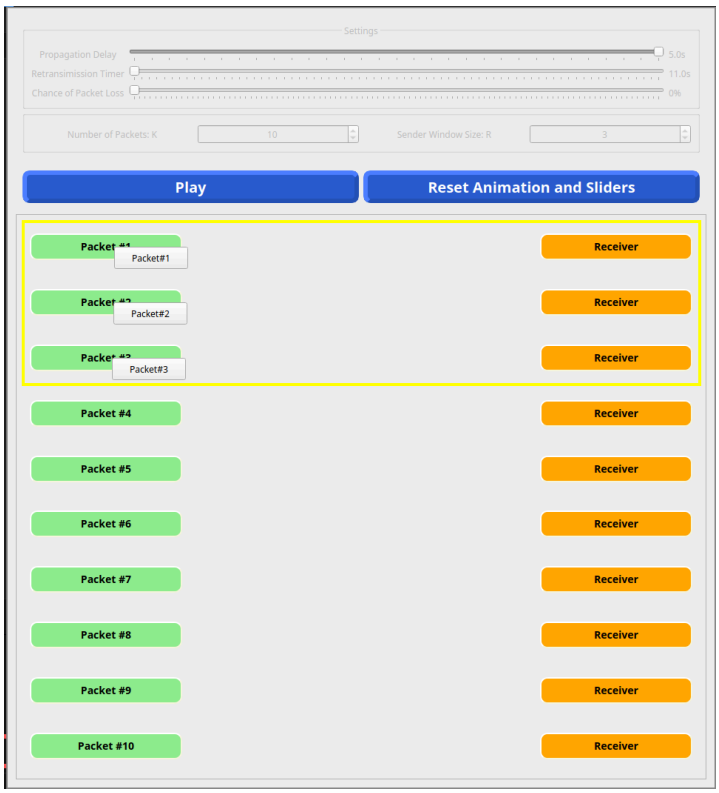
I also created a separate Packet widget. This is a custom QPushButton that represents an individual packet on the screen. It has logic for what happens when it is clicked, such as “dying” if the user wants to simulate a lost packet, and it has simple fading behavior to make the disappearance look smoother. The Packet widget itself does not decide when to move. It only defines what a packet looks like and how it behaves when something happens to it.
The sender receiver widget class uses the static sender and receiver buttons from the UI to figure out where packets and ACKs should start and end. It does not store two packet buttons. Instead, it keeps track of the current sender number, which you can think of as the sequence number that will be sent next when that row is active. When it needs to animate a packet, it creates a Packet widget, places it at the sender position, and then animates it across the screen toward the receiver. It performs a similar process to animate ACKs moving back in the opposite direction. This class is focused on getting the geometry of the two endpoints and handling the visual movement between them.

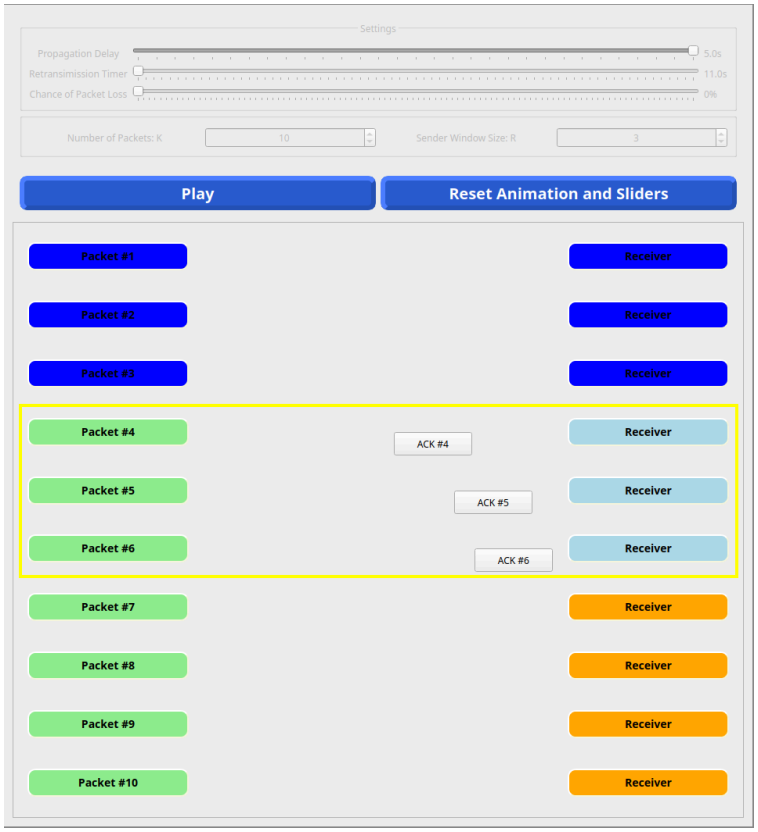
Above that, I have the sender receiver panel. This is where the main Go Back N logic lives. The panel creates and manages multiple sender receiver rows and keeps track of things like the total number of packets, the current base of the window, the next packet to send, and the current window size. It decides when to send a new packet, when to slide the window forward based on cumulative ACKs, and when to trigger retransmissions if something times out or if the protocol detects a loss. It also draws and updates the sliding window highlight so you can see which rows are currently inside the sender’s window.

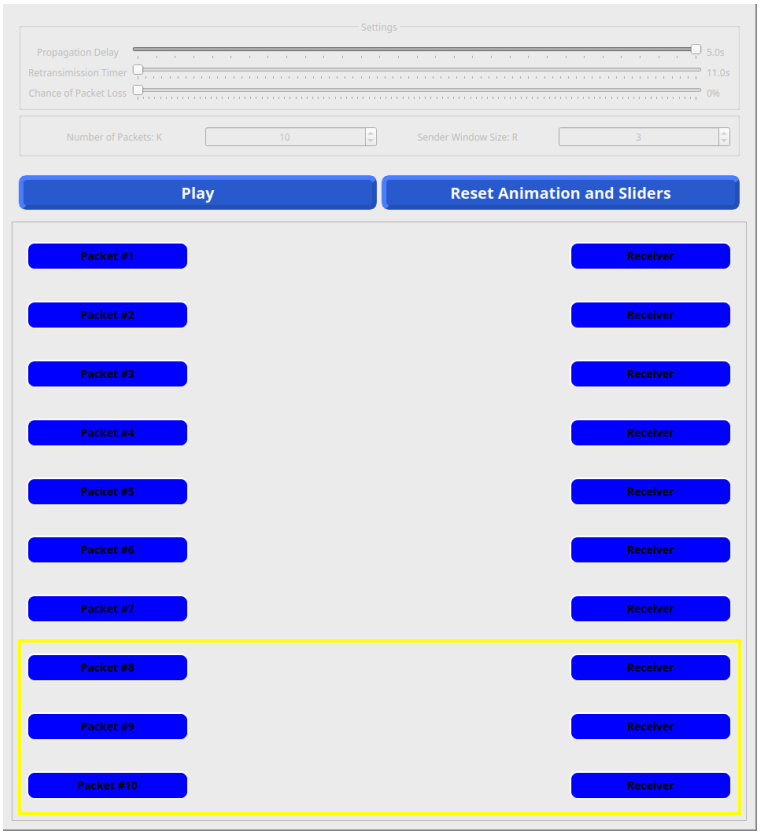
At the top level, the main widget acts like the glue that brings everything together. It places the settings widget, the scrollable sender receiver panel, and the play and reset widget into a vertical layout. It connects signals from the settings widget to the sender receiver panel so that changes in parameters immediately affect the simulation. It also connects the play and reset signals to start, pause, or reset the animation and protocol state across the whole application. In the end, main.py just creates this main widget and shows it, and the rest of the behavior is handled through the interaction between these different components.

Section 3 - Observations From Running the Simulation

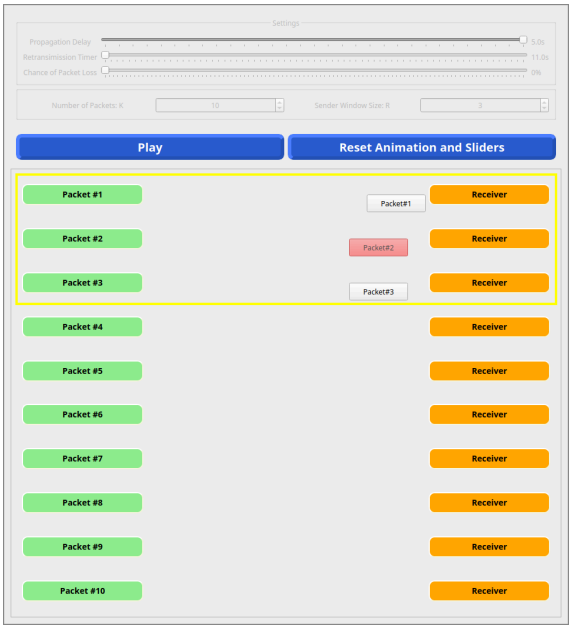
I tested the simulation under several different conditions to see how Go Back N behaves and to make sure the protocol logic was working correctly. When everything is set to normal, meaning no loss and a reasonable propagation delay, the sender quickly fills its window and sends packets one after another. As ACKs come back, the window slides forward and the sender keeps sending the next packets in order. This is the cleanest run and it shows how efficient Go Back N can be when nothing goes wrong.

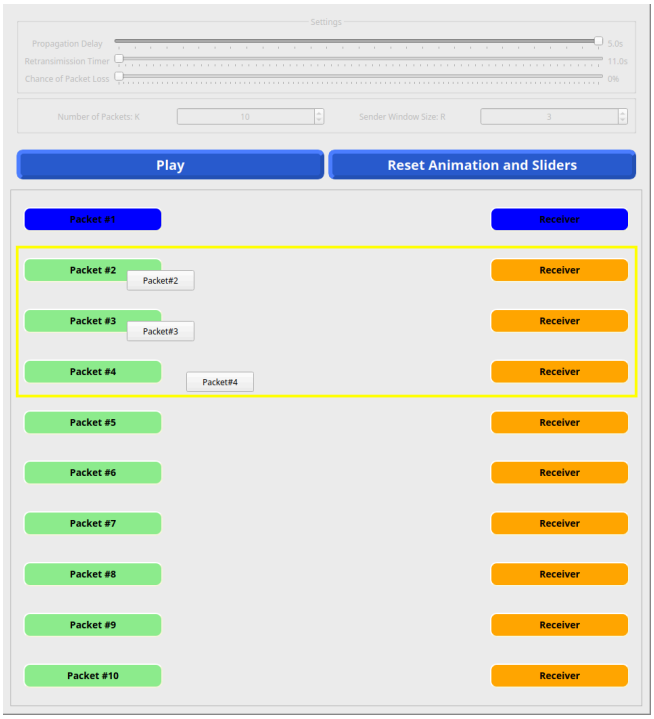



Settings
Propagation Delay
5.0s
Retransmission Timer
11.0s
Chance of Packet Loss
0%
Number of Packets: K
10
Sender Window Size: R
3
Play
Reset Animation and Sliders
Packet #1
Receiver
Packet #2
Receiver
Packet #3
Receiver
Packet #4
ACK #4
Receiver
Packet #5
ACK #5
Receiver
Packet #6
ACK #6
Receiver
Packet #7
Receiver
Packet #8
Receiver
Packet #9
Receiver
Packet #10
Receiver


Settings
Propagation Delay
5.0s
Retransmission Timer
11.0s
Chance of Packet Loss
0%
Number of Packets: K
10
Sender Window Size: R
3
Play
Reset Animation and Sliders
Packet #1
Receiver
Packet #2
Receiver
Packet #3
Receiver
Packet #4
Receiver
Packet #5
Receiver
Packet #6
Receiver
Packet #7
Receiver
Packet #8
Receiver
Packet #9
Receiver
Packet #10
Receiver

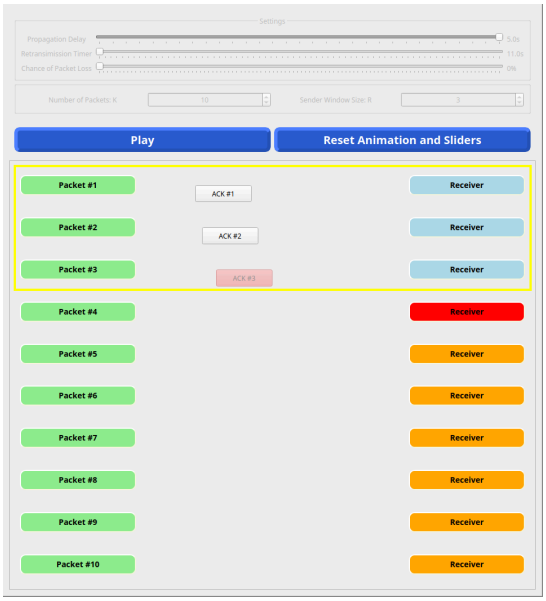
When I manually forced packet loss, the Go Back N behavior became more obvious. If a packet inside the window is lost, the receiver stops advancing and keeps sending back the same cumulative ACK for the last in-order packet it received. The sender does not realize something is wrong until the retransmission timer expires. Once the timer fires, all unacknowledged packets starting from the lost one are retransmitted. You can clearly see the sudden burst of retransmissions because the panel animates them again from the sender to the receiver.

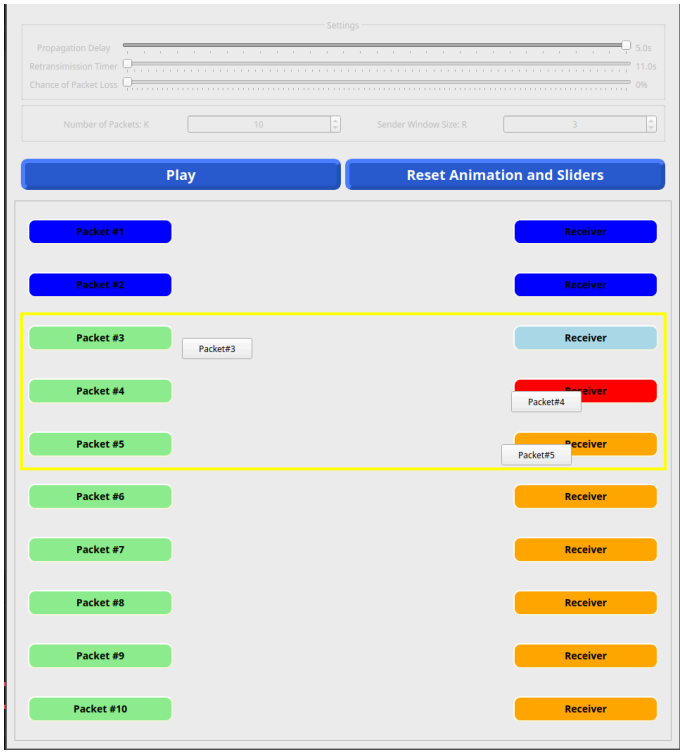




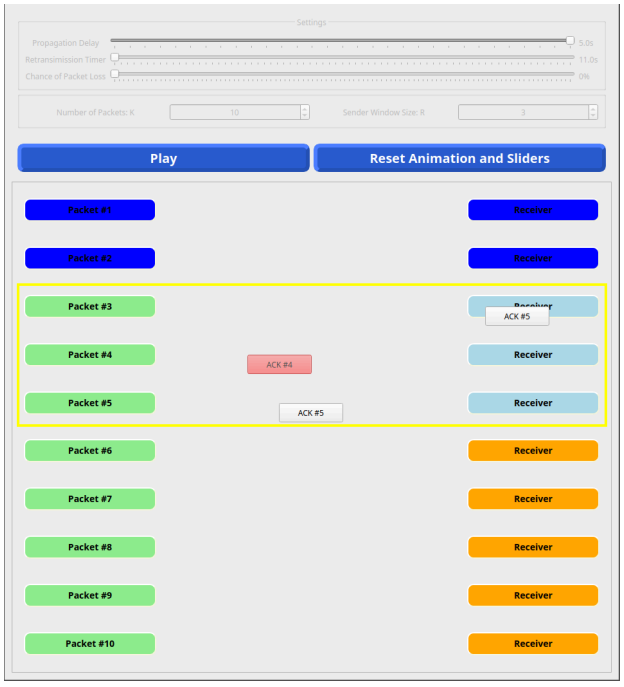
* Window slid to packet 4 after 1 was ACKed and sent packet 4. After retransmission timer signal was sent, packets 2 and 3 were sent again. This perfectly mimics how Go Back N works when the receiver receives out of order packets.

I also tested ACK loss. Losing an ACK does not stop the receiver but it keeps the sender from knowing it can advance the window. The sender keeps sending packets as long as it is still within the window limit, but eventually it gets stuck waiting for an ACK that will never arrive. Once the timer expires, it triggers a full retransmission of the remaining packets, which fixes the problem and slides the window again. However, If a higher ACK is received before the retransmission, it will still slide the window. This is because if a higher ACK is received that means the previous packet was definitely received, so the sender doesn't have to receive that packet. I will demonstrate both happening below:

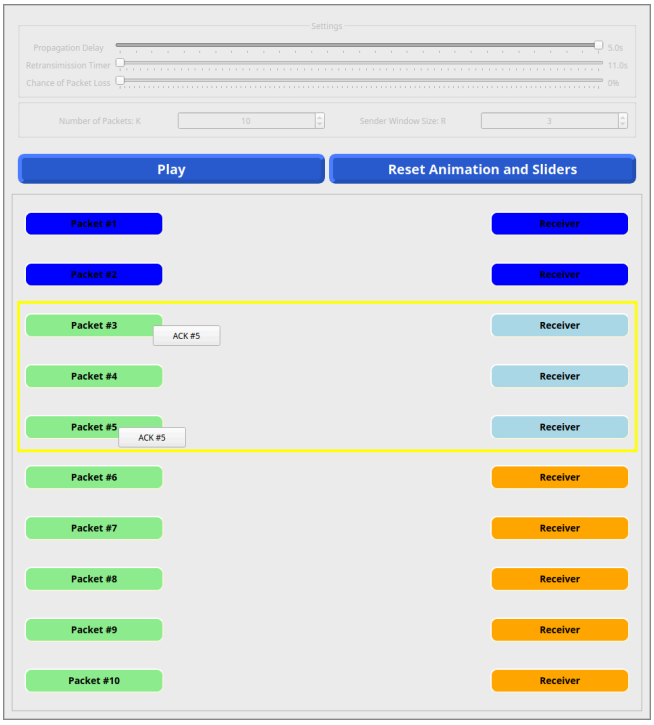




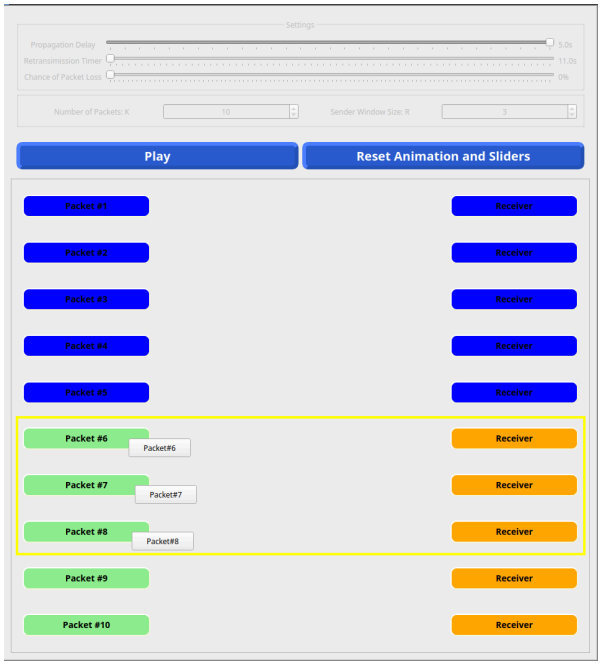
* Packet 3's ACK was not received, no higher ACK was received before the retransmission signal, so packet 3 was sent again, and the window slid down to 5.



* ACK for packet 4 was dropped



* When packet 3 arrived again, packets 1-5 have already been received which is why it is sending back ACK #5 to both the sender slots. As soon as ACK #5 is sent back, the window will slide to 6-8 even though the sender never received ACK #4.



* As expected, when ACK #5 was received the window slid to 6-8. ACK #4 didn't need to be received by the sender. This demonstrates that ACK loss is tolerable, but packet loss is not.

Section 4 - How the Simulation Demonstrates Go Back N

My simulation meets all the key requirements of the Go Back N protocol, and the visual layout makes each part easy to see. The sender window is shown on screen so you can always tell which packets are allowed to be sent at any given moment. The sender only keeps one retransmission timer, which follows the real Go Back N design. The timer is tied to the oldest unacknowledged packet, and once it expires the sender retransmits all packets in the current window starting from that base.

The simulation also uses cumulative ACKs exactly like Go Back N. The receiver only accepts packets that arrive in order. If a packet shows up out of order, the receiver ignores it and sends back the same cumulative ACK for the last correct packet it has. This behavior is visible in the animation. When packet n is lost, the receiver keeps sending back ACK n-1 no matter how many times packet n+1 or higher arrive, which is the correct Go Back N response.

I also included manual packet and ACK loss so I could show why retransmissions happen. If the user clicks a packet to “kill” it, the system treats it as lost and the sender will eventually resend everything in the window. ACK loss works the same way. When the ACK for the newest in-order packet never arrives at the sender, the window freezes until the timeout triggers.

The visual design reinforces the Go Back N rules. Packets move only left to right, ACKs move right to left, and the window box slides forward only when the correct cumulative ACK arrives. Every part of the protocol is represented by an animation or UI update, so you can see exactly what is happening step by step.