

Go-Back-N Protocol Implementation

Austin Berg

Department of Electrical Engineering and
Computer Science, University of Central
Florida, Orlando, Florida, 32816-2450

Abstract — This report presents a simulation framework for the Go-Back-N Automatic Repeat reQuest (ARQ) protocol, implemented using Python and a graphical user interface to visualize protocol behavior in real time. The system models key networking concepts such as windowed transmission, packet loss, acknowledgment loss, and retransmission timing, providing a clear and interactive representation of reliable data transfer mechanisms. By emulating sender, receiver, and network channel components, the simulator enables controlled experimentation with configurable parameters including window size, propagation delay, and loss conditions. The modular architecture supports educational use and protocol analysis while maintaining clarity, responsiveness, and adaptability in both headless and GUI-driven environments. The paper outlines the protocol logic, simulation structure, and user interface design, highlighting its effectiveness in demonstrating core transport-layer reliability principles..

Index Terms — Go-Back-N, ARQ Protocol, Networking, Data Transmission, Python Simulation, Packet Loss, Reliable Communication, Transport Layer.

I. INTRODUCTION

In the field of computer networking, ensuring the reliable transmission of data over potentially unreliable communication channels is a fundamental challenge. Protocols designed to guarantee that all data sent by a source is accurately received by a destination, even in the presence of packet loss or corruption, are essential for modern communications. One such protocol, Go-Back-N (GBN), is an integral part of the family of sliding window protocols used in reliable data transfer systems. The purpose of this project was to implement the Go-Back-N protocol from the ground up and simulate its behavior under different network conditions using a custom-built Python application supported by a graphical interface. This report outlines the design, implementation, testing, and outcomes of this simulation, which was constructed to help demonstrate the operational logic and performance characteristics of the Go-Back-N protocol.

The Go-Back-N protocol works by allowing the sender to transmit multiple packets before needing an acknowledgment for each one. However, the protocol is designed in such a way that if even a single packet is lost or arrives out of order, the sender must go back and retransmit that packet and all subsequent packets in the current window. This behavior allows for increased efficiency under ideal conditions but can introduce redundant transmissions and inefficiencies when loss is frequent. Our goal in this project was to simulate this behavior in an educational and controlled environment, allowing the user to manually introduce loss and delay conditions, observe the protocol's response, and gain an intuitive understanding of its advantages and drawbacks.

II. SYSTEM DESIGN

The simulation system was structured around three primary components: the sender, the receiver, and a simulated network channel that relayed packets and acknowledgments between them. Each of these components was implemented as an independent class in Python and operated concurrently using threading. The design was built to reflect the essential aspects of the Go-Back-N protocol while maintaining simplicity and clarity for educational purposes.

At the sender's side, we implemented a sliding window mechanism, governed by a configurable window size (N). This window allowed the sender to send up to N packets without waiting for individual acknowledgments for each. Each data packet was assigned a sequence number drawn from a fixed-size cyclic space, for example, values ranging from 0 to 7 in the case of an 8-number sequence space. When the sender transmitted the packets, it buffered those that were not yet acknowledged and started a single timer for the oldest outstanding packet. If an acknowledgment did not arrive within a set timeout period, the sender would retransmit all unacknowledged packets starting from the base of the window.

The receiver's design followed the strict in-order delivery model required by Go-Back-N. It accepted only the next expected sequence number and discarded any out-of-order packets. Upon successfully receiving a correct packet, the receiver delivered the data to the application and sent back a cumulative acknowledgment for the highest in-order packet it had received. This acknowledgment scheme allowed the sender to slide its window forward and send additional packets. Out-of-order packets, even if received correctly, were not buffered and were ignored, which led to


a clear illustration of the inefficiencies introduced by Go-Back-N under loss conditions.

The network channel component served as the intermediary between sender and receiver, emulating real-world behaviors such as packet loss, acknowledgment loss, and propagation delays. Users could configure which specific packets or acknowledgments should be dropped, allowing for repeatable and controlled test cases. The channel introduced delay using a threading mechanism and invoked packet drops by checking against the user-specified configuration. This mechanism enabled the simulation of both ideal and degraded network conditions.

III. IMPLEMENTATION

The Go-Back-N protocol simulation was implemented in Python using the threading module to simulate the concurrency inherent in network communication. The sender and receiver operated on separate threads, with additional threads created to handle transmission delays within the network channel. All threads operated concurrently, and synchronization was managed using locks to protect shared state such as the sender's sliding window buffer and timers.

The graphical user interface, as shown in the figure to the right and built using Tkinter, provided an accessible platform for configuring and running simulations. Users could specify the sender's window size, define the data string to transmit, identify which packets or acknowledgments to drop, and set the propagation delay. Once configured, the simulation could be started from the UI, with live logs displayed in a text window showing every action taken by the sender, receiver, and network channel. This log included messages for packet transmission, acknowledgment receipt, retransmissions due to timeouts, and final delivery confirmation.


Go-Back-N Protocol Simulator

Sender Window Size	4
Data to Send	ABCDEF
Packet Loss (seq #s comma-separated)	2
ACK Loss (ack #s comma-separated)	
Propagation Delay (seconds)	0.5

Run Simulation

```

[SENDER] Will attempt to send: ABCDEF
[Sender] Sent: DATA(seq=0, data=A)
[Sender] Sent: DATA(seq=1, data=B)
[Sender] Sent: DATA(seq=2, data=C)
[Sender] Sent: DATA(seq=3, data=D)
[Receiver] Received: DATA(seq=0, data=A)
[Receiver] Accepted: A
[Receiver] Received: DATA(seq=1, data=B)
[Receiver] Accepted: B
[Channel] Packet 2 dropped.
[Receiver] Received: DATA(seq=3, data=D)
[Receiver] Out-of-order. Expected 2, got 3.
[Sender] Received: ACK(seq=0, data=)
[Sender] Received: ACK(seq=1, data=)
[Sender] Sent: DATA(seq=4, data=E)
[Sender] Received: ACK(seq=1, data=)
[Sender] Sent: DATA(seq=5, data=F)
[Receiver] Received: DATA(seq=4, data=E)
[Receiver] Out-of-order. Expected 2, got 4.
[Receiver] Received: DATA(seq=5, data=F)
[Receiver] Out-of-order. Expected 2, got 5.
[Sender] Received: ACK(seq=1, data=)
[Sender] Received: ACK(seq=1, data=)
[Sender] Timeout. Resending from 2.
[Sender] Retransmitting: DATA(seq=2, data=C)
[Sender] Retransmitting: DATA(seq=3, data=D)
[Sender] Retransmitting: DATA(seq=4, data=E)
[Sender] Retransmitting: DATA(seq=5, data=F)
[Receiver] Received: DATA(seq=2, data=C)
[Receiver] Accepted: C
[Receiver] Received: DATA(seq=3, data=D)
[Receiver] Accepted: D
[Receiver] Received: DATA(seq=4, data=E)
[Receiver] Accepted: E
[Receiver] Received: DATA(seq=5, data=F)
[Receiver] Accepted: F
[Sender] Received: ACK(seq=2, data=)
[Sender] Received: ACK(seq=3, data=)
[Sender] Received: ACK(seq=4, data=)
[Sender] Received: ACK(seq=5, data=)
[Sender] All data acknowledged.

=== Simulation Complete ===
Data delivered: ABCDEF

```

IV. CONCLUSION

The Go-Back-N protocol simulation successfully demonstrated the core behaviors and characteristics of a fundamental reliable transport protocol. Through careful system design, modular implementation, and a flexible user interface, the project achieved its goal of offering an educational tool for understanding reliable data transfer in the face of packet loss, acknowledgment loss, and network delays.

Observations from various simulations under different configurations revealed clear patterns consistent with theoretical expectations. In scenarios with no packet or acknowledgment loss, the protocol operated with maximum efficiency, delivering all data with minimal delay. When specific packets were dropped, the protocol correctly retransmitted the affected and subsequent packets after the timeout period, and the receiver properly discarded out-of-order data until the missing packet was received. Even when acknowledgments were dropped, the sender's timeout and retransmission logic-maintained data integrity, albeit at the cost of some redundancy. Increasing the sender's window size allowed for higher throughput but also magnified the effects of retransmissions when losses occurred.

The simulation emphasized the trade-offs involved in Go-Back-N's design. While it is simpler than more advanced protocols like Selective Repeat, it incurs performance penalties under lossy conditions due to its lack of buffering at the receiver. Nevertheless, the protocol's deterministic behavior, reliance on cumulative acknowledgments, and single-timer design made it ideal for educational purposes. The UI-based simulation provided an interactive way to witness these dynamics and fostered a deeper understanding of reliable transport protocols in networking.

This project has laid the groundwork for further exploration into ARQ protocols. Future enhancements could include implementing Selective Repeat for comparison, adding checksum-based corruption detection, and generating automated performance metrics to quantify retransmission rates and throughput under various network conditions.