EECE 7374: Fundamentals of Computer Networks

Engineering Department
Northeastern University

Semester – Fall 2023
Instructor – Dr. Dimitrios Koutsonikolas

Report for
Programming Assignment 2:
Reliable Transport Protocols

By
Soniya Kadam

Maryam Aminu Mukhtar

December 7th, 2022

Academic integrity – We have read and understood the course academic integrity policy.

CONTENTS

Overview

Our objective of this project is to design, test, and implement three reliable data transport protocols: Alternating-Bit (ABT), Go-Back-N (GBN), and Selective-Repeat (SR). We need to implement the three reliable data transfer protocols within the given simulator by writing C code that compiles under the GCC environment, ensuring compatibility and correct operation on a designated host that we can ssh into. The tasks involve creating sending and receiving transport-layer code for these protocols that will execute in a simulated hardware/software environment. The programming interface provided to our routines is closely aligned with that in a UNIX environment; the code should call our entities from above and from below implementing the simulated timers to start and stop and timer interrupts will cause our timer handling routine to be activated. The goal is to develop a reliable data transfer mechanism, ensuring accurate and in-order data delivery under network conditions like packet loss and corruption, emulated in the simulated environment.

Overview of the Five-Layer Network Model

In this programming assignment we will be working with the five-layer network model. Each layer has a specific function essential for effective communication. Layer 1, the Physical Layer, handles the physical transmission of data, dealing with the hardware and transmission mediums. Layer 2, the Data Link Layer, ensures reliable node-to-node data transfer, managing error detection and frame synchronization. Layer 3, the Network Layer, oversees data routing across the network, using protocols like IP to determine data paths. Layer 4, the Transport Layer, is responsible for reliable data transmission, handling error correction and flow control through protocols like TCP and UDP. Finally, Layer 5, the Application Layer, interfaces directly with end-users, facilitating application services and data encoding, and includes protocols such as HTTP and FTP. Each layer interacts systematically with the layers above and below, creating a structured approach to network communication.

Design Requirements

In this programming assignment, we will develop a series of routines critical for the functioning of the ABT, GBN, and SR protocols. These routines include sender and receiver functions for each protocol, tailored to handle data packet transmission, acknowledgment receipt, and error detection. We will also implement timer management functions to handle timeouts effectively, a crucial aspect of ensuring reliable data transmission. Additionally, the routines encompass mechanisms for packet creation, processing of received packets, and handling of corrupted or lost packets. The implementation will adhere to the constraints of the simulated network environment, including managing buffer space and responding to network events. Our focus will be on robust error handling and efficient data flow, ensuring data integrity and order are maintained under varying network conditions. The routines we write will collectively ensure the reliable transfer of data, adhering to the specifications of each protocol. We will be working with two primary data structures: msg and pkt. The msg structure, used for communication between the upper layers and our protocols, consists of a 20-byte data array. This structure allows us to handle data in 20-byte chunks, aligning with the requirements of Layer 5 communication. On the receiving end, our task is to ensure that these 20-byte data chunks are correctly received and delivered to Layer 5. The pkt structure is used for communication between our routines and the network layer, includes a sequence number (seqnum), an acknowledgment number (acknum), a checksum, and a 20-byte payload. Our routines will populate the payload from the message data received from Layer 5, using the other fields to ensure reliable delivery as per the protocol requirements. The Simulated

network environment uses a call to procedure 'tolayer3()' to send packets into the medium (i.e., into the network layer). While our procedures A_input() and B_input() is called when a packet is to be delivered from the medium to your transport protocol layer. The network medium in this simulation can corrupt and lose packets, although it does not reorder them. When we compile and run the program with the provided procedures, we will need to input specific values related to the simulated network environment as command-line arguments. For more detailed information about these parameters and their effects on the simulation, please refer to the project description document.

Routines To Design: Our program requirements and specifications for this project requires us to write 6 routines:

- **A_output(message):** This function is called when the upper layer at the sending side (A) has a message to send to the receiving side (B). It's responsible for ensuring that the data in the message is delivered correctly to the receiving side's upper layer.
- **A_input(packet):** This function is triggered when a packet from the B-side arrives at the A-side. It handles the received (and corrupted) packet.
- **A_timerinterrupt():** This is called when A's timer expires, signaling a timer interrupt. It's typically used for controlling the retransmission of packets.
- **A_init():** This initialization routine is called before any other A-side routines. It sets up any necessary preconditions for the protocol's operation.
- **B_input(packet):** Similar to A_input, this function is called when a packet from the A-side arrives at the B-side, handling the received packet.
- **B_init():** This is the initialization routine for the B-side, called before any other B-side routines to set up required conditions.

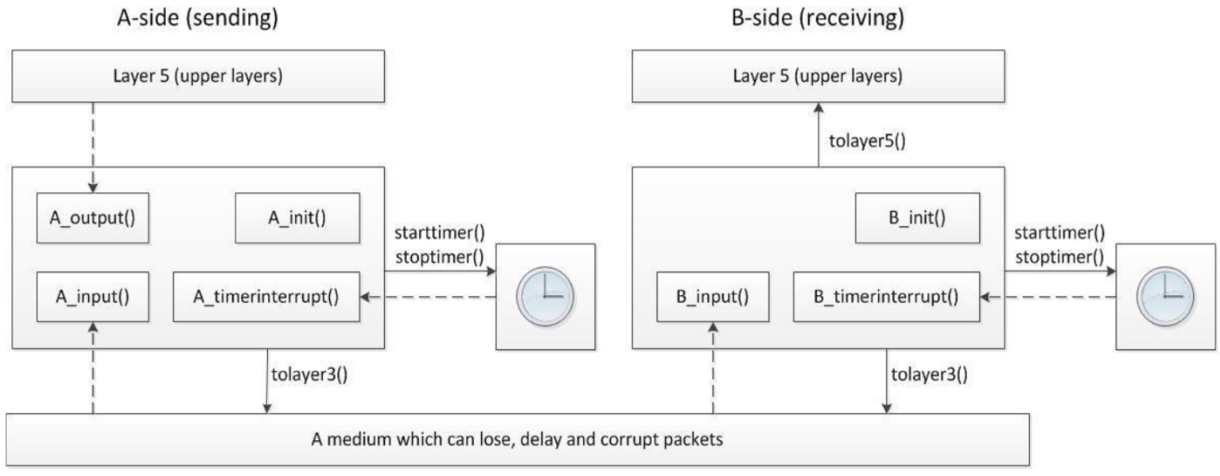


Figure 1. Structure of the environment to simulate the network environment

Provided Routines: Our program specifications for this project provided several key routines that support the functionality of our data transport protocols. These were pre-written and available for integration with the routines we will develop. These interfaces form a foundational backbone for the implementation of our transport protocols, offering essential functionalities like timing control, packet transmission, data delivery, and configuration retrieval. The accurate integration of these

interfaces is crucial for the successful implementation of our Alternating-Bit, Go-Back-N, and Selective-Repeat protocols.

- **starttimer (calling_entity, increment):** This function initiates a timer for either the A-side (calling_entity = 0) or the B-side (calling_entity = 1). The 'increment' parameter is a float that specifies the duration before a timer interrupt occurs. It's important to use this function exclusively with the corresponding entity's routines (A-side or B-side).
- **stoptimer (calling_entity):** This function is complementary to the starttimer, it stops the timer for either the A-side (calling_entity = 0) or the B-side (calling_entity = 1), ensuring precise control over timing mechanisms.
- **tolayer3 (calling_entity, packet):** This routine handles the sending of packets. 'calling_entity' distinguishes whether the packet originates from the A-side (0) or the B-side (1), and 'packet' is a structure of type 'pkt'. This function is crucial for injecting packets into the network towards the respective destination.
- **tolayer5 (calling_entity, data):** This function is used for delivering data to layer 5 from either the A-side (calling_entity = 0) or the B-side (calling_entity = 1). The 'data' parameter is a character array with a fixed size, typically used for unidirectional data transfer to the B-side in this context.
- **getwinsize():** It returns the window size, which is an essential parameter in controlling the flow of data packets, particularly in protocols like Go-Back-N and Selective Repeat.
- **get_sim_time():** This function provides the current simulation time, an essential aspect for managing and debugging time-dependent behaviors in your protocols.

PROTOCOLS

This section of the report provides an overview of three distinct data transport protocols: Alternating-Bit (ABT), Go-Back-N (GBN), and Selective-Repeat (SR). These protocols are pivotal in ensuring reliable data transmission over a network, each with its unique approach to handling packet loss, corruption, and ordering. The project requirements for each protocol are detailed in their respective subsections. For more detailed information about the protocol requirements and their effects on the simulation, please refer to the project description document.

Alternating-Bit Protocol (ABT)

The Alternating-Bit Protocol is a simple form of error control for data transmission. It uses a single bit to keep track of packet sequences, alternating between 0 and 1. Each packet sent must be acknowledged before the next packet is sent. If an acknowledgment is not received within a specified time frame, the packet should be retransmitted.

Project Requirements: The ABT protocol should only use ACK messages. This implementation should handle timeouts and retransmissions effectively. It should perform a check at the sender to make sure that when A_output() is called, there is no message currently in transit. If there is, the data being passed to the A_output() routine should be buffered. It should also manage packet corruption and loss as outlined in the simulation environment.

Go-Back-N (GBN)

Go-Back-N is a more sophisticated protocol than ABT, it allows for the sending of multiple packets before requiring an acknowledgment. It uses a sliding window mechanism to control the flow of packets. If a packet is lost or corrupted, all subsequent packets in the window are retransmitted.
Project Requirements: The GBN protocol can use a finite number of buffers at the sender, with a

recommended size of 1000 messages. If all buffers are occupied, the sender should abort operations.

Selective-Repeat (SR)

Selective-Repeat improves upon Go-Back-N by having the option of selectively retransmitting only the specific packets that were acknowledged as lost or corrupted, rather than the entire window. This protocol also employs a sliding window mechanism but requires more complex tracking of individual packet states.

Project Requirements: The GBN protocol receiver should reply with ACKs to all packets falling inside the receiving window. A key requirement is the effective use of a single timer to simulate multiple logical timers. This is particularly important for managing the scenario where multiple packets sent by the A-side are outstanding and unacknowledged in the network.

PROTOCOL IMPLEMENTATION

In this section, we delve into the detailed implementation of each protocol - ABT, GBN, and SR. Our focus is to provide a comprehensive understanding of how these protocols were implemented by discussing the structure of the code, and how various challenges and issues were addressed during the development process. For each protocol, we'll explore key aspects such as the handling of packet transmission and acknowledgment, the management of timers and buffers, and the strategies used to deal with packet loss, corruption, and other network anomalies.

Alternating-Bit Protocol (ABT)

The Alternating-Bit Protocol is designed to simulate unidirectional data transfer from sender A to sender B, incorporating various network properties such as delay, packet corruption, and loss. We will go through the code and how it works all together when there is a message to send to B.

The program initiates by initializing the necessary headers and defining global variables for entities A and B. These variables are crucial for keeping track of sequence numbers, acknowledgments, and buffer statuses. The program also introduces two key structures: 'pkt' for packets and 'msg' for messages. A significant feature for entity A is the implementation of a message queue, which is used to manage messages that are queued up for transmission. We also define a MAX_BUFFER_SIZE and timeout value to represent the estimated maximum size of the buffer used in the program where data is held temporarily during the transfer process between entities in the network and the timeout value for the network communication protocol.

The functions ‘enqueue’ and ‘dequeue’ in the program are essential for managing the flow of messages within a queue for the buffer storage. The ‘enqueue’ function is designed to insert messages into the queue. It first checks if the current size of the queue is less than the maximum buffer size (`MAX_BUFFER_SIZE`). If the queue is not full, it adds the message to the end of the queue and increments the queue's size. This function ensures that messages are stored and ready to be processed in an orderly manner, respecting the queue's capacity limits. On the other hand, the ‘dequeue’ function is responsible for removing messages from the queue. It retrieves the message at the front of the queue, adjusts the front pointer to the next message, and decreases the queue's size. This function is crucial for processing and sending messages in the order they were received; not mixing up the sequence numbers thereby maintaining the integrity of the communication process. Together, these functions effectively manage the queuing system, ensuring that messages are handled efficiently in the network simulation.

The functions '**compute_checksum**' and '**is_corrupted**' are integral to ensuring data integrity in the network communication simulation. The '**compute_checksum**' function calculates the checksum of a packet to allow for error-detection in the packet's data. We find the checksum value by summing the sequence number and acknowledgment number of the packet. Then, it iteratively adds the values of each byte in the packet's payload (the 20 bytes from msg) to this checksum. This computed checksum is a simple yet effective way to verify the integrity of the packet's data during transmission. On the other hand, the '**is_corrupted**' function utilizes '**compute_checksum**' to determine if a packet is corrupted. It compares the newly computed checksum of the packet (at the receiver) with the packet's original checksum value. If these values do not match, that indicates the packet has been corrupted during transmission. The function then returns a boolean result indicating the corruption status of the packet. This mechanism is crucial for the network protocol to identify and handle corrupted packets, ensuring reliable data communication.

To keep the program organized we created the functions '**create_packet**', and '**create_ack_packet**' for packet management in the network communication simulation. The '**create_packet**' function is responsible for constructing a new packet. It initializes a packet structure with the sequence number and sets the acknowledgment number to zero as we don't need it. The function then copies a fixed-size message (the 20 bytes from msg) into the packet's payload, ensuring null-termination at the end. After filling the packet, it calculates the packet's checksum using '**compute_checksum**'. This checksum is essential for later verifying the integrity of the packet during transmission to B.
The '**create_ack_packet**' function, on the other hand, is specifically designed to create ACK packets. It initializes an ACK packet with the provided acknowledgment number and sets the sequence number to zero, as it is not needed in the ACK packets. The payload of the ACK packet is cleared to ensure it contains no data, and then the function calculates the checksum for the ACK packet as even ACK packets can get corrupted during transmission.

Next, we have the first function for the sender (A) '**A_output**' it is the interface between the application layer (layer 5) and the network layer (layer 3). It is responsible for handling outgoing messages from the application layer and organizing them for transmission over the network. When this function is called, it first prints the message that needs to be sent for debugging and then enqueues it, ensuring that all outgoing messages are first stored in the queue before being sent. This queuing mechanism is crucial for managing the flow of data, especially when acknowledgments for previous messages are pending. The function then checks if the sender (A) is currently waiting for an acknowledgment for a previously sent packet. If it is not waiting for an acknowledgment and there are messages in the queue, it proceeds to dequeue the next message. The dequeued message is then used to create a new packet using the '**create_packet**' function with the current sequence number ("A_seqnum"). The packet's checksum is calculated within '**create_packet**'. Once the packet is created, it is sent to the network layer using tolayer3 for transmission. Immediately the packet is sent the function starts a timer ('**starttimer**') with the predefined timeout period (TIMEOUT = 20). This timer is used to detect lost packets and trigger retransmissions if necessary. Finally, the function sets an ACK flag ("A_waiting_for_ack") to indicate that it is now waiting for an acknowledgment for the sent packet, and it also stores the packet as the last packet sent ("A_last_packet"). This is so it can be used for potential retransmission in case the acknowledgment is not received within the timeout period or there is packet corruption somewhere.

The function '**A_input**' function is called from layer 3 to handle incoming packets at A, in this project an acknowledgment, after they arrive for processing at the transport layer (layer 4). Upon being called, the function first logs the received and expected ACK numbers for debugging purposes. It then proceeds to verify two key conditions: whether the received packet is not corrupted and whether the acknowledgment number in the packet matches the expected acknowledgment number at A. If both conditions are met, indicating that the packet is valid and the acknowledgment is correct, the function stops the timer that was started when the packet was initially sent. This is to prevents unnecessary retransmission of the packet as it has been successfully acknowledged. The function then toggles the sequence number ("A_seqnum") and the expected acknowledgment number ("A_expected_ack") to the next packet, preparing for the transmission of the next message. We also reset the flag indicating that A is waiting for an acknowledgment here to allow new packets to be sent. Next, the function checks if there are additional messages queued for transmission, by checking the size of '**A_message_queue**' if it is greater than zero. If there is buffered messages the function dequeues the next message, creates a new packet with the updated sequence number, and sends this packet to the network layer. It then restarts the timer and updates the ACK flag to indicate it is waiting for an acknowledgment for this new packet. However, if the received packet is either corrupted or the acknowledgment number does not match the expected value, the function logs an error message indicating an invalid ACK or a corrupted packet.

Then, we have the '**A_timerinterrupt**' function to handle situations when A's timer goes off. It could be because an acknowledgment for a packet is not received within a certain timeframe (indicated by the "TIMEOUT" variable). This function could also be triggered when a timeout occurs due to potential packet loss or corruption. It resends the last packet sent by A ("A_last_packet") to ensure it is the exact data that was affected that reaches B. The last function for A is '**A_init**' it is called at the start to initialize all the variables for entity A, setting up the initial sequence number, ack flag and the message queue.

Afterwards, we have the '**B_input**' function for the receiver (B), it is called from layer 3 when a packet arrives to layer 4 at B. The function starts by logging the sequence number of the received packet and the expected sequence number at B, providing a clear indication of the packet's arrival and its sequence status. It then proceeds to verify two key conditions: whether the packet is not corrupted and whether its sequence number matches the expected sequence number at B. If both conditions are met the function confirms the packet's validity, then proceeds for debugging to copy the first 20 characters of the packet's payload into a temporary buffer, ensuring the string is null terminated for proper processing and logged. This payload is then sent to the application layer (layer 5), signifying successful receipt and processing of the packet's data. Subsequently, the function creates an ACK packet using the '**create_ack_packet**' function, with the sequence number of the received packet ("B_expected_seqnum"). As in ABT the acknowledgment number sent back by the receiver typically matches the sequence number of the packet that was correctly received. This ACK packet is then sent back to layer 3, signaling to entity A that the packet has been received correctly. The function logs the details of the ACK packet, including its sequence number, acknowledgment number, and checksum, for debugging purposes. After sending the ACK, the function updates "B_expected_seqnum" by toggling it, preparing for the next expected packet. This update is necessary to maintain the correct order of packet processing. In cases where the packet is either corrupted or does not have the expected sequence number, the function logs an

error message, indicating an issue with the received packet. This will trigger a packet retransmission when A's timer goes off after not receiving an ack for the packet. The last function for B is '**B_init**' it is called at the start to initialize all the variables for entity B, setting up the initial expected sequence number.

We noticed when B updates its expected sequence number after sending an ACK, but that ACK gets corrupted enroute to A, we have a situation where B is expecting the next packet (with the updated sequence number), but A, not receiving the ACK, retransmits the previous packet. This leads to a mismatch in expectations between A and B as the receiver's state has moved forward, but the sender's state is still on the previous packet. So, we updated the function to handle scenarios where the acknowledgment might have been lost or corrupted on the way to the sender, prompting a packet retransmission. We added an If statement to check if the received packet's sequence number matches the sequence number of the last acknowledged packet ("B_last_acknowledged_seqnum"), the function recognizes this as a retransmission and resends the last acknowledgment packet ("B_last_ackpacket") to A.

Another issue we had was initially, in our protocol, we treated the messages as strings, utilizing standard string functions like strcpy and strlen. This approach seemed intuitive, given the common use of strings for text data. However, we encountered issues with extra characters appearing in our print statements, which led to confusion and inefficiency in our message handling. Upon consulting with our professor, we realized the root of the problem: our messages were not standard null-terminated strings but rather a sequence of bytes and characters. We then switch our approach. and changed the string-specific functions to treat the messages as raw bytes and characters. This change significantly improved the accuracy of our message processing, as it aligned with the actual nature of the data we were handling, thereby eliminating the anomalies we previously encountered. We found when we print the payload for debugging there are still extra characters and after researching about it we realized its due to how in C, when you use functions designed for printing strings (like printf with %s format specifier), they expect a null-terminated string and as we are using it to print a sequence of bytes that may not always be null-terminated strings it can lead to incorrect printing. We conformed this with the fact the checksum for the messages were the same at both the sender and receiver verifying the message was never affected even when the print statement said otherwise.

Go-Back-N (GBN)

The Go-Back-N Protocol is designed to simulate unidirectional data transfer from sender A to sender B, incorporating various network properties such as delay, packet corruption, and loss. We will go through the code and how it works all together when there is a message to send to B. We started by first implementing the easier ABT protocol and then extended the code to implement the more difficult GBN protocol. We start the program again by initializing the necessary headers and defining global variables for entities A and B. We added a window size variable and updated the buffer to function based on the window size. The program also introduces the same two key structures: 'pkt' for packets and 'msg' for messages. A significant new feature for entity A is the implementation of window size. In ABT, only one packet is sent at a time and must be acknowledged before the next is sent. However, GBN allows for multiple packets to be sent within a window before requiring an acknowledgment. This window size dictates the number of packets that can be outstanding (sent but not yet acknowledged) at any given time. To accommodate this, we updated A's logic to handle a window of packets, tracking which have been sent and

acknowledged. This involves maintaining a buffer of packets to be sent and implementing a sliding window mechanism. Additionally, the acknowledgment processing in A needs to be modified to handle cumulative acknowledgments, where a single acknowledgment can confirm the receipt of multiple packets. The B receiver will also require updates to acknowledge multiple packets and to handle out-of-order packets according to the GBN protocol rules. We expect these changes to significantly increase the throughput of our protocol, as multiple packets can be in transit simultaneously, making it a more efficient use of the network capacity.

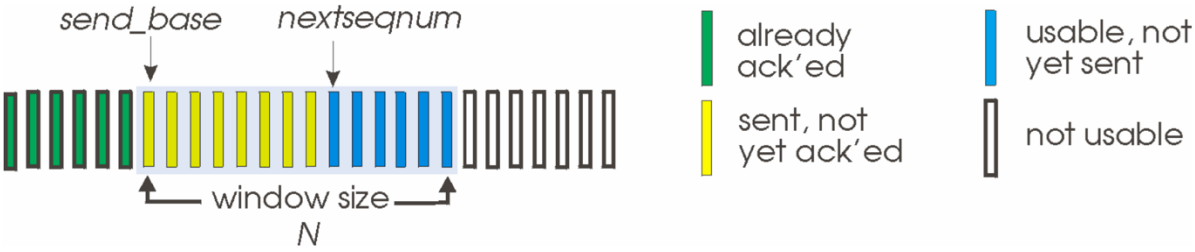


Figure 2. Go-Back-N sender structure

The functions '**enqueue'** and '**dequeue'** in the program essential for managing the flow of messages within a queue for the buffer storage don't change from ABT but we created simple '**is_full**' and '**is_empty**' function to handle the check on the queue's capacity. The integral functions '**compute_checksum**' and '**is_corrupted**' that we use to ensure data integrity during simulation transmissions from ABT also remain the same.

We created a function '**send_window_packets**' to be responsible for sending all packets within the sender's window that have not yet been sent. The function operates in a loop, continuing as long as the next sequence number ("A_nextseqnum") falls within the current window which is defined by "A_send_base + WINDOW_SIZE" and if the message queue is not empty. Within this loop, the function dequeues a message from the queue, creates a new packet using '**create_packet**' with the dequeued message and the current sequence number, it then stores this packet in "A_buffer" in case of later retransmission. After storing, the function sends the packet to layer 3 so it can get sent to B and increments "A_nextseqnum" to point to the next sequence number. After exiting the loop the function checks if there are any packets that have been sent but not yet acknowledged ("A_send_base < A_nextseqnum"), the function then starts a timer ('**starttimer**') to handle potential timeouts The timer let us detect if packets are lost and need retransmission. The variable "is_timer_running" is set to 1, which we use to keep track of if the timer is active.

The core functionality of '**A_output**' is updated to revolve around the sliding window, as defined by "WINDOW_SIZE". The function first checks if the window is not full; if the next sequence number is less than the sum of the base of the window and the window size. If there is room in the window, the function proceeds to create a packet from the message, with the current sequence number. The packet is then stored in A_buffer for potential retransmission and the packet is sent to the network layer using 'tolayer3'. To set the timer to trigger retransmissions and detect packet loss, we had the function start a timer with the predefined timeout if the packet being sent is the first in the window ("A_send_base" equals "A_nextseqnum"). After sending the packet, the function increments the sequence number ("A_nextseqnum") to prepare for the next packet. If the window is full, the function enqueues the message for later transmission and logs the size of the

message queue. This queuing mechanism ensures that messages are not lost when the window is saturated and will be sent as soon as space becomes available. We use "A_buffer" to store the packets for retransmissions to manage packets that have been sent but not acknowledged while '**A_message_queue**' will be used to buffer the messages when the window is full to manage messages that are waiting to be sent.

The function '**A_input**' is designed for handling acknowledgments received from the receiver (B). This function updated so it can handle cumulative ACK's. The function starts by logging the acknowledgment number of the received packet for debugging. It then checks two key conditions: whether the packet is not corrupted and whether the acknowledgment number is within the current sending window, defined by "A_send_base" and "A_nextseqnum". The acknowledgment number must be greater than or equal to "A_send_base" and less than "A_nextseqnum" to be considered valid as that means it is with the window of the packets sent. If the ACK is valid, the function acknowledges this by sliding the sending window forward. This sliding is achieved by updating "A_send_base" to one more than the received acknowledgment number, effectively moving the window forward and acknowledging all packets up to that number. The function also updates "A_last_acked_seqnum" to keep track of the last acknowledged sequence number for debugging. For the timer, the function checks if all the outstanding packets (those that have been sent but not yet acknowledged) are acknowledged using "A_send_base" equals "A_nextseqnum". If there are no more packets awaiting acknowledgment the function stops the timer. If there are still outstanding packets, the timer is restarted to ensure that the packets do not exceed their allowed time in transit without acknowledgment. Finally, the function calls '**send_window_packets**' to check if there are any buffered messages in '**A_message_queue**' that can now be sent due to the window sliding forward. This ensures that the sender continuously sends packets as long as the window allows and there are messages to be sent.

Then, we have the '**A_timerinterrupt**' function to handle situations when A's timer goes off. It could be because an acknowledgment for one or more sent packets has not been received within the expected timeframe ("TIMEOUT" variable). This function could also be triggered when a timeout occurs due to potential packet loss or corruption. The function begins by logging that the timer interrupt has occurred and immediately restarts the timer with the same timeout period to ensure that the timer is handling the latest transmission attempt for the multiple packets. It then enters a loop, retransmitting all packets from the "A_send_base" (start of the window) up to "A_nextseqnum" (sequence number of the next packet to be sent). This retransmission is necessary because, in GBN, the loss of an ACK or a packet requires the sender to resend all packets in the window that have not been acknowledged. Each packet is resent by calling 'tolayer3', ensuring that they are retransmitted to B.

The last function for A is '**A_init**' it is called at the start to initialize all the variables for entity A. It sets the window size based on a system or user-defined parameter -w by calling '**getwinsize()**'. The function initializes "A_send_base" and "A_nextseqnum" to zero, setting it up for the first packet transmission. It also initializes the message queue ('**A_message_queue**'), which will store messages waiting to be sent. It also dynamically allocates memory for "A_buffer", which is an array of packets used to store the copies of the packets sent (but not yet acknowledged) within the window. This allocation is based on the window size, ensuring that there is enough space to store all packets in the transmission window. If memory allocation fails, the function handles this error by logging it and exiting the program.

Next, we have the '**B_input**' function for the receiver (B), this function is called from layer 3 at B when a packet arrives from layer 4. It first checks if the packet is not corrupted and its sequence number matches the expected sequence number ("B_expected_seqnum"), the function then sends the packet to the application layer. It then creates and sends an ACK packet to layer 3 to send to A using the current expected sequence number. This ACK is cumulative as it is acknowledging all packets up to that sequence number. The ACK packet is stored as "B_last_ackpacket" for possible retransmission. The function also increments the expected sequence number to the next expected packet. The function checks for any other type of packet, whether the received packet is out of order, a duplicate, or a retransmission, the function resends the last ACK packet ("B_last_ackpacket"). This approach handles any packets that do not match the current expected sequence number, ensuring that the sender knows it was the last correctly received packet. In cases where the packet is corrupted, it also resends the last ACK packet. This ensures that only packets in the correct sequence are processed and appropriately acknowledges both new and duplicate packets. The last function for B is '**B_init**' it is called at the start to initialize the variable for entity B, setting up the initial expected sequence number and '**A_cleanup**' to free up the dynamically allocated buffer at A and prevent memory leaks.

Selective-Repeat (SR)

The Selective-Repeat Protocol is designed to simulate unidirectional data transfer from sender A to sender B, incorporating various network properties such as delay, packet corruption, and loss. We will go through the code and how it works all together when there is a message to send to B. We started by first implementing the GBN protocol and then extended the code to implement the more difficult SR protocol. We updated both the sender (A) and the receiver (B). For A, the key update is the implementation of a single timer that effectively manages multiple logical timers for each packet within the window. This because in SR each packet is treated independently has its own acknowledgment and timeout. A must also be capable of handling individual acknowledgments and retransmitting only the packets that have not been acknowledged within the timeout period, rather than resending the entire window of packets as in GBN. For the receiver B, a significant new feature is the ability to buffer multiple out-of-order packets. Unlike in GBN where out-of-order packets are discarded, SR allows the receiver to accept and buffer these packets. This change requires maintaining a receiving window and acknowledging each packet that falls within this window, even if they arrive out of order. The receiver must keep track of which packets have been received and send ACKs for each of them, ensuring that the sender knows which packets arrived at B successfully in the window.

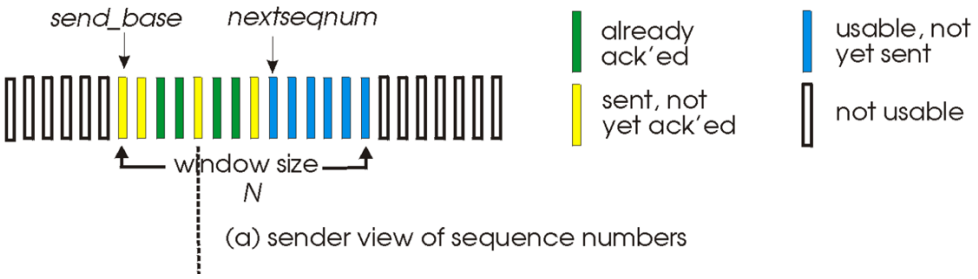


Figure 3. Selective-Repeat sender structure

We start the program again by initializing the necessary headers and defining global variables for entities A and B. We added a "MAX_SEQ_NUM" variable to manage the sequence numbers of

the packets sent and received to ensure that the sequence numbers within the window are unique and distinguishable from those in the next window. We went with a value of 256 to ensure we don't have a sequence number wrap around issue as the highest window size in the tests was 50 and a window size needs be less than or equal to half of the maximum sequence number. We added an "A_acknowledged" array to keep track of which packets have been acknowledged by the receiver in case of a timeout or error. At B, we created "B_buffer" to store out of order packets at the receiver, "B_received" array to keep track of the received packets at B and "B_send_base" to track the start of a receiving window.

The helper functions '**enqueue**', '**dequeue**', '**compute_checksum**' '**is_corrupted**', '**create_packet**', and '**create_ack_packet**' in the program don't change from GBN. We added two new helper functions; '**process_buffered_packets**' is designed to handle the processing of packets that have been buffered at the receiver's end. It operates by looping through the received packets (stored in "B_buffer") as long as there are consecutive packets starting from the expected sequence number. For each packet it finds that is next in order, the function delivers its payload to the application layer using 'tolayer5', and then clears the flag in the "B_received" array, to show that the packet has been processed. The expected sequence number is then incremented, wrapping around if its higher than the max sequence number. This will ensure that out of order packets are processed in sequence and delivered to the application layer correctly.

The '**A_output**' function after being called, first logs the message it has received for transmission. It then checks if the sender's window is not full, by comparing the next sequence number ("A_nextseqnum") with the sum of "A_send_base" and the window size. If the window is not full the function creates a packet from the message with the current sequence number. This packet is stored in "A_buffer" for potentially retransmission later. The packet is then sent to layer 3 to get sent to B. Next, we set up the timer, so if the packet being sent is the first in the window; checked using "is_timer_running" the function starts a timer with the predefined timeout period to keep track of lost packets and trigger retransmissions if necessary and we set is_timer_running to 1 to signify its active and the first packet in the window has been sent. The sequence number ("A_nextseqnum") is then incremented to prepare for the next packet ensuring it wraps around the maximum sequence number limit to keep the sequence number within the valid range. If the window is full, the function enqueues the message for later transmission. This will make sure messages are not lost when the window is saturated and will be sent as soon as space becomes available.

The '**A_input**' function is called from layer 3 to handle ACK packets when they arrive for layer 4 at the sender. The function first checks if the received packet is not corrupted and if its acknowledgment number falls within the current sending window; "A_send_base" tells us the start of the window and "A_nextseqnum" tells us the cutoff for the window. If these conditions are met, the function marks the packet as acknowledged in the "A_acknowledged" array that tracks the acknowledgment status of each packet in the window. Next, we check if the received ACK number matches "A_send_base" (start of the sending window), if it does the function enters a loop to slide the window forward by moving only when the lowest unack'ed packet receives an ACK. It then clears the acknowledgment status for the base packet and increments "A_send_base". The sliding will continue as long as the next packets have also been acknowledged (tracked by "A_acknowledged"). The timer management in the function checks if "A_send_base" is equal to "A_nextseqnum", meaning all sent packets have been acknowledged the function stops the timer

as there are no more packets left in the window. If there are still unacknowledged packets and the next packet in the window is not acknowledged, the function restarts the timer to give the packets more time. If the packet is corrupted or the ACK number is outside the current window, the function logs that an invalid ACK has been received and waits for a timeout or a retransmission from B.

If the packet is valid, the function calculates the upper bound of the sender's window based on "A_send_base" and "WINDOW_SIZE", wrapping around using "MAX_SEQ_NUM" when necessary. The function then checks if the acknowledgment number in the packet falls within the current sender's window by comparing "acknum" with "A_send_base" and the windows upper bound. If the acknowledgment is valid and within the window, the function marks the corresponding packet as acknowledged in the "A_acknowledged" array. Next, the function slides the sender's window forward as long as consecutive packets at the base of the window have been acknowledged. For each acknowledged packet, the acknowledgment status is reset and the "A_send_base" is incremented to move the window forward. After sliding the window, the function checks if there are any buffered messages in "**A_message_queue**", it then dequeues the message and calls '**A_output**' to send it. This will ensure that buffered messages are sent as soon as space becomes available in the sender's window. The timer is then started based on the state of the window. If "A_send_base" equals "A_nextseqnum", indicating all sent packets have been acknowledged, the timer is stopped. Otherwise, the timer is restarted to wait for the remaining unacknowledged packets ACKs.

Then, we have the '**A_timerinterrupt**' function to handle situations when A's timer goes off it signifies that the oldest unacknowledged packet has not been received within the expected timeframe. The function starts by logging the timeout event. It then checks if the oldest packet "A_send_base % WINDOW_SIZE" in the buffer array A_acknowledged has not been acknowledged yet. If the packet is unacknowledged, the function retransmits the packet sending it to layer 3. After retransmitting the oldest packet, the function restarts the timer using '**starttimer**' with the "TIMEOUT" value and set "is_timer_running" to 1, indicating that the timer is active again. This will ensure that all lost or delayed packets are retransmitted. The last function for A is '**A_init**' it is called at the start to initialize all the variables for entity A. It sets the window size by calling '**getwinsize()**' like in GBN. The function initializes "A_send_base" and "A_nextseqnum" to zero, setting it up for the first packet transmission. It also initializes the message queue ('**A_message_queue**'), which will store messages waiting to be sent. It also dynamically allocates memory for "A_buffer" and "A_acknowledged". The memory allocation is checked for success, and the program exits if allocation fails. The "A_acknowledged" array is then initialized to zero, indicating that initially, no packets have been acknowledged.

Next, we have the '**B_input**' function for the receiver (B), this function is called from layer 3 at B when a packet arrives from layer 4. The function first logs the receipt of the packet, including its sequence number and the expected sequence number at B. It then checks if the packet is not corrupted using '**is_corrupted**'. If the packet is valid, the function finds the lower and upper bounds of the receiver's window. It then checks if the received packet's sequence number falls within the expected window. If the packet is within the window, the function sends an ACK for packet using 'tolayer3' and '**create_ack_packet**. If the packet's sequence number equals "B_expected_seqnum", specifying it is the next expected in-order packet, the function delivers the packet's payload to the application layer immediately and increments "B_expected_seqnum". It

then calls '**process_buffered_packets**' to process any buffered packets that have now become in-order due to the increment. For packets that are out of order but within the window, the function buffers them and marks them as received in "B_received" to be later processed with '**process_buffered_packets**'. This buffering and processing will ensure that packets are delivered to the application layer in the correct order, despite potential out-of-order arrivals or delays in the network. The last function for B, '**B_init**' is called once before any other routines in entity B to initialize. It starts by setting "WINDOW_SIZE" based on the command line value. It initializes "B_send_base" and "B_expected_seqnum" to 0. Memory allocation is performed for `B_buffer`, which holds the packets in the window, and "B_received", the array used to track whether packets in the window have been received.

We ran an issue with the sequence numbers during testing with different window sizes. We started seeing errors related to the sequence number wraparound. The issue was worse as the window size increased, leading our tests to fail due to the reuse of sequence numbers for new packets while older packets with the same numbers were still in transit or awaiting acknowledgment. To resolve this, we slowly increased the maximum sequence number, first to 400 and then to 600 which we found to the best for the tests we were running. This increase effectively expanded the sequence number space, reducing the likelihood of wraparound within the window. A larger sequence number space is always better as it allows for uniquely identifying each packet, reduces the risk of errors due to number reuse, and supports larger window sizes, which can enhance throughput in network. After choosing the higher maximum sequence number, we effectively mitigated the wraparound problem, ensuring more reliable and efficient transmissions and also a higher throughput when using larger window sizes.

IMPLEMENTATION TESTING

For the programming assignment, the implementation of the three protocols were tested with different settings/parameters described in the project description document. I started with the most basic tests and then moved on to more involved tests like sanity and advanced tests to check for duplicate and/or out-of-order packets at B and to confirm that the behavior of the protocols is the expected one under some very simple tests (e.g. no packets are delivered under 100% loss).

ANALYSIS AND REPORT

We were provided with a set of experiments (see section 6.1 in the project description document) to compare the implementation of the three different protocols' performance, consisting of various loss probabilities, corruption probabilities, and window sizes.

Timeout Scheme

In our protocol implementation, we first started with a timeout of 20 as we know it takes a message an average of 5-time units to arrive at the other side, so we tripled it to be on the safe side due to possible delays and buffering and incase our program isn't the most optimal the message will still have enough time. We then experimented with different timeout values (high and low) to determine the best values for each protocol by observing their impact on the overall performance. This allowed us to identify a timeout threshold that had the best performance in terms of throughput and the different loss probabilities, corruption probabilities, and window sizes for the different protocols. The method we ended up choosing was based on these tests shown below.

A. Alternating-Bit Protocol (ABT)

ABT	
Average at Timeout = 10	
Loss	Throughput
0.1	0.019898
0.2	0.0199881
0.4	0.0200749
0.6	0.0101858
0.8	0.0025965

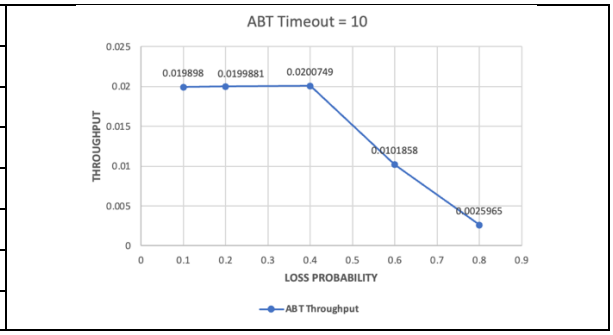


Figure 4. ABT average throughput at different losses with a Timeout of 10

At timeout 10, the throughput is relatively stable across different loss probabilities, with the best performance seen at a loss probability of 0.1 and the throughput decreasing as the loss probability increases, which we expect it to do. This tells that timeout of 10 is efficient for retransmissions without waiting too long or timing out prematurely.

ABT	
Average at Timeout = 20	
Loss	Throughput
0.1	0.0199543
0.2	0.0200457
0.4	0.0128183
0.6	0.005439
0.8	0.0013576

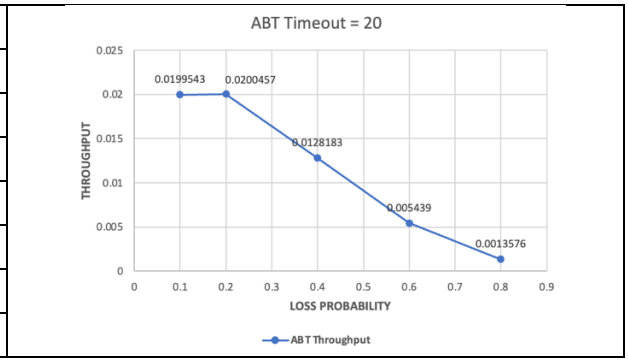


Figure 5. ABT average throughput at different losses with a Timeout of 20

At timeout 20, we have mixed results. For loss probabilities of 0.1 and 0.2, the throughput is similar to that at a timeout of 10. However, at higher loss probabilities, the throughput drops significantly, with a huge decrease at a loss of 0.6. This suggests that this longer timeout can negatively impact the performance, particularly at higher loss rates, because the protocol will end up waiting too long to retransmit lost packets.

ABT	
Average at Timeout = 50	
Loss	Throughput
0.1	0.0174756
0.2	0.0122096
0.4	0.0057787
0.6	0.0022411
0.8	0.0005433

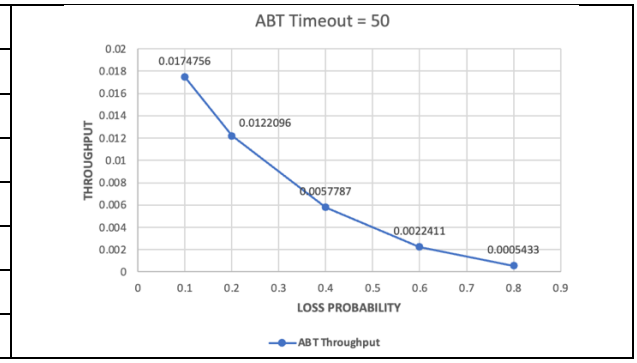


Figure 6. ABT average throughput at different losses with a Timeout of 50

At timeout 50, at the jump we know it's not a good option as the throughput generally decreases across all loss probabilities.

Therefore, for ABT we went with a timeout threshold of 10 as it offered the best performance, balancing the need to retransmit promptly against waiting unnecessarily improving overall throughput at different loss probabilities. Using a higher timeout value even while potentially reducing the number of timeouts it won't necessarily improve throughput, as it can delay retransmissions greatly.

B. Go-Back-N (GBN)

GBN	
Average at Timeout = 10	
W =10	
Loss	Throughput
0.1	0.0159506
0.2	0.0179091
0.4	0.0172643
0.6	0.0155514
0.8	0.0072333

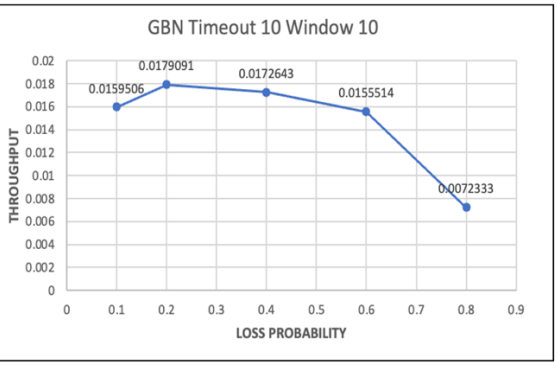


Figure 7. GBN average throughput at different losses with a Timeout of 10

At timeout 10, the throughput slightly decreases and gets worse as loss probability increases. This quicker timeout allows for faster retransmission of lost packets but can also start retransmitting too early resulting in retransmissions before they are necessary if packets are simply delayed rather than lost.

GBN	
Average at Timeout = 20	
W =10	
Loss	Throughput
0.1	0.0199063
0.2	0.0201295
0.4	0.0157015
0.6	0.0154574
0.8	0.0073063

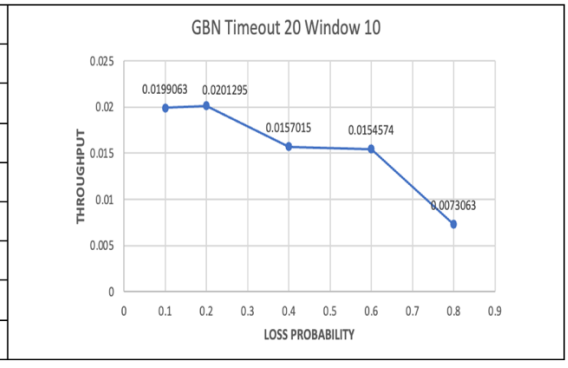


Figure 8. GBN average throughput at different losses with a Timeout of 20

At timeout 20, we see an overall improvement in throughput across most loss probabilities compared to the timeout of 10, suggesting that this timeout strikes a better balance between retransmission reaction and waiting for delayed packets. The performance drop at higher loss probabilities is also less, indicating that the protocol handles retransmissions there more efficiently.

GBN	
Average at Timeout = 50	
W =10	
Loss	Throughput
0.1	0.0198803
0.2	0.0198911
0.4	0.0151893
0.6	0.0082679
0.8	0.0023411

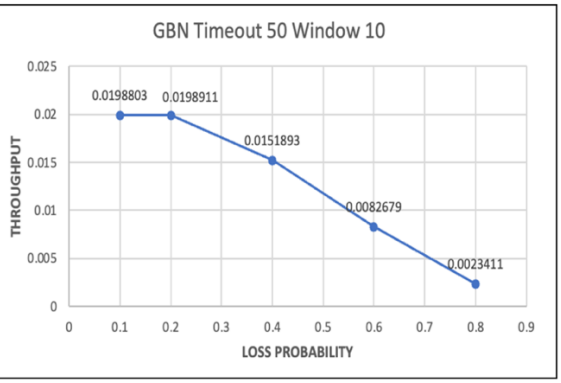
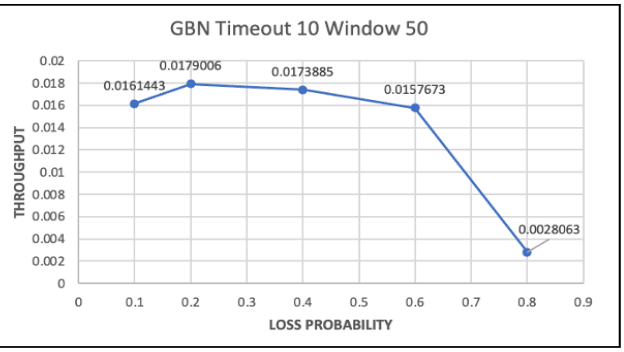


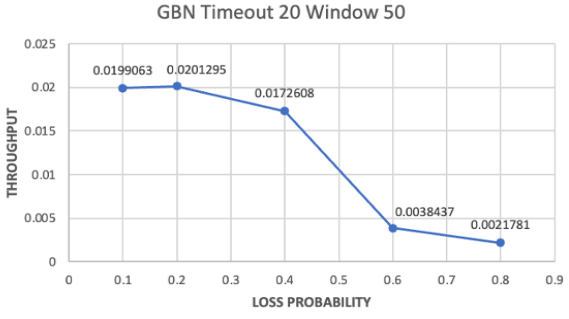
Figure 9. GBN average throughput at different losses with a Timeout of 50

At timeout 50, the throughput decreases significantly across all loss probabilities telling us that it's making it waiting too long before retransmitting packets.

GBN	
Average at Timeout = 10	
W = 50	
Loss	Throughput
0.1	0.0161443
0.2	0.0179006
0.4	0.0173885
0.6	0.0157673
0.8	0.0028063



GBN	
Average at Timeout = 20	
W = 50	
Loss	Throughput
0.1	0.0199063
0.2	0.0201295
0.4	0.0172608
0.6	0.0038437
0.8	0.0021781



GBN	
Average at Timeout = 50	
W = 50	
Loss	Throughput
0.1	0.0199096
0.2	0.0148966
0.4	0.0043115
0.6	0.0032974
0.8	0.0021865

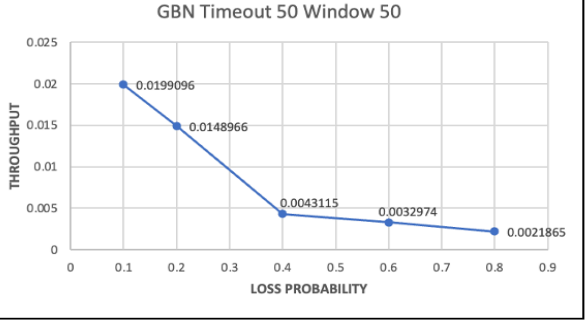


Figure 10. GBN average throughput at different Timeout and losses with a window of 50

From these results we went with a timeout of 20 as it offered the best performance for the GBN protocol. It provided an optimal balance that will minimize unnecessary retransmissions and handle delayed packets better without sacrificing responsiveness to packet loss. This timeout threshold will allow the GBN protocol to maintain a higher throughput across a range of loss probabilities.

C. Selective-Repeat (SR)

SR	
Average at Timeout = 10	
W =10	
Loss	Throughput
0.1	0.0001599
0.2	0.0000902
0.4	0.0000502
0.6	0.000046
0.8	0.0000237

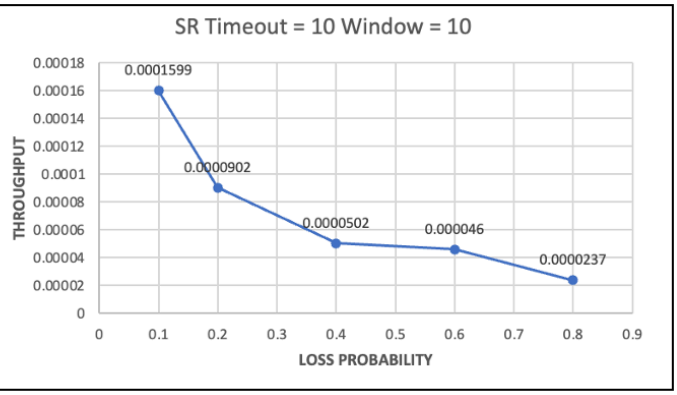


Figure 11. SR average throughput at different losses with a Timeout of 10

At timeout 10, the throughput is extremely low across all loss probabilities. This may be due to the protocol not waiting long enough for acknowledgments before retransmitting, leading to unnecessary retransmissions and increased network congestion, especially as the loss probability increases.

SR	
Average at Timeout = 20	
W =10	
Loss	Throughput
0.1	0.0001212
0.2	0.000082
0.4	0.0000615
0.6	0.000044
0.8	0.0000238

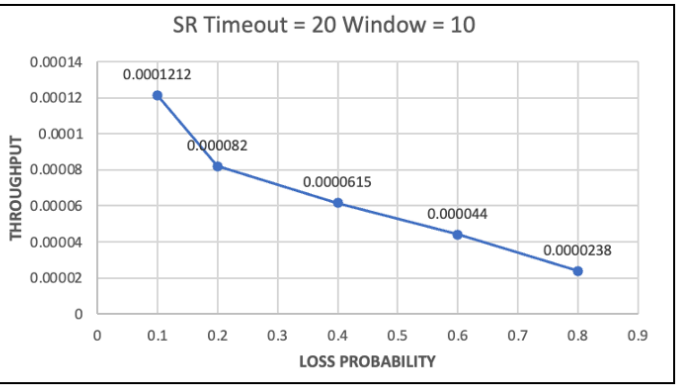


Figure 12. SR average throughput at different losses with a Timeout of 20

At timeout 20, there is improved throughput for all loss probabilities compared to timeout 10, suggesting that it's waiting slightly longer for acknowledgments before retransmitting but it's still not allowing enough time for acknowledgments to arrive.

SR	
Average at Timeout = 50	
W =10	
Loss	Throughput
0.1	0.0001647
0.2	0.0001101
0.4	0.0000659
0.6	0.0000422
0.8	0.000022

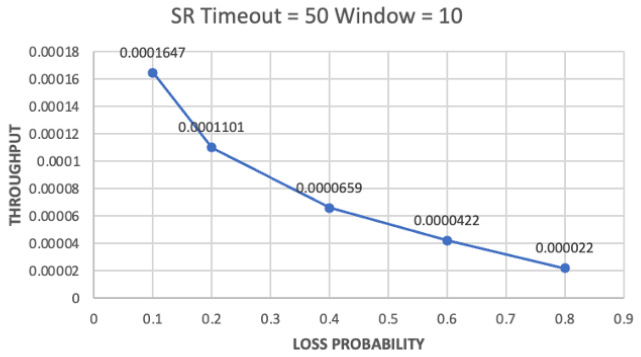
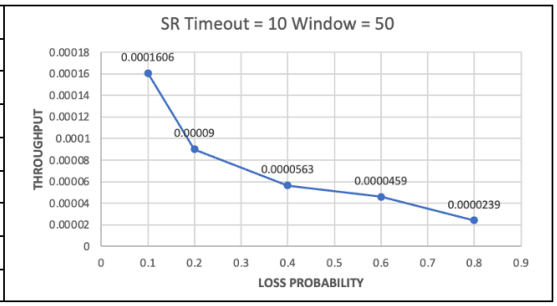


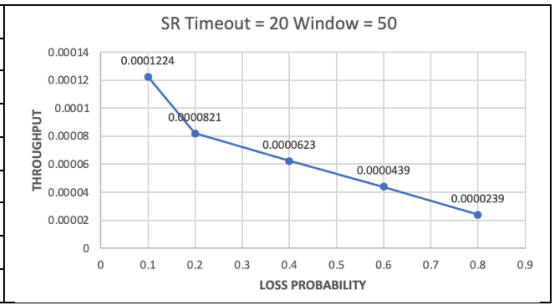
Figure 13. SR average throughput at different losses with a Timeout of 50

At timeout 50, there is some improvement in the throughput for most loss probabilities compared to timeout 20, suggesting that it's waiting slightly longer for acknowledgments before retransmitting. The throughput isn't as high as we would like it as it's still not allowing enough time for acknowledgments to arrive resulting in less packets being delivered. This could also be as a result of our program's inefficiency and bad performance.

SR	
Average at Timeout = 10	
W =50	
Loss	Throughput
0.1	0.0001606
0.2	0.00009
0.4	0.0000563
0.6	0.0000459
0.8	0.0000239



SR	
Average at Timeout = 20	
W =50	
Loss	Throughput
0.1	0.0001224
0.2	0.0000821
0.4	0.0000623
0.6	0.0000439
0.8	0.0000239



SR	
Average at Timeout = 50	
W =50	
Loss	Throughput
0.1	0.0001647
0.2	0.0001096
0.4	0.0000656
0.6	0.0000424
0.8	0.0000239

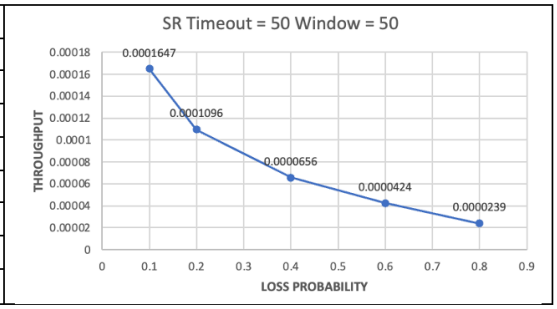


Figure 14. SR average throughput at different Timeout and losses with a window of 50

From these results we went with a timeout of 50 as it offered the best performance for the SR protocol and if we went lower the performance was worst. It provided some balance that will minimize unnecessary retransmissions and handle delayed packets better without sacrificing responsiveness to packet loss. The overall low throughput across all timeout settings in SR could point to inefficiencies or performance issues within the implementation of the protocol itself. This could be because of bad buffer management, inefficient handling of acknowledgments and timeouts, or other algorithmic challenges that are not directly related to the network conditions. Given more time, we would delve deeper into the program's logic and structure to try and identify and rectify these inefficiencies.

IMPLEMENTATION OF MULTIPLE SOFTWARE TIMERS IN SR

In our SR protocol implementation, we had only one hardware timer, and may have many outstanding, unacknowledged packets in the medium, so we had to think about how to use the single timer to implement multiple logical timers. We did some research on the possible ways we would implement it and considered different options like using get_sim_time() to get the current simulation time for each packet and keep an array of it to compare with timer interrupt and find which packet timed out. Instead, we utilized the single hardware timer to manage the timing of multiple packets by using functions like ‘A_output’, ‘A_input, and ‘A_timerinterrupt’, as well as through the use of the ‘is_timer_running’ flag and the ‘A_acknowledged’ array. When sending a packet in ‘A_output’, we check if the timer is running already before starting it, ensuring it only starts with the first packet in the window and not for every subsequent packet. In ‘A_input’, I mark and keep track of packets that have been acknowledged in ‘A_acknowledged’ and If the acknowledged packet is the oldest unacknowledged one, I slide the window forward and restart or stop the timer based on whether there are any unacknowledged packets left that way the timer wont timeout if we still have packets we are waiting for to be ack’ed. The ‘A_timerinterrupt’ function handles the timeouts when the happen based on unacknowledgments and packet loss by checking ‘A_acknowledged’ for the oldest packet that has not been acknowledged yet and retransmitting it if necessary and restart the timer after. This allows us to efficiently manage multiple packet timeouts using just one hardware timer, reducing complexity while ensuring reliable transmission.

PERFORMANCE COMPARISON

Experiment 1

For the first experiment we compared the three protocols by changing the loss probabilities: 0.1, 0.2, 0.4, 0.6 and 0.8. Since window size is not considered for Alternating Bit Protocol, the throughput used for both the graphs will be the same, for Go Back - N and Selective Repeat protocols the window size plays a crucial role in the throughput values, according to the given values of Window Sizes 10 and 50.

Table 1: Window Size 10

Window = 10			
Loss	**ABT**	**GBN**	**SR**
	Throughput	Throughput	Throughput
0.1	0.019898	0.0199063	0.0001647
0.2	0.0199881	0.0201295	0.0001101
0.4	0.0200749	0.0157015	0.0000659
0.6	0.0101858	0.0154574	0.0000422
0.8	0.0025965	0.0073063	0.000022

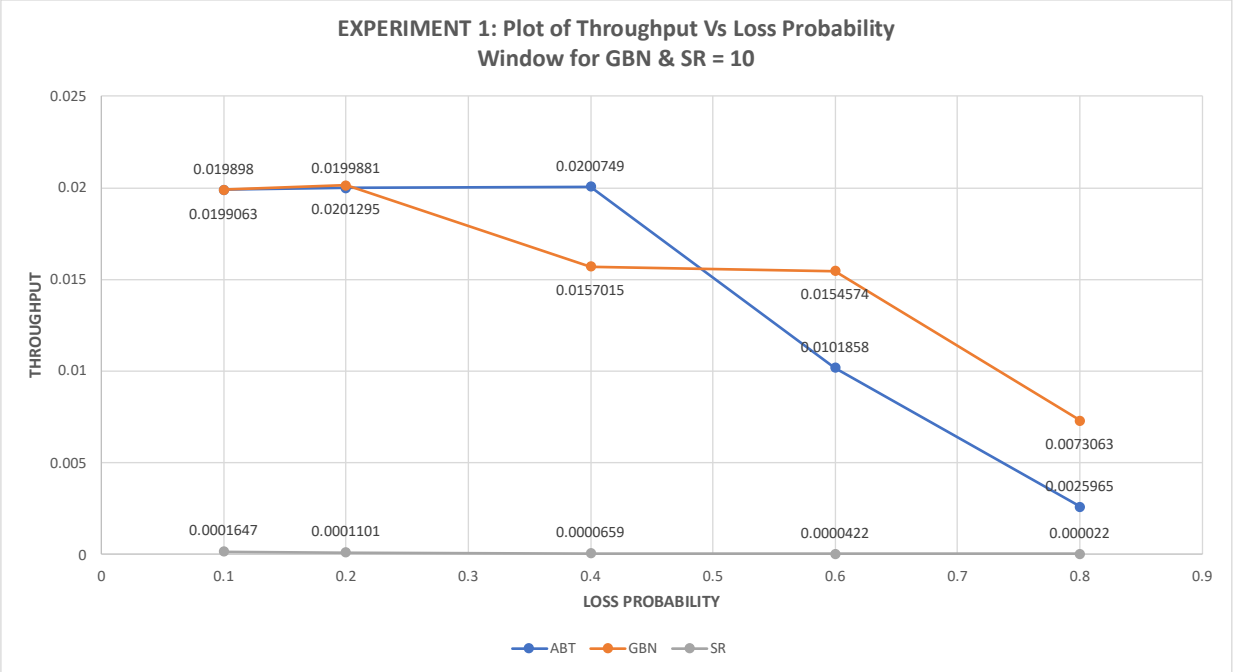


Figure 15. ABT GBN & SR average throughput at different losses with a window of 10

For this graph the window size for GBN & SR is 10 and the loss probability is varied to check the efficiency of the protocols. The corruption rate is set to 0.2 and number of messages is 1000. From the throughput we are getting at different loss probabilities we can see that there is good efficiency provided by ABT & GBN even with increased loss probability. For SR protocol, the general poor throughput in SR across all timeout settings may indicate inefficiencies or performance problems in the protocol's actual implementation. Ineffective acknowledgment and timeout handling, poor

buffer management, or other algorithmic issues unrelated to network circumstances might be the source of this. Even with SR inefficiencies, we can still see that there is a gradual decrease in the throughput with the increase in loss probabilities. This concludes that all the three protocols are functioning correctly as all of them as are giving throughput proportional to the loss probabilities.

Table 2: Window Size 50

Window = 50			
Loss	**ABT**	**GBN**	**SR**
	Throughput	Throughput	Throughput
0.1	0.019898	0.0199063	0.0001647
0.2	0.0199881	0.0201295	0.0001096
0.4	0.0200749	0.0172608	0.0000656
0.6	0.0101858	0.0038437	0.0000424
0.8	0.0025965	0.0021781	0.0000239

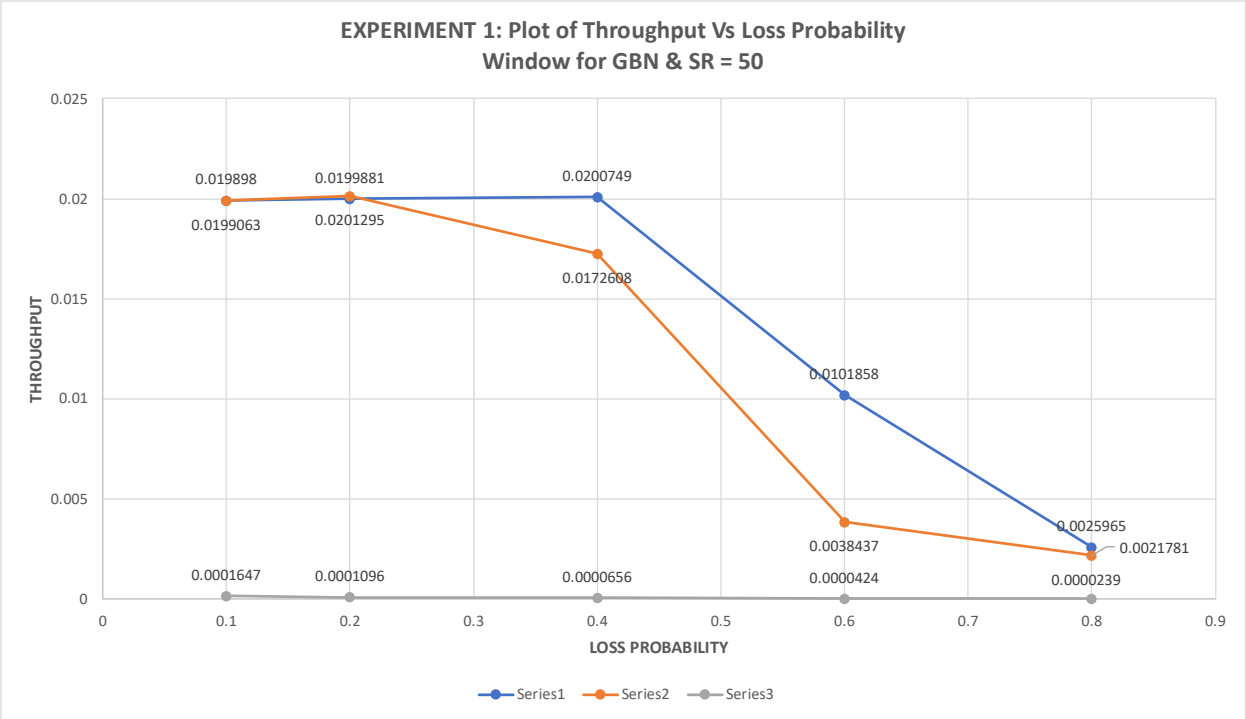


Figure 16. ABT GBN & SR average throughput at different losses with a window of 50

For this graph the window size for GBN & SR is 50 and the loss probability is varied to check the efficiency of the protocols. The corruption rate is also set to 0.2 and number of messages is 1000. From the throughput we are getting at different loss probabilities we can see that there is a good efficiency provided by ABT & GBN even with increased loss probability. For SR protocol, the general poor throughput in SR across all timeout settings may indicate inefficiencies or performance problems in the protocol's actual implementation. Ineffective acknowledgment and timeout handling, poor buffer management, or other algorithmic issues unrelated to network

circumstances might be the source of this. Even with SR inefficiencies, we can still see that there is a gradual decrease in the throughput with the increase in loss probabilities. This shows that all the three protocols are functioning correctly as all of them as are giving throughput inversely proportional to the loss probabilities.

Experiment 2

For the second experiment we compared the three protocols by changing the window size to different values: 10, 50, 100, 200 & 500. Since window size is not considered for Alternating Bit Protocol, so for ABT loss probability will play a crucial role. For Go Back - N protocol and Selective Repeat protocol window size plays a crucial role in the throughput values. There will be 3 graphs with loss probability of 0.2 0.5 & 0.8 respectively.

Table 3: Loss of 0.2

Loss = 0.2			
Window Size	**ABT**	**GBN**	**SR**
	Throughput	Throughput	Throughput
10	0.0199881	0.0201295	0.0001101
50	0.0199881	0.0201295	0.0001096
100	0.0199881	0.0201295	0.0001107
200	0.0199881	0.0201295	0.0001109
500	0.0199881	0.0201295	0.0001108

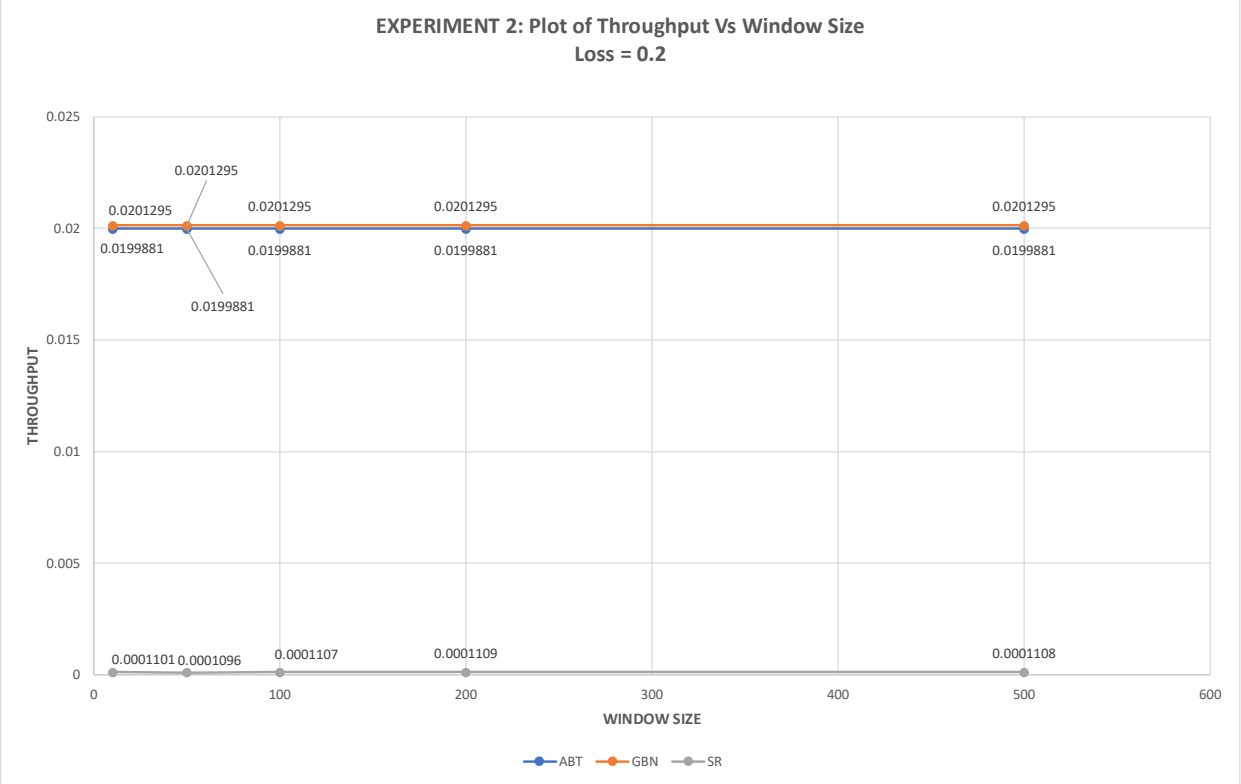


Figure 17. ABT GBN & SR average throughput at different window with a loss of 0.2

For loss at 0.2, ABT is giving a consistent throughput as it is unaffected by the window specifications. For GBN we are still getting higher but a consistent throughput this could be for various reason as this is only happening when the loss probability is 0.2. At some point due to increase in window side there should have been an increase in the throughput. Similarly in SR the throughput is the changing a little but not drastically this could be because of the inefficiencies for the implemented code for SR protocol as mentioned earlier.

Table 4: Loss of 0.5

Loss = 0.5			
Window Size	**ABT**	**GBN**	**SR**
	Throughput	Throughput	Throughput
10	0.0154392	0.0093816	0.0000501
50	0.0154392	0.0081178	0.0000499
100	0.0154392	0.0083715	0.0000501
200	0.0154392	0.0083992	0.0000496
500	0.0154392	0.0083797	0.0000497

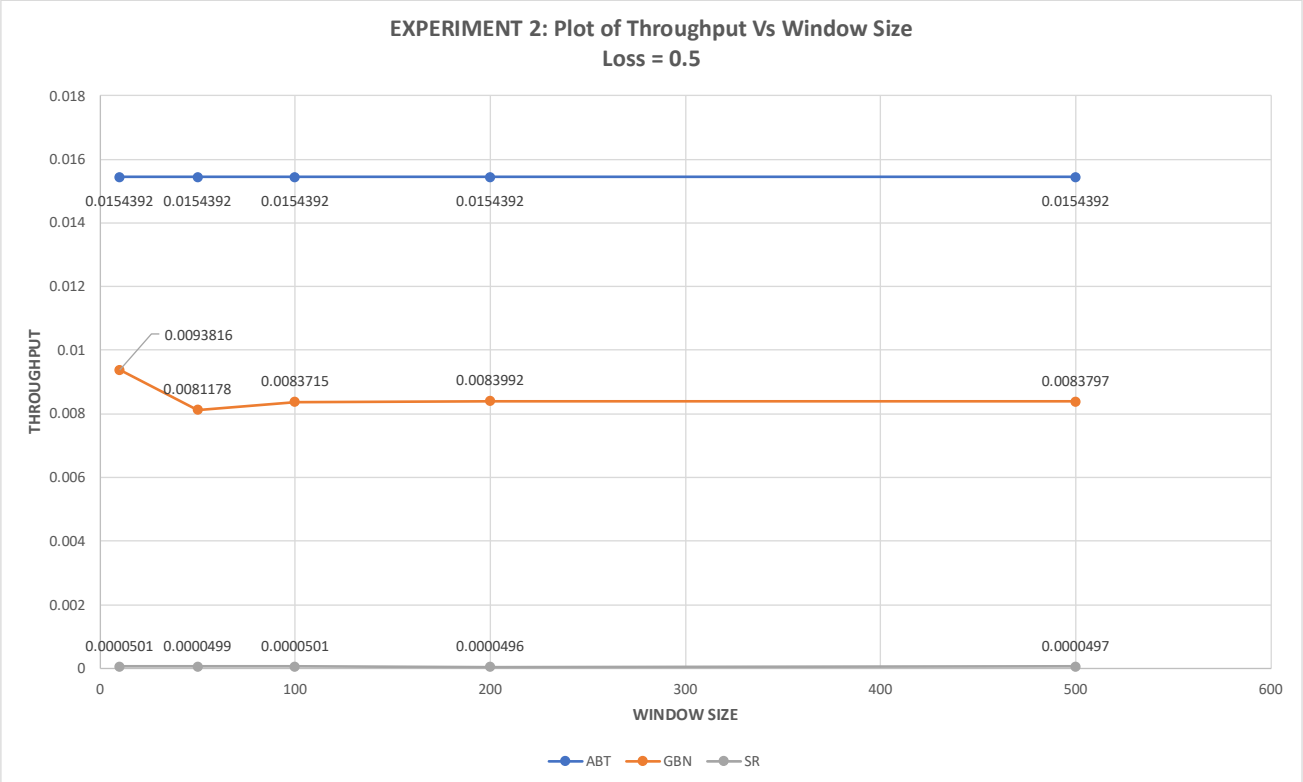


Figure 18. ABT GBN & SR average throughput at different window with a loss of 0.5

For loss at 0.5, ABT is giving a consistent throughput as it is unaffected by the window specifications. For GBN we are getting good throughput that changes with the window size. In SR protocol we can see that there is a good consistency with the changes occurring in the window size. The throughput for both ABT & GBN is good considering moderate loss probability, for SR

the throughputs, even if they satisfy the conditions of the experiment, are very low because of inefficiency of the code.

Table 4: Loss of 0.8

Loss = 0.8			
Window Size	**ABT**	**GBN**	**SR**
	Throughput	Throughput	Throughput
10	0.0025965	0.0073063	0.000022
50	0.0025965	0.0021781	0.0000239
100	0.0025965	0.0018461	0.0000239
200	0.0025965	0.0015771	0.000024
500	0.0025965	0.0015637	0.0000237

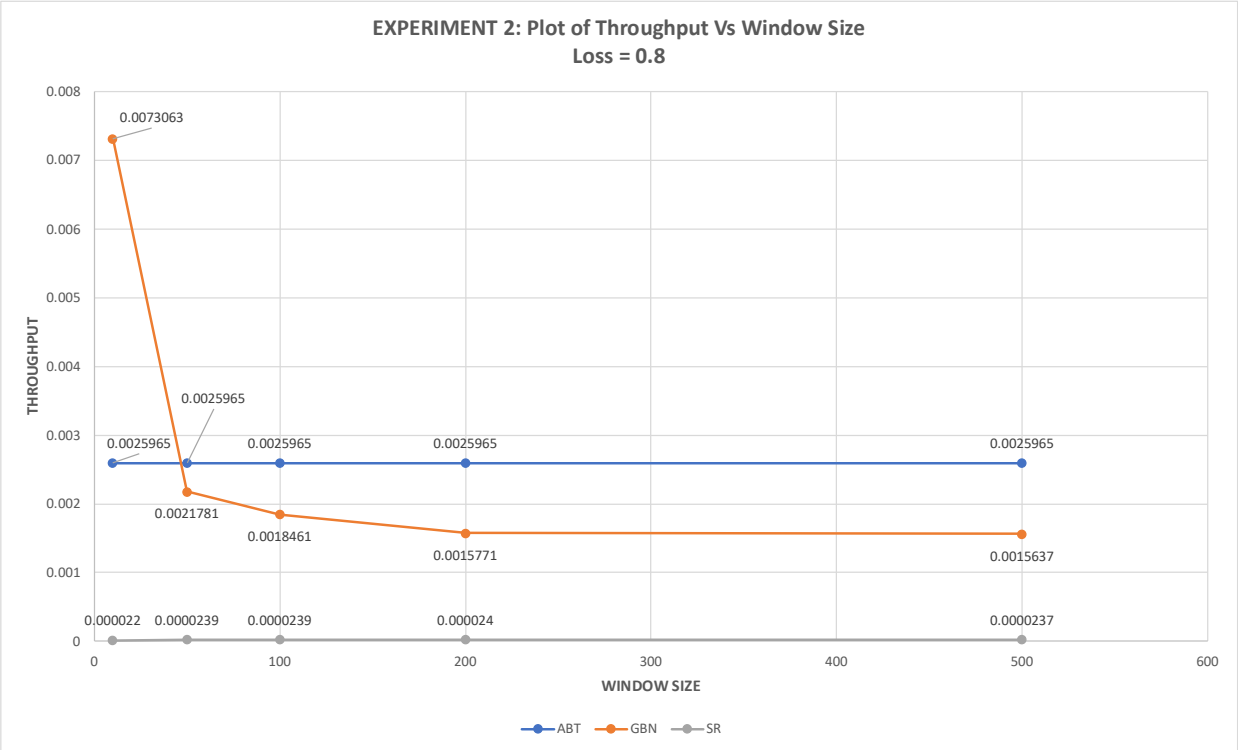


Figure 19. ABT GBN & SR average throughput at different window with a loss of 0.8

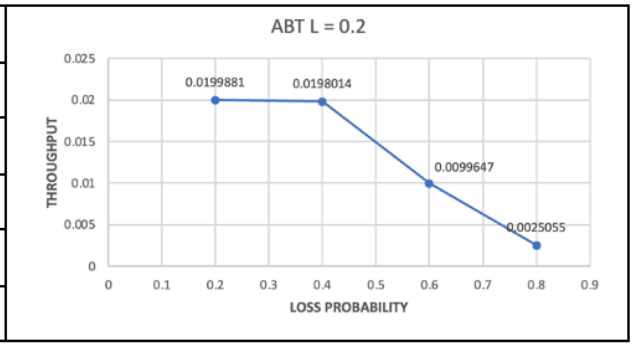
For Loss at 0.8, ABT is still giving a consistent throughput as it is unaffected by the window specifications. For GBN we are getting good throughput that changes with the window size. In SR protocol we can see that there is a good consistency with the changes occurring in the window size. The throughput for both ABT & GBN is good considering very high loss probability, for SR the throughputs, even if they satisfy the conditions of the experiment, are very low because of ineffective acknowledgment and timeout handling, poor buffer management, or other algorithmic issues unrelated to network circumstances might be the source of this.

Extra Tests And Graphs: Protocols Tested For Corruption And Reliability

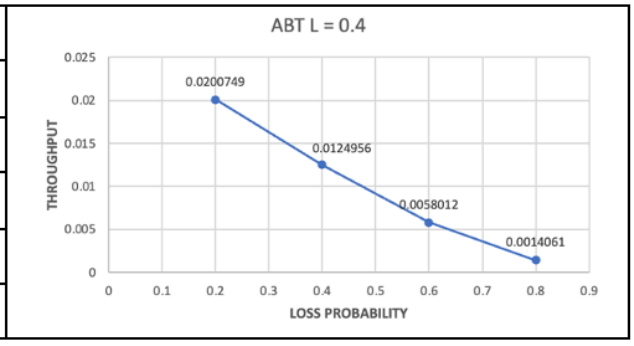
For both the experiments we have either changed the loss probabilities or the window size, but we have not analyzed the performance of the protocols by changing the corruption rate. Analyzing the effect of change in the corruption rate could give us more understanding on the effectiveness of the code.

A. Alternating-Bit Protocol (ABT)

ABT L = 0.2		
Loss	Corruption	Throughput
0.2	0.2	0.0199881
0.2	0.4	0.0198014
0.2	0.6	0.0099647
0.2	0.8	0.0025055



ABT L = 0.4		
Loss	Corruption	Throughput
0.4	0.2	0.0200749
0.4	0.4	0.0124956
0.4	0.6	0.0058012
0.4	0.8	0.0014061



ABT L = 0.8		
Loss	Corruption	Throughput
0.8	0.2	0.0025965
0.8	0.4	0.0014693
0.8	0.6	0.0006727
0.8	0.8	0.0001937

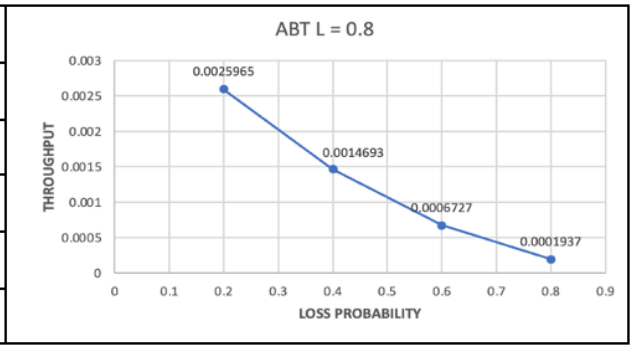
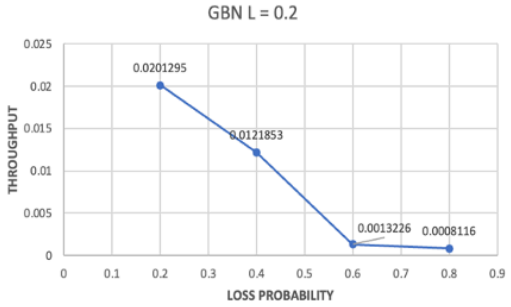


Figure 20. ABT average throughput at different corruption rate with a loss of 0.2, 0.4 & 0.8

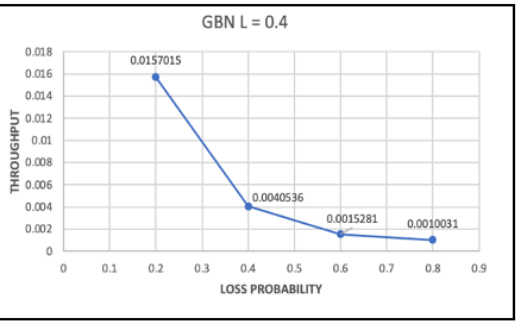
For Alternative Bit Protocol, the two factors that would affect the change in throughput are loss probabilities and the corruption rate. We have already seen the efficiency of the protocol when there is a change in the loss probabilities. Therefore, here we have kept the loss probability constant, and we have changed the corruption rate. Ideally increase in corruption rate should decrease the throughput of the protocol. That is exactly what can be seen in the above graphs. This concludes that the code is working effectively and is correctly modulating itself according to the corruption rates.

B. Go Back N (GBN)

GBN L = 0.2		
Loss	Corruption	Throughput
0.2	0.2	0.0201295
0.2	0.4	0.0121853
0.2	0.6	0.0013226
0.2	0.8	0.0008116



GBN L = 0.4		
Loss	Corruption	Throughput
0.4	0.2	0.0157015
0.4	0.4	0.0040536
0.4	0.6	0.0015281
0.4	0.8	0.0010031



GBN L = 0.8		
Loss	Corruption	Throughput
0.8	0.2	0.0073063
0.8	0.4	0.0052978
0.8	0.6	0.0034074
0.8	0.8	0.0018465

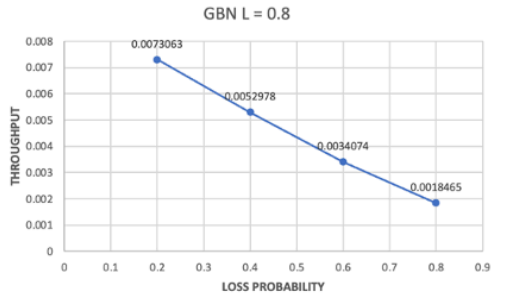
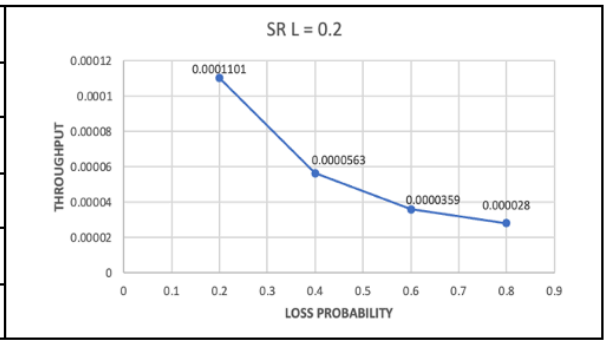


Figure 21. GBN average throughput at different corruption rate with a loss of 0.2, 0.4 & 0.8

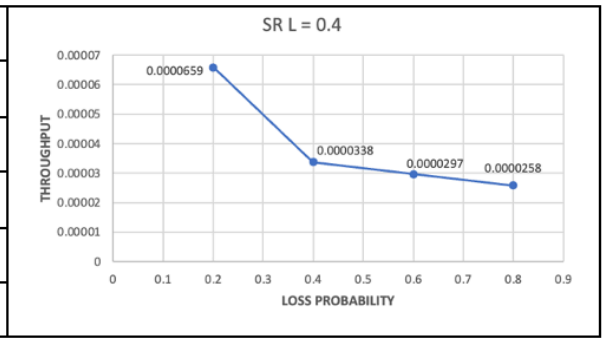
For Go Back -N protocol, the three factors that would affect the change in throughput are loss probabilities, window size, and the corruption rate. We have already seen the efficiency of the protocol when there is a change in the loss probabilities and window sizes. Therefore, here we have kept the loss probability constant, and we have changed the corruption rate. Ideally increase in corruption rate should decrease the throughput of the protocol. That is exactly what can be seen in the above graphs. This concludes that the code is working effectively and is correctly modulating itself according to the corruption rates.

C. Selective Repeat (SR)

SR L = 0.2		
Loss	Corruption	Throughput
0.2	0.2	0.0001101
0.2	0.4	0.0000563
0.2	0.6	0.0000359
0.2	0.8	0.000028



SR L = 0.4		
Loss	Corruption	Throughput
0.4	0.2	0.0000659
0.4	0.4	0.0000338
0.4	0.6	0.0000297
0.4	0.8	0.0000258



SR L = 0.8		
Loss	Corruption	Throughput
0.8	0.2	0.000022
0.8	0.4	0.0000201
0.8	0.6	0.0000201
0.8	0.8	0.0000201

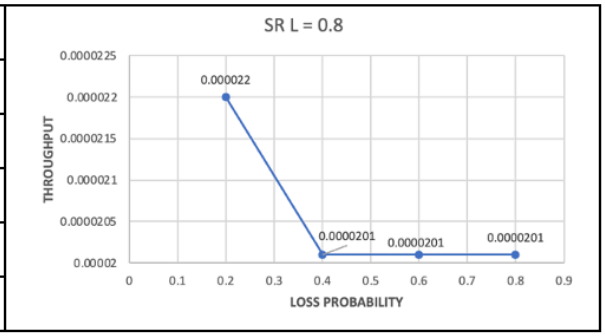


Figure 22. SR average throughput at different corruption rate with a loss of 0.2, 0.4 & 0.8

For Selective Repeat protocol, the three factors that would affect the change in throughput are loss probabilities, window size, and the corruption rate. We have already seen the efficiency of the protocol when there is a change in the loss probabilities and window sizes. Therefore, here we have kept the loss probability constant, and we have changed the corruption rate. Ideally increase in corruption rate should decrease the throughput of the protocol. That is exactly what can be seen in the above graphs. This concludes that the code is working effectively and is correctly modulating itself according to the corruption rates.

SUMMARY

We were able to implement and test the three reliable data transport protocols; Alternating-Bit (ABT), Go-Back-N (GBN), and Selective-Repeat (SR) in C, under the simulated network environment. These implementations were thoroughly tested against different scenarios, including

high loss and corruption rates to demonstrate the efficiency and robustness of these protocols and how they handle errors and out-of-order packets. Our findings revealed that the ABT protocol is more stable, effectively balancing retransmissions and throughput across varying loss probabilities. The GBN protocol excelled in handling errors and out-of-order packets, particularly with larger window sizes, demonstrating adaptability and efficiency under stress conditions. However, the Selective-Repeat (SR) protocol, despite its potential, showed lower throughput indicative of underlying inefficiencies, possibly due to suboptimal buffer management or acknowledgment handling and inefficient implementation so the program does not handle losses and corruption very well as with little losses we found the SR protocol had really low throughput and all packets getting delivered with loses and corruption had very little packets delivered to the application layer at B. We recognize there are various areas to work on and improve, with more time we would look deeper into the program's logic and structure to address inefficiencies, particularly in areas like buffer management, timer handling and managing window sizes better. These insights highlight ABT's robustness, GBN's adaptability, and areas for improvement in SR. For a more detailed analysis and comprehensive insights, refer to the 'Analysis and Report' section of our documentation. This attempt has allowed us to increase our understanding of the intricacies of network protocols, particularly in implementing reliable data transport mechanisms. It has also provided us with a valuable opportunity to apply the theoretical knowledge gained in class to a practical application.