Report of PA2: Reliable Transport Protocols

We have read and understood the course academic integrity policy

* Taoran Liu Jinbo Li

***Abstract*—This report briefly describes the logic and data structure of three programming. In the second part, we compare the three protocols' throughput under different loss rates and window size values. We give several reasonable explanations for their performance based on the data and graphs.**

***Index Terms*—ABT GBN SR Throughtput analysis**

I. Introduction

Reliable transport protocol is a protocol that provides reliability on a best-effort network. In this PA, we program three protocols: Alternating-Bit Protocol, Go-Back-N, and Selective Repeat. Alternating-Bit Protocol is a simple network protocol operating at the link layer that retransmits lost or corrupted messages using FIFO semantics. Go-Back-N is a link layer protocol that uses a sliding window method for reliable and sequential delivery of data frames. Selective Repeat protocol allows the receiver to accept and buffer the frames following a damaged or lost one.

II. Implementation ABT Protocol

A. Timeout Scheme

For the ABT protocol, we set the value of timeout to 20.0. When sender A does not receive the correct ACK, A will resend this packet to B after timeout. After resending the packet, A starts a new timer with a value of 20.0.

B. Software Logic

For ABT protocol, we adopt a queue buff for sender A. When A_output is called, we put the message into a packet and push the box into the buff queue. When StateA equals 0, which means A waits for a message from layer 5, A sends the first packet in the queue to receiver B. When A receives the correct ACK, which means the ACK sequence number is correct and not corrupt, A pops this packet from the buff queue.

For receiver B, B has the value of seqB, which is the packet sequence number B wants to receive. If B receives a packet with the same sequence number as seqB, B will send an ACK with the acknowledge number seqB to A. Then, B will deliver this packet to layer 5. If B receives a packet with a sequence number less than the value of seqB, B will send the last ACK to A. If B gets a corrupt packet, B just ignores this corrupted packet.

Identify applicable funding agency here. If none, delete this.

C. Data Structure

We use a queue as the data structure.

$QueueInit()$:
initial the buff queue of sender A. This function is in the A_Init.

$QueuePush()$:
when function $A\_output()$ is called, push the packet into the queue. Please note that the packet that enters the queue first exits the queue first.

$QueueFront()$:
gets a packet from the queue. Please note that it will not pop this packet because this packet may be resent to B.

$QueuePop()$: pops a packet from the buff queue. We only run this function when A receives a correct ACK in the ABT protocol.

$QueueEmpty()$:
determines whether the queue is empty. If the queue is empty, A will not send any packets to B and wait for the messages from layer 5 (application layer).

III. Implementation GBN Protocol

A. Timeout Scheme

Similar to the ABT protocol, we've set the timeout value to 20.0. However, in the case of GBN, only the base file within a transmission window can trigger timer interruption. When the timer interruption is triggered, all packets within that window are retransmitted. Upon receiving an ACK for the base file, the window moves, and a new timer is initiated.

B. Software Logic

On the sender A side, we've implemented a singly linked list as a buffer, employing a dual pointer method to manage the moving windows. The left pointer of the moving window represents the base file packet awaiting acknowledgment, while the right pointer denotes the last packet sent to layer 3. Upon receiving a packet, the program immediately places it at the back of the buffer. If the window has not reached its maximum size, the right pointer expands the window. When this packet becomes the first in the window, a timer is initiated. Upon receiving an uncorrupted ACK, the timer stops, and the left pointer moves and pops out the previous packet. This process continues until the packet in question is removed, given GBN's use of an accumulating ACK mechanism. After removing the packet, the right pointer moves, sending all accumulated packets within the next

window size and then initiating a new timer.
On the receiver B side, the program remains exactly the same as ABT, with no alterations.

Regarding the timer interrupt program, each time the interrupt is triggered, the right pointer returns to the left pointer and then moves in steps equivalent to the maximum window size, sending all packets within that window and initiating a new timer."

C. Data Structure

The linked list structure mirrors the Queue structure we established for the ABT protocol. In this implementation, we only introduce two pointers to define the boundary of the moving window. Notably, this technique is only implemented on the A side as a buffer, maintaining simplicity on the B side.

The time complexity of the timer interrupt will be represented as $N \times n$, where N denotes the number of times the interrupt is triggered, and n represents the maximum window size. In the worst-case scenario, the overall time complexity will amount to $N^2 \times n$.

IV. Implementation SR Protocol

A. Timeout Scheme

For the SR protocol, every packet is assigned the same timeout value, which is equal to $Interval$. However, during our experiments, particularly when using large window sizes, we noticed instances where certain packets were retransmitted despite the receiver (B side) having already received and sent back the acknowledgment (ACK). This phenomenon can be attributed to using the same $Interval$ value as in ABT and GBN protocols. With larger windows, the transmission delay tends to increase, subsequently leading to an increase in the overall Round-Trip Time (RTT). To address this, we've adopted a strategy: maintaining a smaller $Interval$ value to ensure efficient program execution with smaller windows, and setting a larger $Interval$ value for window sizes larger than 100.

B. Software Logic and Multiple Time Implementation

For the SR protocol, implementing software multiple timers poses a challenging task. On the sender side (A), we maintain the singly linked list as the buffer and utilize a dual pointer system for the moving window. However, instead of employing a single timer for the same window, we record the simulation time for each packet's sending time as a timestamp. When starting a new timer, we calculate the remaining time by subtracting the time elapsed between the timestamp and the current time from the total $Interval$. To facilitate this, we've developed a novel data structure to manage which packet triggers the timer interrupt and which timestamp is used to calculate the next timer.

Within the A side, the singly linked list and dual pointer remain unchanged, while a rotational dual-linked list serves as the timer calculator. Upon receiving a new packet, we place it in the buffer. If space remains in the window, we move the right pointer to add the packet to the window's tail and also add it to the tail of the rotational dual-linked list. Upon sending the packet, we record and attach a timestamp. If this packet initiates the window, a new timer is started. When receiving and placing packets into the window and rotational linked list, this process occurs from the backward end, with packets closer to the rotational linked list's head having earlier timestamps. Upon receiving an ACK, the pointer searches the singly linked buffer within the window size and flags the corresponding packet as received. Subsequently, the flagged packet is removed from the rotational dual-linked list as it's no longer needed for timer calculations. If the acknowledged packet is the base file, the left pointer moves, pops out all accumulated packets with the acknowledgment flag in sequence, and then the right pointer moves to add new packets to the window and rotational dual-linked list. A new timer is initiated for the next expected acknowledgment packet. The timer calculation uses the $Interval$ minus the duration between the timestamp of the rotational linked list's head packet and the current simulation time, as the head packet holds the earliest timestamp and is set to expire next.

Regarding the rotational dual-linked list in the A side, we didn't create a separate linked list and nodes. Instead, we attached the rotational dual-linked list's pointer system to the original singly linked list. Essentially, nodes within the window possess two different pointer systems: one from the singly linked list and the other from the rotational dual-linked list, without an interface. Utilizing this data structure not only conserves memory but also allows us to traverse the singly linked list to find and extract packets while using the dual-linked list to calculate and retransmit expired packets, significantly reducing time complexity in each scenario.

During a timer interrupt, the program retransmits the header packet in the rotational dual-linked list, assigns a new timestamp, and rotates it to the tail. Subsequently, it uses the present header's timestamp to initiate a new timer.

On the receiver side (B) for the SR protocol, a separate singly linked list serves as the buffer. Initially, B maintains a sequence number for the next expected received packet. Upon receiving an uncorrupted packet, B immediately sends an ACK. B then searches its buffer linked list, creating empty spaces for sequence numbers smaller than the received packet until placing the received packet in the correct order. Afterwards, B checks its buffer's head to determine if the expected packet has been received. If received, it's sent to Layer 3, repeating this process until the next expected packet's place is empty."

C. Data Structure

Our node structure for rotational dual-link list:

```
typedef struct QueueNode {
    QDataType val;
    float timeStamp;
    int buf_seq;
    int check_arr;

    // pointer for singly linked list
    QueueNode* next;
    // pointer for rotational linked list
    QueueNode* loopnext, *loopbefore;
} QueueNode;
```

Rotated()

The function we rotate the rotational Dual-linked list, pop the head, and connect it behind tail. It only operate the pointer in rotational Dual-linked list

cutAndConnect()

The function we disconnect node with the rotational Dual-linked list. It only operate the pointer in rotational Dual-linked list

QueuePutInPlace()

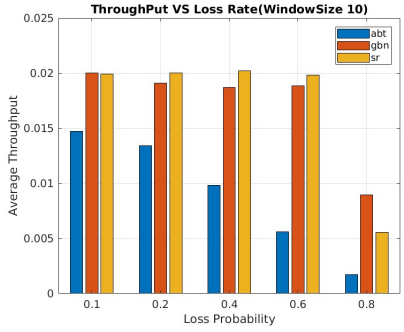
The function we use for put received into right place in B side buffer.

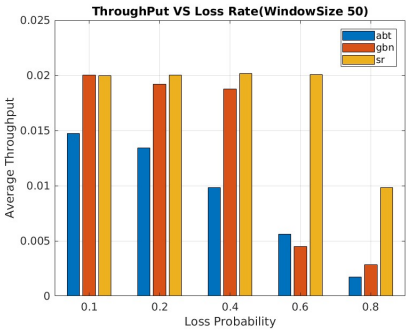
V. Performance Comparison

In real life, regarding transport protocols, varying packet loss rates can have distinct effects and result in differences in transportation throughput. Additionally, different protocols with varying window sizes also exhibit different efficiencies. In this study, we test three protocols under varying loss rates and using different window sizes (for GBN and SR), comparing the average throughput and analyzing the results.

A. Experiment 1: Throughput and Loss Probabilities

In this case, With loss probabilities: 0.1, 0.2, 0.4, 0.6, 0.8, we compare the 3 protocols' throughputs at the application layer of receiver B. And use 2 window sizes: 10, 50 for the Go-Back-N and the Selective-Repeat protocol. We use histogram plot the result of Throughput against loss probabilities.





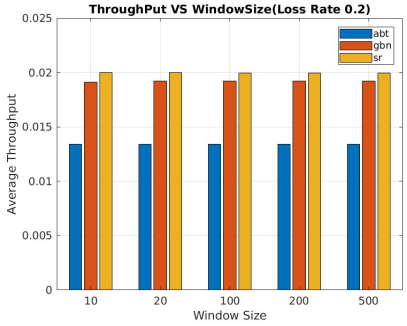
The histograms clearly demonstrate that as the loss probabilities increase, the average throughput decreases. In most scenarios, the SR protocol outperforms GBN, and GBN, in turn, demonstrates better performance than ABT.

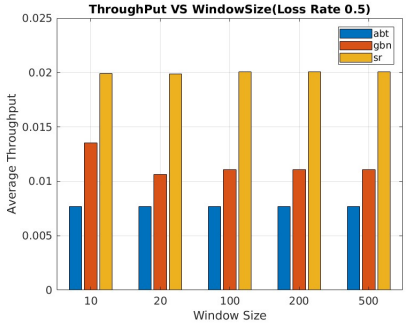
In the first histogram, when loss probabilities range between 0.1 and 0.6, the average throughput of SR and GBN remains relatively stable. This stability is attributed to these protocols effectively delivering nearly all packets to the receiver, approaching the theoretical optimal value even at relatively low loss rates. Consequently, the change in loss probabilities does not significantly impact the average throughput of SR and GBN. However, for the ABT protocol, the decrease in throughput with rising loss rates is strikingly evident. This decline is primarily due to the ABT protocol's longer total waiting time, sending one packet at a time and waiting for acknowledgment. The accumulation of time for timeouts or retransmissions exacerbates this effect. In contrast, the GBN and SR protocols utilize a sending window, allowing for overlapping waiting or timeout times within the same window, resulting in a shorter total waiting time. Higher loss probabilities in the ABT protocol lead to increased timeouts and retransmissions, resulting in a noticeable decline in average throughput.

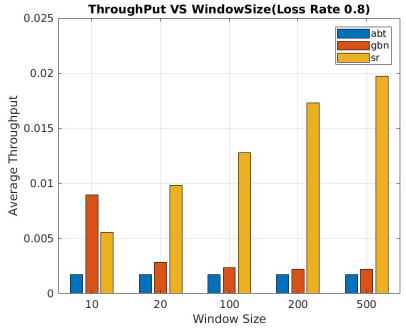
In the second histogram, the impact of loss probabilities is more pronounced in GBN compared to SR, especially with a larger window size. In GBN, triggering a timeout requires the resend of all packets in the window, using a fixed timeout time for each packet within that window. On the other hand, in the SR protocol, only the timed-out packet is retransmitted, and a timer is set solely for the next expected timeout packet using its timestamp. With an increased window size, the GBN protocol experiences a renewal and elongation of the average timeout time for packets within the window, unlike SR. As loss probabilities rise, these accumulated timer issues within the GBN protocol significantly deteriorate its throughput performance.

B. Experiment 2: Throughput and Window Size

In the second experiment, we set the window size:10, 50, 100, 200, 500 with a loss rate of 0.2, 0.5, 0.8 for ABT, GBN, and SR. We use a histogram to plot the throughput result against window size and compare the three protocols' throughput at the application layer of receiver B.







At loss rates of 0.2 and 0.5, an increase in window size results in stable throughput across all three protocols. When the loss rate is 0.2, both GBN and SR achieve their highest throughput, contributing to the observed stability. At a loss rate of 0.5, despite GBN having only half the throughput compared to the previous scenario, its throughput remains stable. One plausible explanation for this lies in the moderate nature of the 0.5 loss rate. The base file in the GBN window is less prone to loss, allowing the window to progress upon receiving acknowledgments, thus renewing the timer without accumulating timeout instances. Consequently, the lower likelihood of buffer A's base timing out reduces subsequent packet retransmissions, minimizing the impact on overall throughput.

Remarkably, at a loss rate of 0.8, SR experiences increased throughput with larger window sizes. The expanded window enables SR to transmit more packets within the same timeframe, leading to increased retransmissions after timeouts. Moreover, as more packets are transmitted within the same window, bandwidth utilization improves. Additionally, the SR protocol assigns a unique timer to each packet, resulting in almost complete overlap of waiting times for each packet within a window (considering very short transmission delays). Consequently, timeout waiting times do not accumulate, contributing to an overall increase in throughput with window size.

On the other hand, at a loss rate of 0.8, the throughput of GBN decreases because receiver B does not have a buffer, which means more packets (even ACKed packets) are retransmitted. As the window size gets larger, more packets will be retransmitted, which means it will waste more time; thus, the throughput decreases.

It's noteworthy that at a loss rate of 0.8 and a window size of 10, GBN performs better than SR. This may be attributed to GBN's higher retransmission rate in scenarios with relatively high loss rates, resulting in shorter average waiting times for each packet. For instance, GBN retransmits all ten window packets only after the base file triggers a timeout, while SR waits for individual packet timeouts before retransmitting. However, the discernible difference in throughput in the histogram suggests that this advantage is not significant. With larger window sizes or relatively lower loss rates, SR still demonstrates better performance.