# Programming Assignment 2- Report

by

[Armaan Goyal]
[armaango-50170093]
[MNC- CSE 589]

**I have read and understood the course academic integrity policy.**

## TABLE OF CONTENTS

INTRODUCTION AND TIMEOUT SCHEMES 1

INTRODUCTION 1
TIMEOUT SCHEMES 1
TIMER STRATEGY FOR SR PROTOCOL 3

EXPERIMENT 1 5

EXPERIMENT 1 – PART 1 5
EXPERIMENT 1 – PART 2 6

EXPERIMENT 2 8

EXPERIMENT 2 – PART 1 8
EXPERIMENT 2 – PART 2 9
EXPERIMENT 2 – PART 3 10
ADDITIONAL ANALYSIS 11

## INTRODUCTION

In this programming assignment we have been asked to implement using code, three reliable data protocols, namely Alternating Bit protocol (ABT), Go Back N protocol (GBN) and Selective Repeat protocol (SR).

We were given a simulator code which tries to simulate the sending and receiving of data from a sender to a receiver where actually the two should be separate machines yet we simulate both of them on a single machine. We were also given a basic structure for all the three protocols, which included all the different functions or routines that we needed to implement. We have implemented only unidirectional data transfer i.e. 'A' will always send the data packets and 'B' will receive the data and acknowledge the packets. The main motive is to ensure reliable in order delivery of packets and then analysis of the performances of all the three protocols under different parameters.

## TIMEOUT SCHEMES

For implementing all the three protocols I have implemented a very simple static timer value on a basic level. Yet the values and implementation are varied across the three protocols.

1. For ABT I decided to go ahead with a static timeout value of 12 time units. The reason for choosing this value is that since we were given that one way delay of each packet can be assumed to be 5 units so allowing some additional window in the timeout value seemed reasonable enough to account for any calculations that might have to be performed to check for the integrity of the data at both ends and also to account for any delays in the medium. Also since in ABT only a single packet is sent out in the medium at one time and the next can only be sent out only after the first is successfully acked, a fixed timer makes sense. I tried out several values ranging from 18 to 10 and based on the initial performances I visualized, I went ahead with the above-mentioned value, 12.0 time units.

2. Moving on to the next protocol, i.e. GBN, I have again used a static timeout value for the implementation but here I have varied the timeout values for different window sizes. We were given few window values to test for, like 10, 20, 100, 200 and 500. In GBN we use a window of messages to be transmitted instead of a single message like in ABT, and if any one message times out, all the messages after the said message have to be retransmitted, so I analyzed and realized that it is better to increase the timeout values for increasing window sizes.

For this purpose I started my timeout value from 20 for a window size of 10 and kept on increasing the same by a step value of 2 for each different window size i.e. value 28 for a window size of 500.

I started the base value at 20 time units to allow for possible retransmissions for any losses that might occur.

3. For the last protocol, Selective Repeat, I again used a constant timeout value across the entire implementation. For this protocol, I did not need to use varying timers for different values because here we implement a separate software timer for each packet that is sent. Therefore a single fixed timeout value of 16 time units was used. This value was also fixed after running several tests with varying performances.

## TIMER STRATEGY FOR SR PROTOCOL

For the implementation of SR protocol we needed to implement a strategy using which a single hardware timer provided by the OS could be used to implement multiple software timers for each packet that was being sent in the varying window sizes. This means at one point we could have upto 500 timers running for each packet on the fly.

To implement this, I used a structure, which includes the default packet structure provided in the basic code. Additionally it contains variables that provide additional information about the packet like, the timestamp at which it was sent, whether the timer is running for that particular packet, whether the packet has been successfully acknowledged among other parameters. The struct definition is given below:

```
struct packetData {
    struct pkt packet;
    int sentBit;
    int canBeSentBit;
    int ackedBit;
    float timestamp;
    bool timerRunning;

};
```

The sentBit, canBeSentBit and ackedBit are integer variables used as binary variables with possible values 0 and 1. 0 represents a not sent, cannot be sent and not yet acked values whereas 1 represents the opposite. Timestamp as the name suggests, gives the time of sending the packet. timerRunning is a bool type variable which specifies whether the actual timer is running specific to this particular packet or not. The initial value for the timestamp variable is -1 which specifies that it has not yet been sent and is still either in the window or the buffer used to store the messages.

The strategy used here is that after a packet has been sent and not successfully acked, it is considered to be on the fly. So when the first packet is sent from A, the timer is started with the default timeout value. Now if before the expire of this timeout, a successful ack is received and assuming that more packets have been sent on the fly, the system, loops through the timestamp values of the sent packets and checks for the minimum timestamp value packet which has not yet been acked. Then it calculates how much time has elapsed since the time the packet was sent, and it restarts the hardware timer with a new time out value which is the time remaining for the timer to time out, i.e. it calculates the difference of the default timeout and this time remaining value.

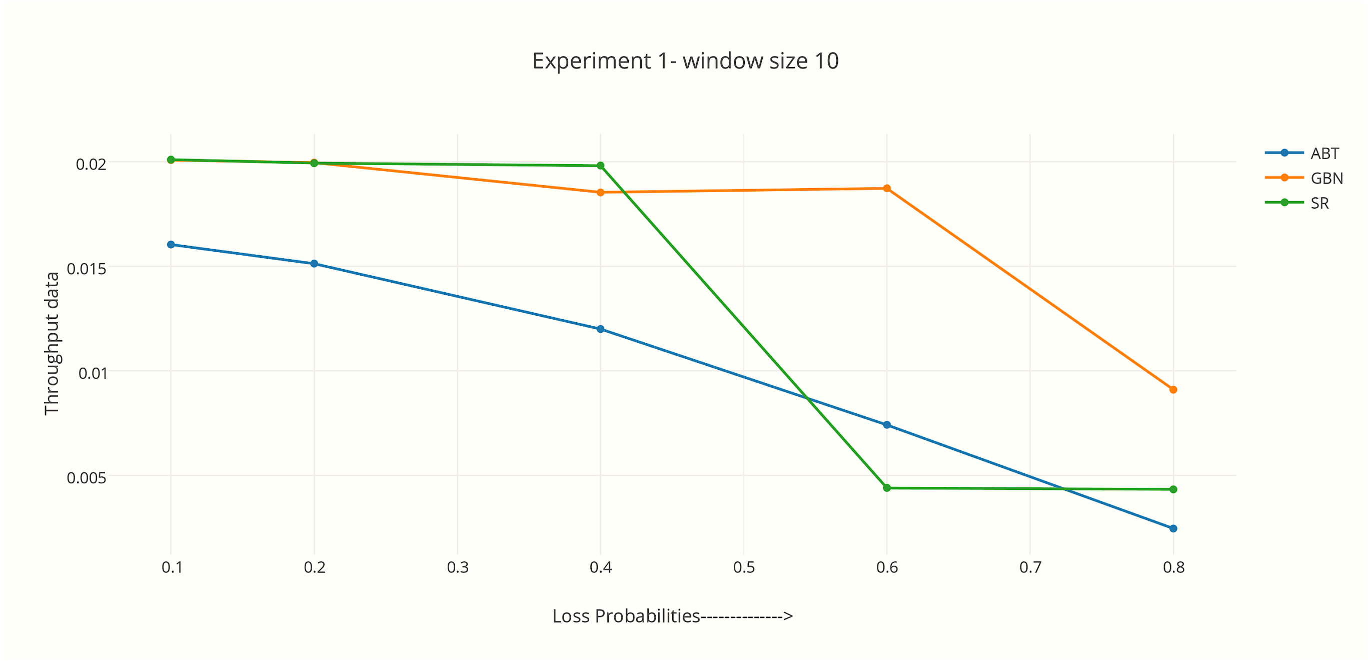
Even if a timer interrupt occurs, the same approach is followed where the minimum timestamp value is calculated and the timer is restarted. If in either case there is no unacked packed on the fly, the timer begins with the default timeout value(16 in this case). To implement this strategy, I have also used an additional Boolean variable **isTimerRunning**, which specifies whether the hardware timer is running at a given time or not. It comes in handy to decide whether to stop and restart the timer or simply start the timer.

This strategy relies on the local time, which is obtained using the get sim time function provided by the simulator.

## EXPERIMENT 1

### PART -1

The first Experiment for the performance comparison asks us to keep the corruption probability fixed at 0.2 and vary the loss probability between the following values, {0.1, 0.2, 0.4, 0.6, 0.8}. The window size is kept fixed at 10 for the first part. The throughput graph for the various runs' data averaged is shown below:



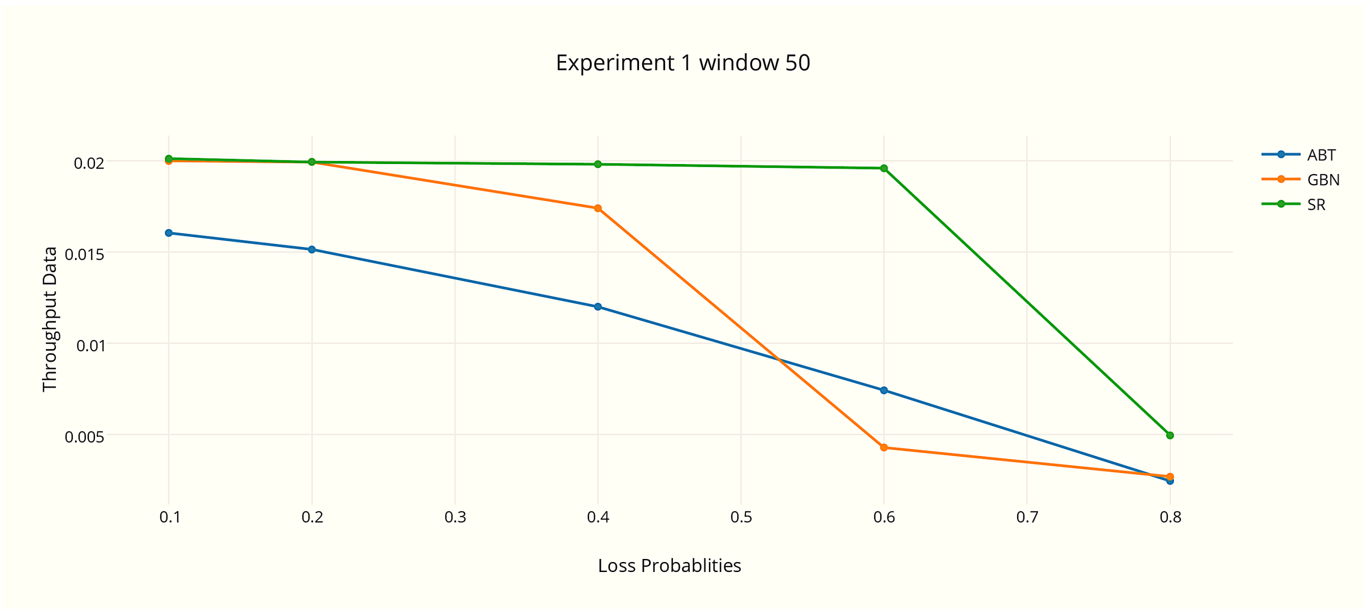
As is clearly evident from the graph, ABT shows a gradual decrease in the throughput data, which is expected since the number of retransmissions increases with the increasing loss, and hence the total time to send the packets increases thus reducing the throughput.

For the GBN protocol, for low loss values the performance seems optimal but as the loss is increased beyond 0.5, there is a sudden drop. From the data received from the test run, I noticed that the number of retransmissions increases by a large number and therefore the throughput value falls so much.

Selective repeat is an efficient protocol but window size 10 seems to be a less than adequate value for the protocol since it can be seen that the performance takes a nosedive after the loss value goes beyond 0.4, even though it seems to stabilize between 0.6 and 0.8 values. This is because once the probability of loss increases beyond half or 0.5, the sender window takes a lot of time to advance since it has to wait for the base sequence number packet to get acked before moving further. The number of retransmissions is obviously lower here, yet the throughput isn't that good at high values because of the slow moving window both at the sender and receiver ends.

## PART - 2

The second part of the first Experiment for the performance comparison asks us to keep the corruption probability fixed at 0.2 and vary the loss probability between the following values, {0.1, 0.2, 0.4, 0.6, 0.8}. The window size is kept fixed at 50 for this part. The throughput graph for the various runs' data averaged is shown below:



Here again, the ABT performance is the same,(not affected by window sizes).

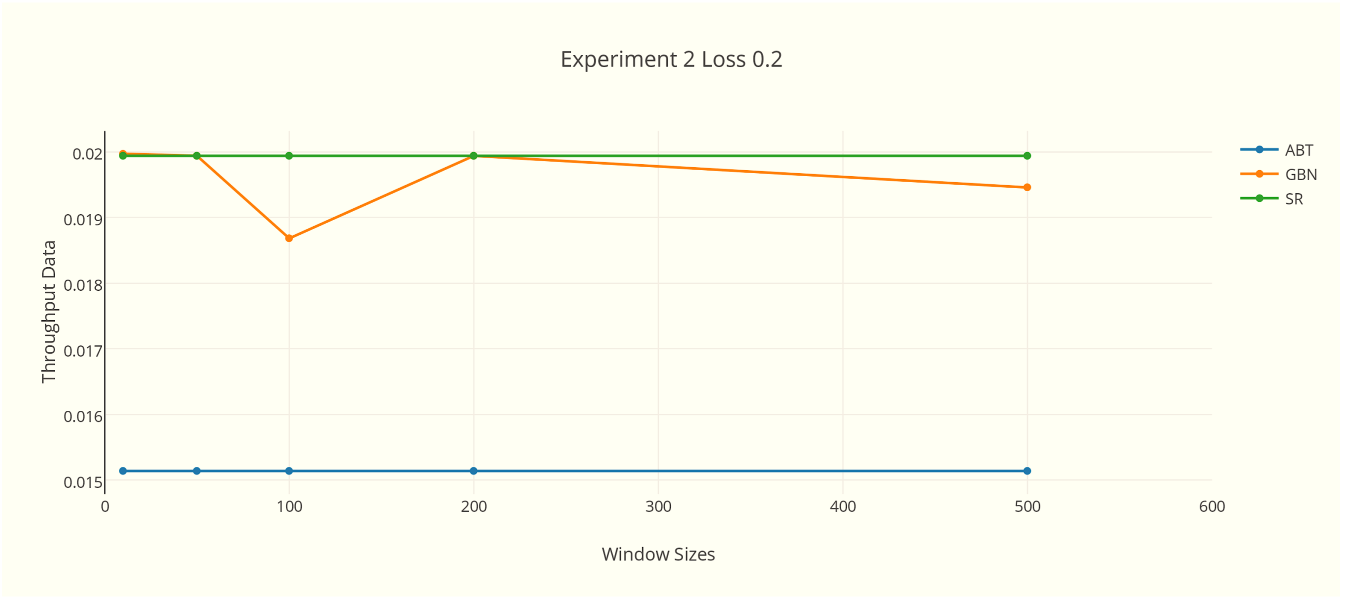
GBN performance for this case can be seen to be falling quicker as compared to the first case with window size 10. This is expected since the larger the size of the window in GBN, the more the number of retransmissions will be at higher losses.

For SR, we can see the behavior or the performance improve as compared to the window size 10 and it is in line with our expectations too. This is so because for a larger window, the system has more number of available packets to send and even in case of out of order delivery, a larger number of packets and acks can be buffered at both ends and once the missing packet in the sequence gets delivered, a large number of packets can be successfully delivered at the application layer for B and similarly the window can move quickly ahead at the sender side A.

## EXPERIMENT 2

### PART -1

The second experiment has 3 parts. For the first part we have the loss probability fixed at 0.2 and the window sizes are varied from 10 to 500. The throughput comparison for the three protocols across different windows is shown below in the following graph:

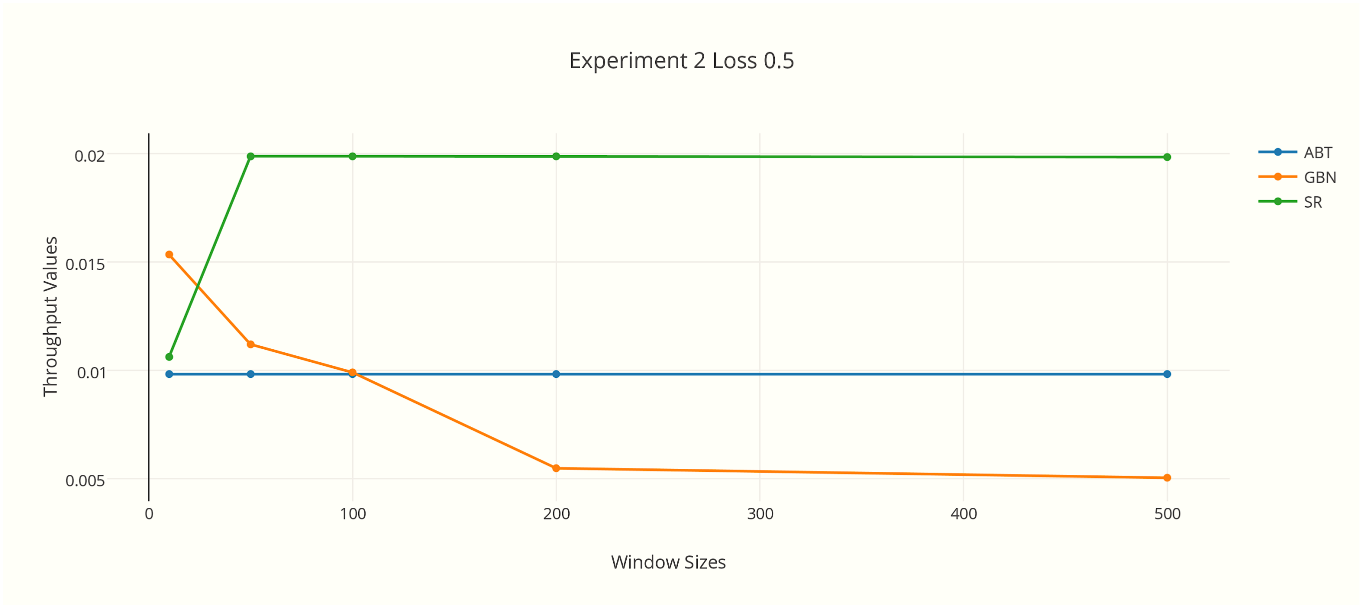


On an overview we can see that the performance for ABT is much lower as compared to GBN and SR. Individually the performance across different window sizes for a loss value of 0.2 seems to be pretty much stable at a given level. For ABT it anyways doesn't depend on the window size so its constant. For SR also the behavior observed from the other data like retransmissions and time taken is seen to be pretty constant which suggests that SR is already performing at its ideal value and doesn't show much variation in performance on increasing window size.

For GBN however, a dip is observed in the middle at the window size 100 but it goes back to its higher performance again for larger window sizes. The dip at 100 can be expected yet the rise in performance afterwards is unexpected. Possible reason can be that the implementation requires optimization maybe by using adaptive timeout values or some other optimizations that can smoothen the curve here. One possible reason can be the outliers in the 10 runs that lower the average overall. Yet if the values are observed closely, we see that the dip in the throughput is not that high.

## PART -2

For the second part we have the loss probability fixed at 0.5 and the window sizes are varied from 10 to 500. The throughput comparison for the three protocols across different windows is shown below in the following graph:



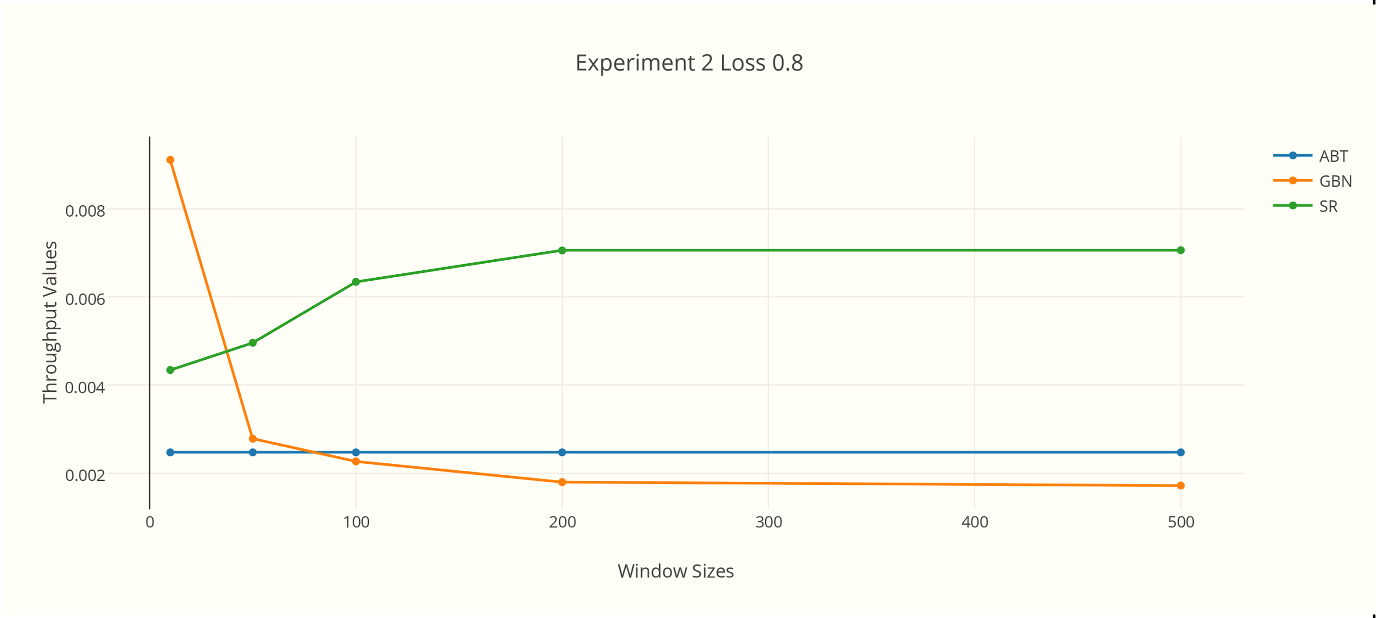
In this case where loss value is 0.5, the behavior of the three protocols is as expected. ABT stays constant.

For GBN, as predicted, the throughput starts to drop at each step up in the window size. As discussed earlier, this is an expected behavior because of the rising number of retransmissions for larger windows for higher loss values.

Similarly for SR we can see that the performance shoots when the window size is increased and then maintains a consistent level, which is its ideal throughput possible for the given parameters.

## PART -3

For the third part we have the loss probability fixed at 0.8 and the window sizes are varied from 10 to 500. The throughput comparison for the three protocols across different windows is shown below in the following graph:



In this case where the loss is as high as 0.8 coupled with a corrupt probability of 0.2, this kind of provides a threshold case of the simulation.

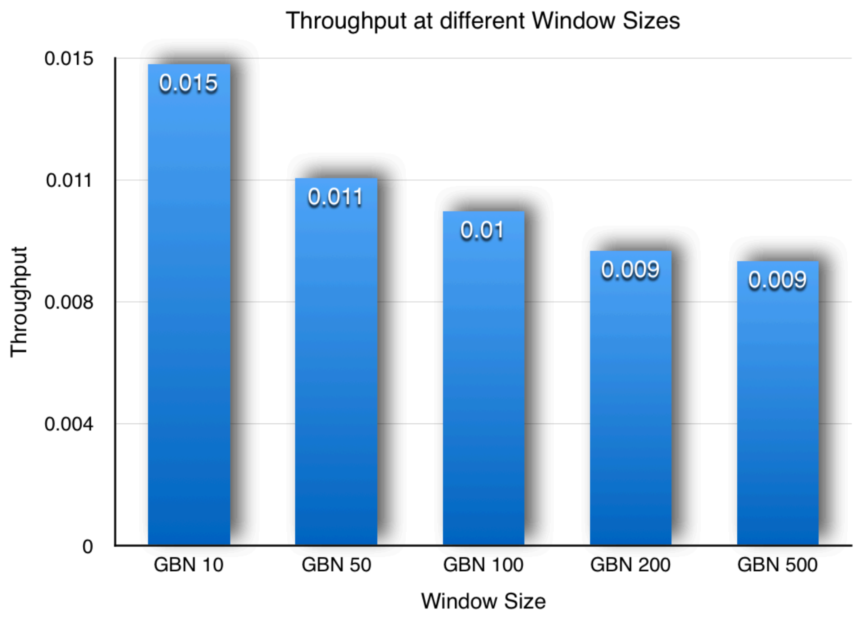
We can see that ABT holds constant.

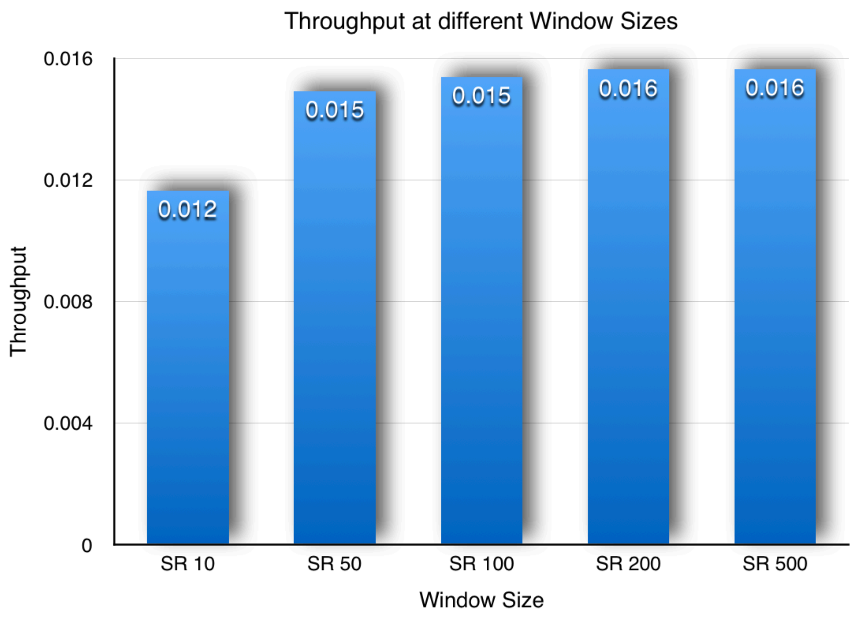
GBN once again, takes a major hit in throughput initially when the window size varies from 10 to 50 and then gradually keeps reducing to its lowermost value. This happens in line with our expectations and the root cause is again the same, higher window size, more retransmissions and higher time.

SR exhibits a normal performance behavior. With increase in window size it gradually starts to give higher throughput values because even for such large loss values, the number of retransmissions and buffers are lower when the window size increases.

## ADDITIONAL ANALYSIS AND CONCLUSION

In addition to the required graphs I wanted to do some additional analysis for the GBN and SR protocols. I constructed the following bar graph for both the protocols where I averaged the throughput values at the three loss values of 0.2, 0.5 and 0.8 and plotted them against the different window sizes.





The first graph represents throughput values for GBN and the second one for SR.

Both the graphs can easily be seen to reinforce the observations discussed in the entire report, which also form the conclusion of the analysis part.

It can easily be concluded that all three protocols have certain parameters under which they are better than the others.

Overall we can say that SR and GBN perform better than the ABT protocol because ABT implements the stop and wait mechanism, which isn't really favorable practically.

Between GBN and SR I would say that SR is a better performer under worse conditions of higher loss values because of lower number of retransmissions.

For GBN we have seen that increase in window size causes its performance to fall, so GBN is favorable only with smaller window sizes.

For SR an increase in the window size improves the performance because of larger buffer available to store out of order packets.

## ACKNOWLEDGEMENTS

1. **To construct the line plots the following link was used,** https://plot.ly/feed/