I have read and understood the course academic integrity policy under this link.
http://www.cse.buffalo.edu/faculty/dimitrio/courses/cse4589_f14/index.html#integrity".

Analysis

For the protocols, a value of 15(15.00) time units has been taken for ABT, while a value of 25 time units has been taken for GBN and SR.
This value has been selected because, for ABT it takes an average of 5 time units for a packet to arrive at the other side when there are no other messages in the medium.
So for a packet to be sent, and a corresponding ACK to be received by the sender, it will take at least 10 time units at the minimum before the sender can expect an ACK.
If there are messages in the medium, it can take even longer.
Hence a value of 15 has been selected as the timeout value, as this value is not too low, such that the sender does not keep on sending duplicate packets thinking that it should have received an ACK by then, and not too large to result in a long delays between sending lost packets when there is packet loss or corruption.
For GBN and SR on the other hand, there would be a whole window of messages present in the medium, due to which the actual time delay could be even longer.
Hence an optimum value of 25 time units has been selected.
Also, experimentally a timer delay of value 25 seems to be optimum for the case of throughput for gbn, as we do not end up sending packets too quickly, for example for the case where my time delay is 10 and loss is 0.8, throughput falls, which may be due to the fact that the window is not moving forward, even though the retransmission rate has gone up.
At the same time we do not want to be sending packets too late (Time delay = 35) in case of loss.

Timer Implementation:
For implementing multiple hardware timers in software I have used a new structure and the following vectors:

```
struct time_data (Defined in header file sr.h)
  {
     int seq_num;
     float time;
     bool acknowledged;
  };
```

This stores the sequence number, time when the timer for this packet will expire and whether this packet has been acknowledged or not.

```
vector<struct pkt> bufferA;
vector<struct pkt> window;
vector<struct time_data> time_windowA;
float TIMER_DELAY 25.00;
```

Now we have a vector of packets, a vector of type time_data and a vector of packets, which we will use as the buffer.

Every time a new message is passed down from layer5, if the window is full, it is put into the buffer vector, else we create a struct of type time_data, put the same sequence number as that of next_seq_num into the seq_num member. We initialize the time member to simulator time + the timer delay (This is the time at which the packet will be expecting the ack message).
When the timer interrupt occurs, we only need to check whether the timeout time of a packet in the window has expired or not by comparing it to the current simulator time, and if it has, and ack has not yet been received, we will resend that packet and update the time member to simulator time + TIMER_DELAY. We check whether an ack has been received by updating the member "**acknowledged**" of the time_data struct.
If an ack number is received before an interrupt for that packet occurs, we will update the acknowledged variable of the time_data struct in the vector and set it to true.
Now we will set the new timer interrupt to be the least timeout value from among the packets, which have not yet been acked in time_windowA minus the current time of the simulator. Which means that after the current time + this difference, there will be a timer interrupt for the packet with the least time. I.e. we set the next timer interrupt to occur for the packet with the least time. We do this every time we receive an ACK, or when a timer interrupt occurs.
Ex: 5 packets received from layer 5,1 time unit after another, Window size 5:
Time_windowA: vector<pkt>

Packet Seqnum:	1	2	3	4	5
Packet Time	1+TD	2+TD	3+TD	4+TD	5+TD
ACK	false	false	false	false	false

TD (Let us consider time delay: 10 here)
Window: vector<pkt>

Packet Seqnum:	1	2	3	4	5
Message:	a	b	c	d	e

After Ack 3 from receiver at time 7:

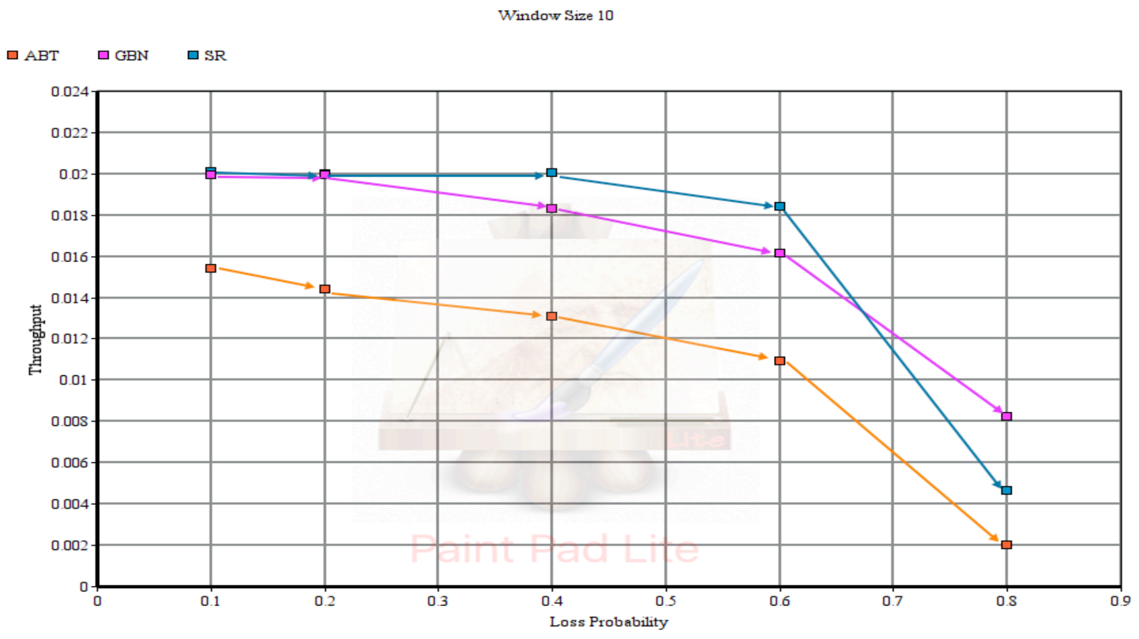
Packet Seqnum:	1	2	3	4	5
Packet Time	11	12	13	14	15
ACK	false	false	true	false	false

At timer interrupt at time 11(1+TD)
Send packet1, which has not yet been Acked and update timer value for this packet (TD+simulator_time (21)).
At timer interrupt at time 12(1+TD)
Send packet2, which has not yet been Acked and update timer value for this packet (TD+simulator_time (22).
Continue the same for packet 4 and 5. Note: **Not packet 3 as ACK received.**
If base seqnum 1 has been received, update the window i.e. Remove 1 from both windows and continue.

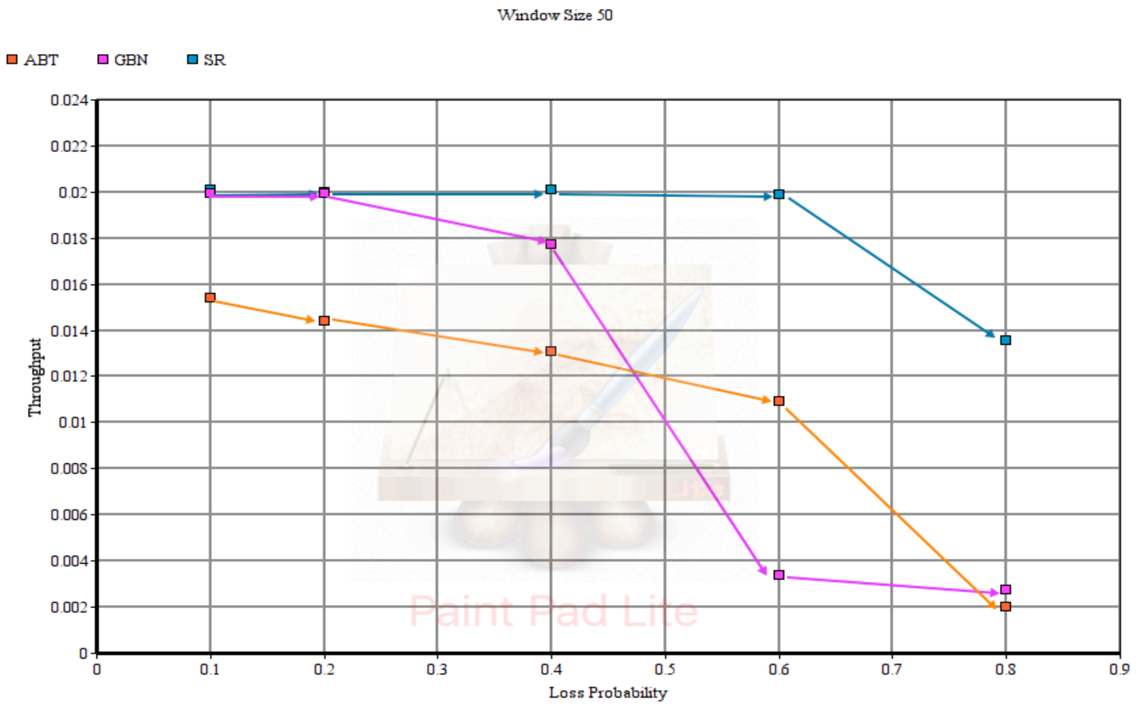
Packet Seqnum:	2	3	4	5
Message:	b	c	d	e

Packet Seqnum:	2	3	4	5
Packet Time	22	13	24	25
ACK	false	true	false	false

Experiment 1:



Graph of Throughput v/s Loss Probability for **Window size 10 above.**

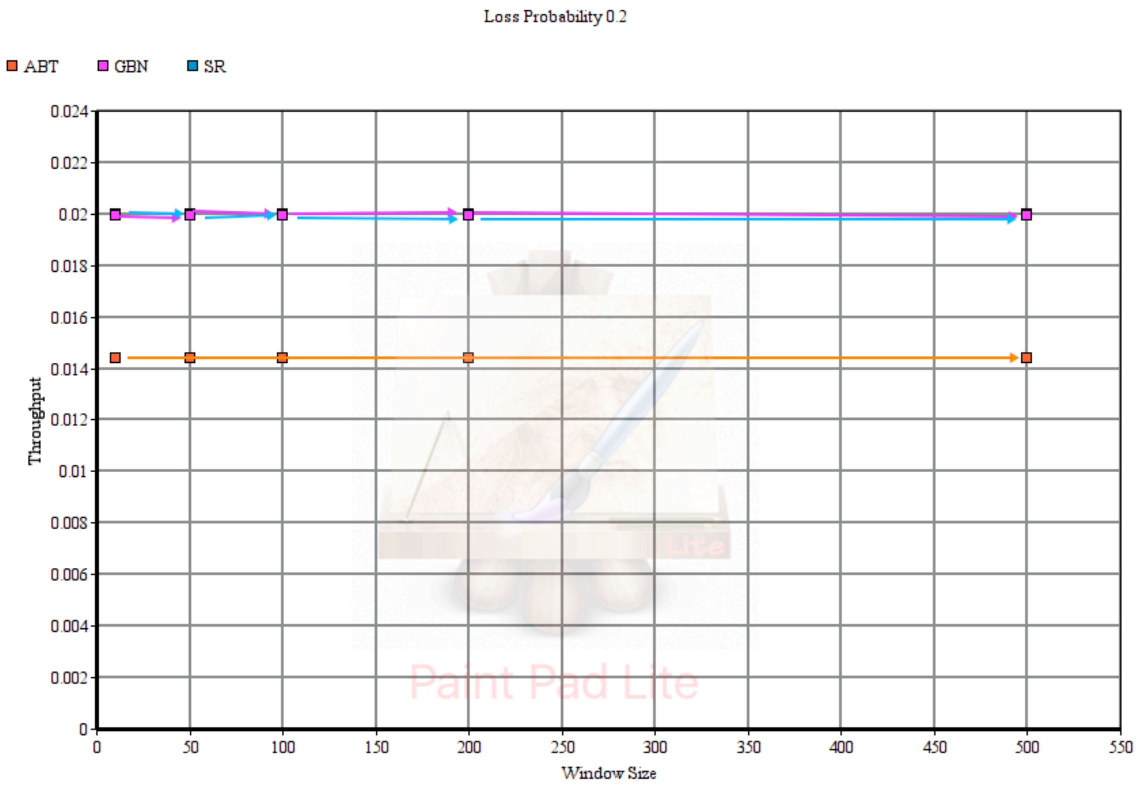


Graph of Throughput v/s Loss Probability for **Window size 50 above.**

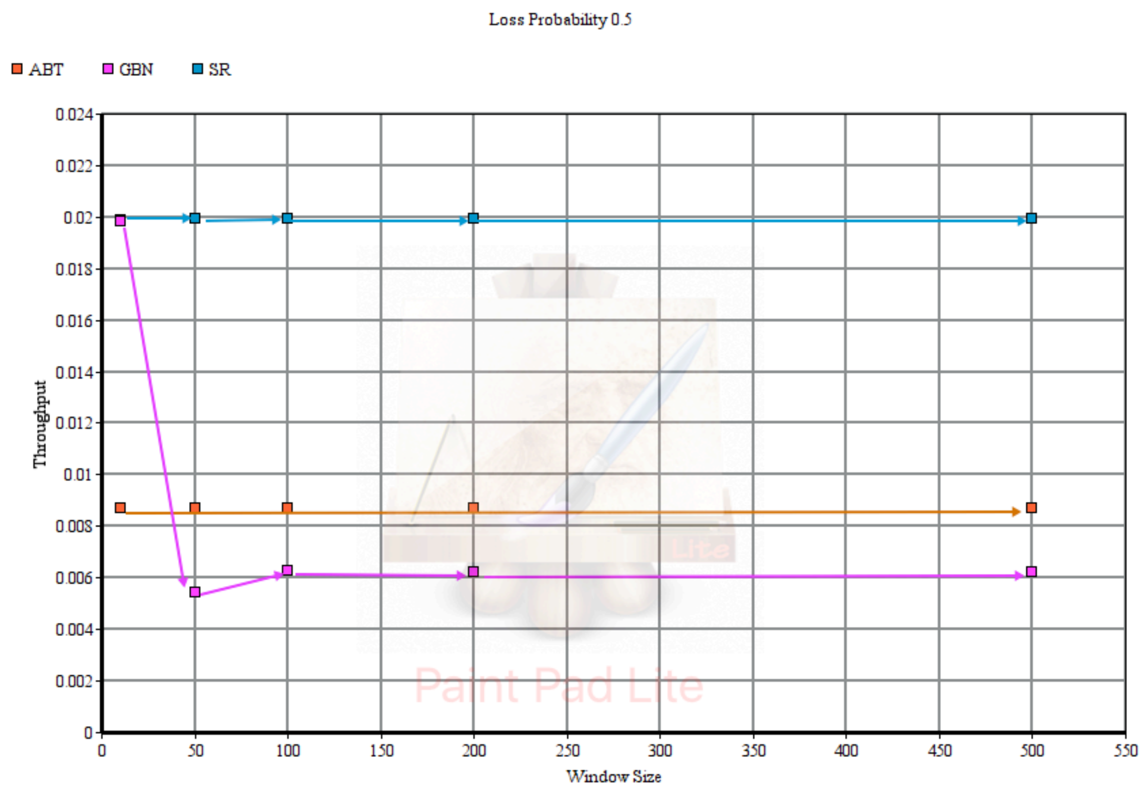
From the above two graphs, it can be clearly seen that GBN performance is poor when the loss probability steadily increases as does the window size.

GBN performance falls drastically when the loss probability increases above 0.4. For window Size 50, even ABT (0.011) has better performance compared to GBN (0.003) for loss probability of 0.6.
GBN offers better performance as compared to SR and ABT (General) for very small window sizes (10) and high loss rates.

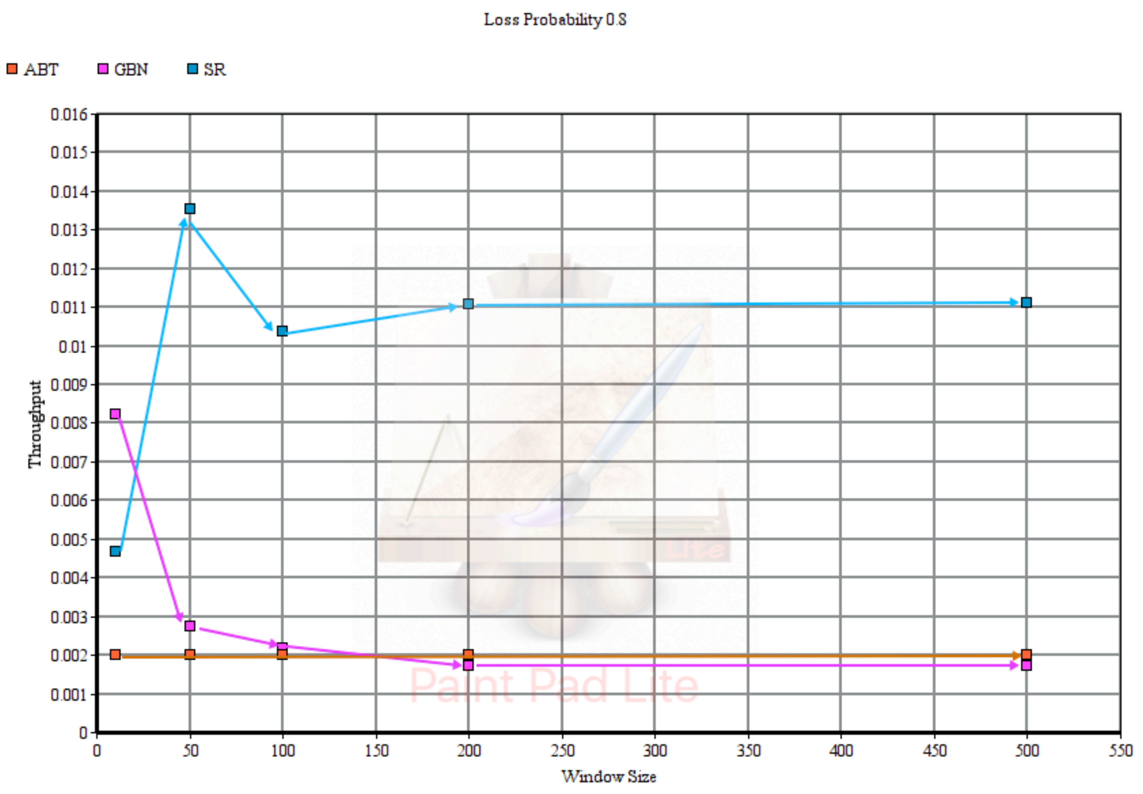
Experiment 2:



Graph of Throughput v/s Window size for **loss Probability 0.2 above.**

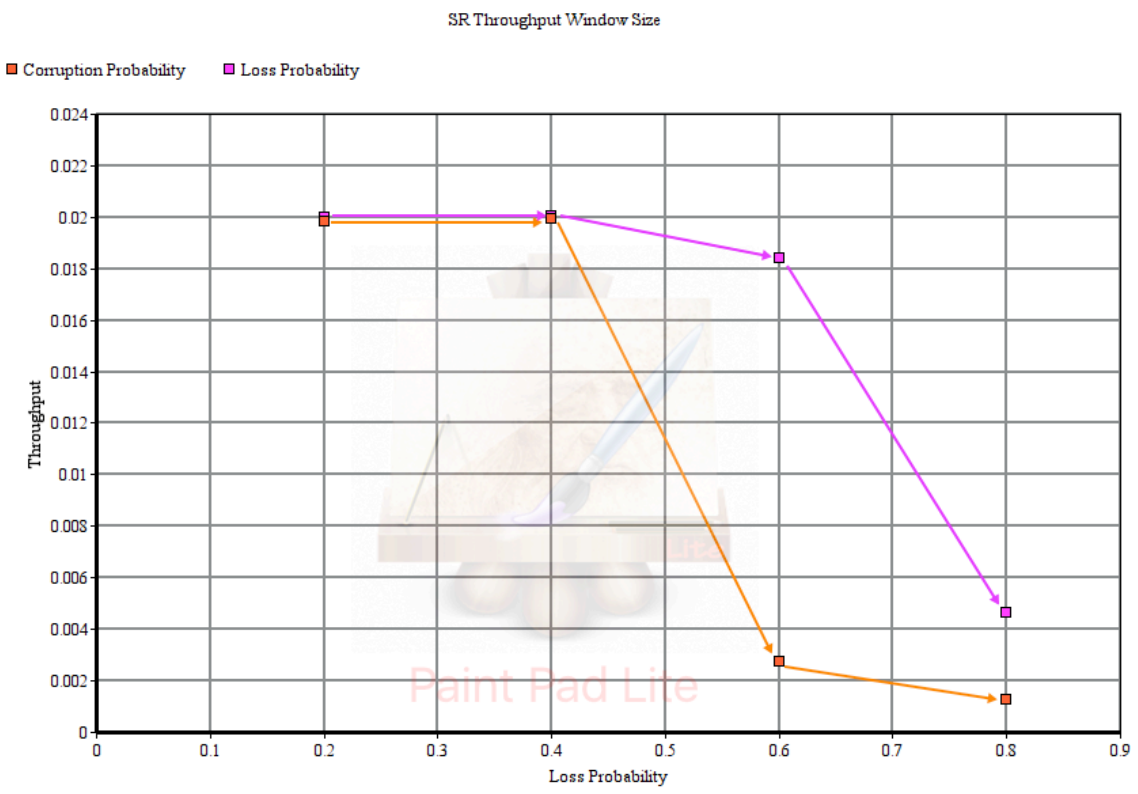


Graph of Throughput v/s Window size for **loss Probability 0.5 above.**
We can see a difference in the throughput of SR and GBN.



Graph of Throughput v/s Window size for **loss Probability 0.8 above.**

For very small window values, GBN has better performance than SR. This will be because, for smaller window sizes, the probability of sending and receiving a base sequence number increases, and as GBN sends the whole small window at a time, the window moves quicker compared to SR for high losses.
When the window gets bigger, GBN is poor for higher loss probabilities. Also the number of messages sent by A transport layer is quite large, while the messages received by B's application layer is quite low. This must be because A is sending a large number of messages and even though they may have been received by the receiver, but since B does not buffer messages received in the window other than the base message, and sends ACK messages only till the last received ACK, the window moves forward very slowly. In fact, ABT has better performance as compared to GBN for high loss rates.
For SR on the other hand, for large window sizes, it only sends those messages whose ACKs have not yet been received. Hence the probability of a window base message being sent and received increases, as the window will send this packet again if it does not receive an ACK and not send those packets whole ACKs have been received. Hence in the case of SR, performance (Considering the messages delivered to Application layer B) is still good



Graph for Throughput v/s Probability for Corruption and Loss **above for Window Size 10**.
For higher levels of corruption, throughput falls more as compared to equivalent packet loss. For higher level of corruption, throughput falls compared to loss throughput. This would be because the receiver does not send back any ACK packets if any packets are corrupted. As a result the window hardly moves forward. In the case of high loss, the receiver would at least send an ACK if a packet did manage to reach it.
All the observations are consistent with what we expected.