ANALYSIS_ASSIGNMENT2

CSE 489/589 – Modern Networking Concepts

PARUSH GARG (50248921)

I, Parush Garg, have read and understood the course academic integrity policy.

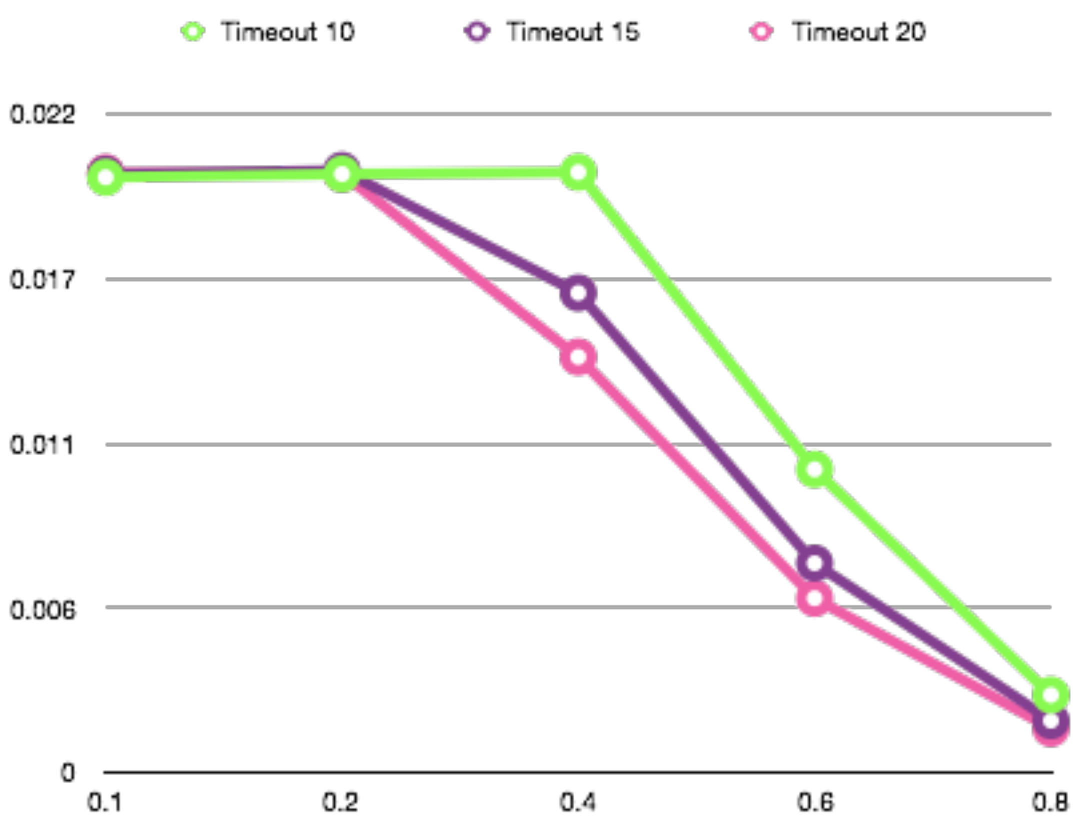
1.   Time out for all the three protocols

- **Alternating Bit Protocol**

This protocol is tested on three values of time out i.e. 10, 15 and 20. I choose constant timeout to be **10** units since the protocol achieves the best throughput as shown in the graph.

ABT (Timeout - 10) [Throughput(y-axis) VS Loss(x-axis)]

Loss	Timeout 10	Timeout 15	Timeout 20
0.1	0.0199092	0.0200185	0.0200663
0.2	0.0200174	0.0201389	0.0200394
0.4	0.0200798	0.0160426	0.0139018
0.6	0.0101442	0.0070065	0.0058354
0.8	0.0025986	0.0017433	0.001484

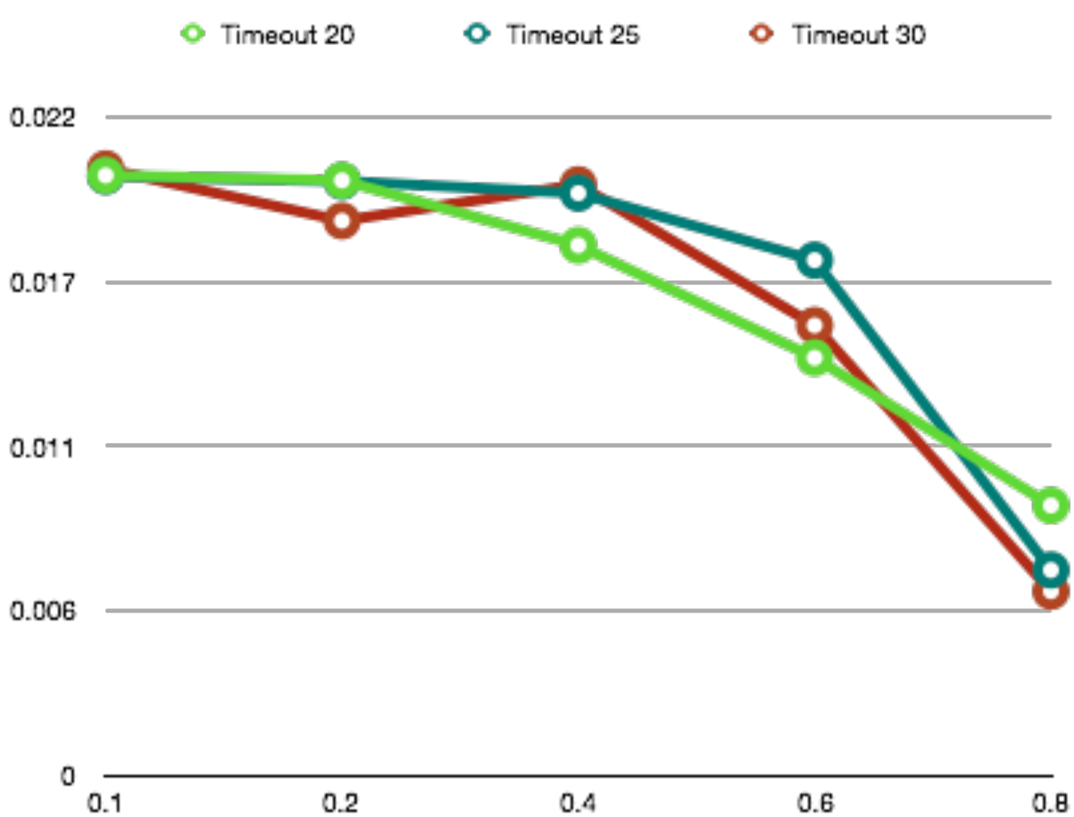


- **Go Back N Protocol**

This protocol is also tested on three different values for timeout i.e. 20, 25 and 30. I choose the value to be **25** since in the graph it shows the optimal behavior beating all the other values till the loss probability close to 0.5.

GBN (Timeout - 10) [Throughput(y-axis) VS Loss(x-axis)]

Loss	Timeout 20	Timeout 25	Timeout 30
0.1	0.0200482	0.0200236	0.0202686
0.2	0.0198799	0.0198561	0.0185297
0.4	0.0177136	0.0194559	0.0197379
0.6	0.0139553	0.017217	0.0150366
0.8	0.0090214	0.006864	0.0061559

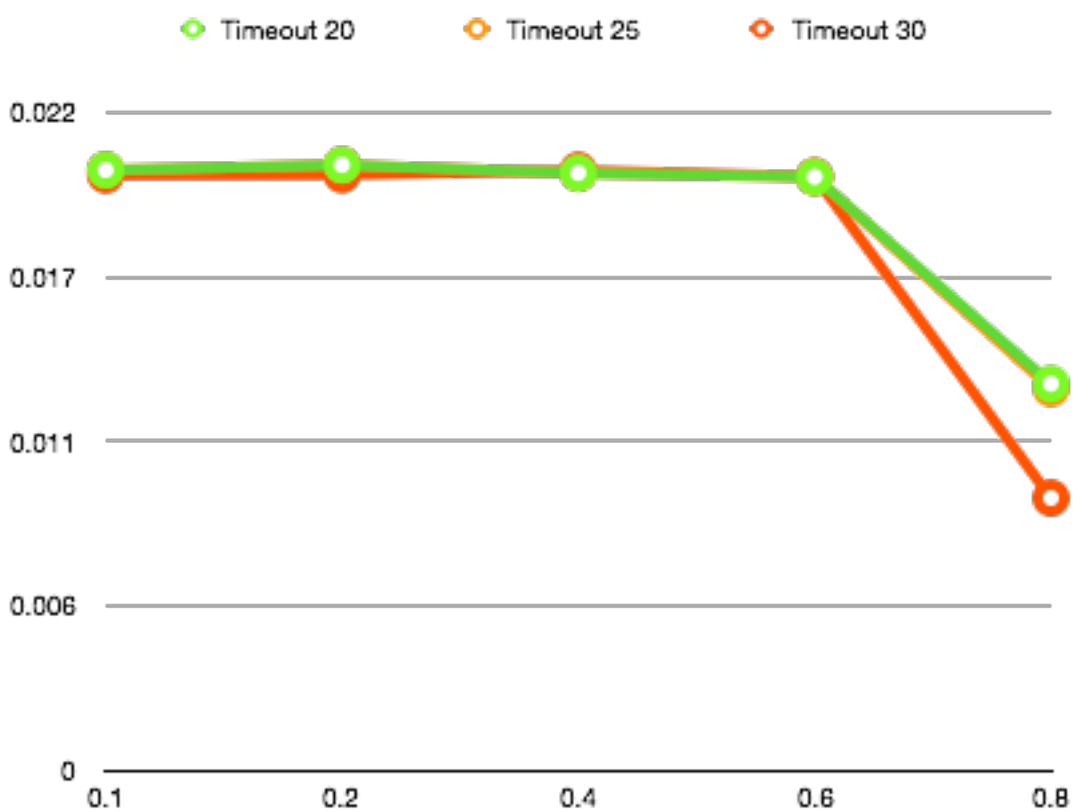


- **Selective Repeat Protocol**

This protocol is also tested on three different values for timeout i.e. 20, 25 and 30. I choose the value to be **20** since in the graph it shows the best behavior achieving the highest throughput with respect to other two values.

SR (Timeout - 10) [Throughput(y-axis) VS Loss(x-axis)]

Loss	Timeout 20	Timeout 25	Timeout 30
0.1	0.0200743	0.0200743	0.0199099
0.2	0.0202426	0.0202426	0.0199199
0.4	0.0199742	0.0199742	0.0200617
0.6	0.0198461	0.0198461	0.0198584
0.8	0.0129292	0.012776	0.009111



2. Implementation of multiple software timers in SR using a single hardware timer

Variables from the code:

cur_time = starttime of the packet.
Firstminimum = the first minimum time of unACKed packet in the vector for which the timeout happened.
Secondminimum = the second minimum time of unACKed packet in the vector with respect to which next timeout needs to be set.
Packetnumber_first = Sequence number of the first minimum cur_time packet in the vector.
Packetnumber_second = Sequence number of the second minimum cur_time packet in the vector.
Sendbase = base value in the sender window.
Packet_time_vector = vector which stores information of packets and their respective timers.
Nextseqnum = index number till which the vector has been filled in sender side.

In SR, I have used the following logic:

: Interrupt happens
: Initialize Packetnumber_first and Packetnumber_second with sendbase.
: Initialize **firstminimum** and **secondminimum** with timer of packet stored at sendbase index in packet_time_vector vector.
: Iterate over vector containing **cur_time** and packet information, check for an **unACKed packet**.
: Find the **first minimum and second minimum times** from all the unACKed packets. Store their sequence number in variables defined above.
: If first minimum and second minimum are same which means there is only one unACKed packet, send the packet and **start the timer with original timeout**.
: Else, send the packet of sequence number matching with first minimum timer value, set it's cur_time to be get_sim_time(), send the packet and start the timer with **value = (second minimum timer – (get_sim_time() – original timeout))**.

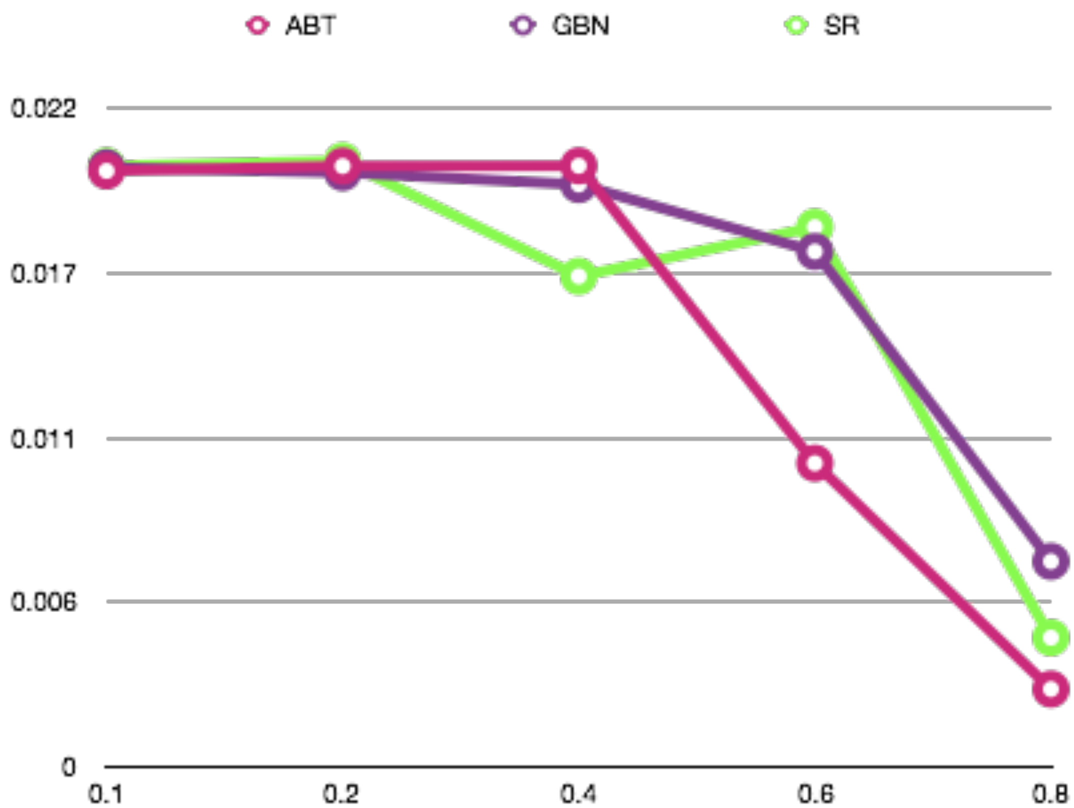
```
void A_timerinterrupt()
{
    float time_of_packet;
    int packetnumber_first;
    int packetnumber_second;
    float firstmin, secondmin;
    packetnumber_first = packetnumber_second = sendbase;
    firstmin = secondmin = packet_time_vector.at(sendbase)->cur_time;

    for (int i = sendbase; i < sendbase + senderwindow && i < nextseqnum; i++)
    {
        if(packet_time_vector.at(i)->packet->acknum == -1)
        {
            /* If current element is smaller than first
            then update both first and second */
            if (packet_time_vector.at(i)->cur_time < firstmin)
            {
                secondmin = firstmin;
                firstmin = packet_time_vector.at(i)->cur_time;
                packetnumber_first = packet_time_vector.at(i)->packet->seqnum;
            }
            /* If packet_time_vector.at(i) is in between first and second
            then update second  */
            else if (packet_time_vector.at(i)->cur_time < secondmin && packet_time_vector.at(i)->cur_time != firstmin)
            {
                secondmin = packet_time_vector.at(i)->cur_time;
                packetnumber_second = packet_time_vector.at(i)->packet->seqnum;
            }
        }
    }

    //This means only single packet in vector with unACKed status. Send it!
    if(secondmin == firstmin)
        starttimer(0, delay_RTT); //Starting timer for base packet
    //
    else
        starttimer(0, packet_time_vector.at(packetnumber_second)->cur_time - (get_sim_time() - delay_RTT)); //Starting timer
    //cout<<"Sending below packet"<<endl;
    packet_time_vector.at(packetnumber_first)->cur_time = get_sim_time();
    tolayer3(0,*(packet_time_vector.at(packetnumber_first)->packet));
}
```

3. ***Experiment 1***

- **Window size: 10 for all protocols**

Window size 10 [Throughput(y-axis) VS Loss(x-axis)]

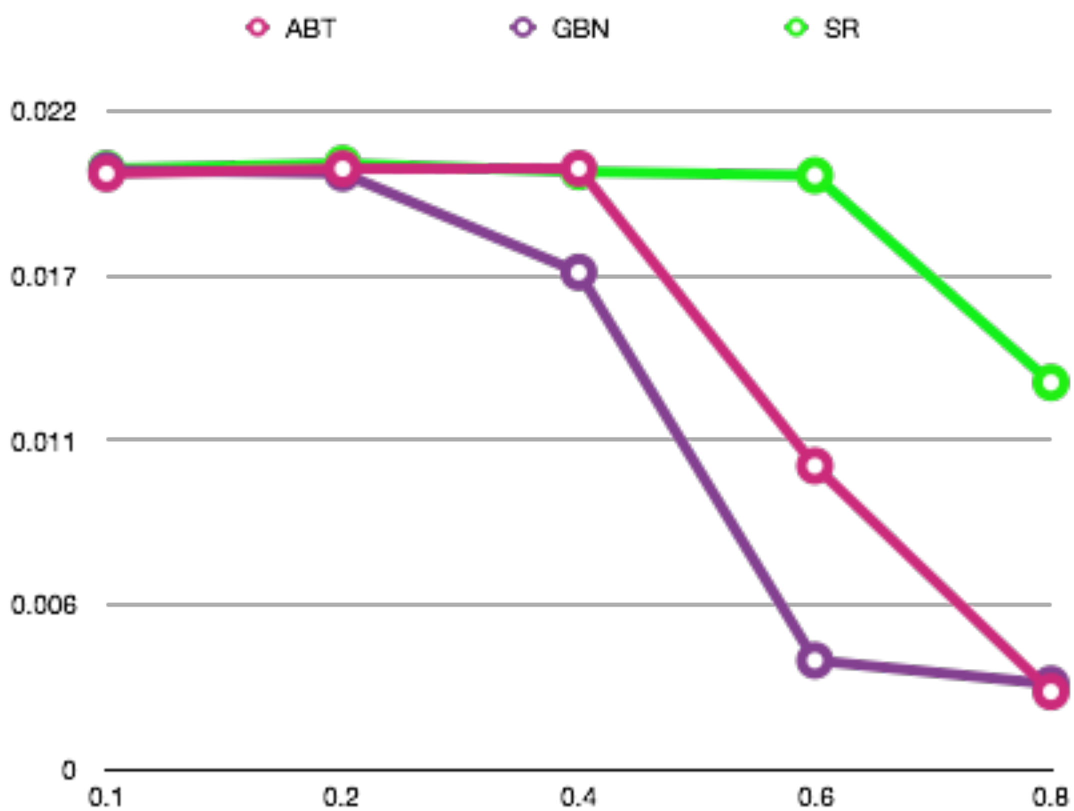
Loss\Protocol	ABT	GBN	SR
0.1	0.0199092	0.0200236	0.0200743
0.2	0.0200714	0.0198561	0.0202445
0.4	0.0200798	0.0194559	0.0163869
0.6	0.0101442	0.017217	0.0180321
0.8	0.0025986	0.006864	0.0043153



- **Window size: 50 for all protocols**

Window size 50 [Throughput(y-axis) VS Loss(x-axis)]

Loss\Protocol	ABT	GBN	SR
0.1	0.0199092	0.0200236	0.0200743
0.2	0.0200714	0.0199212	0.0202426
0.4	0.0200798	0.0166104	0.0199742
0.6	0.0101442	0.00364409	0.0198461
0.8	0.0025986	0.0028564	0.0129292



For **ABT**, as the probability value of loss increases, meaning more number of packets are being lost, throughput decreases for all the protocols. **Throughput is inversely proportional to loss**. This makes sense as it will ignore the packets if sender has not got ACK for previous packets. Moreover, ABT does not buffer packets as well. Window size does not matter as it is always 1 in ABT. Therefore, graph is similar for ABT in both the window sizes.

For **GBN**, if one packet loss happens, it sends all the packets again in the current window. **GBN will work good for small window sizes.** As the window size and loss increases, the throughput is not stable and decreases since if a loss occurs, then all packets in the window will be sent once again. Therefore, in graph for GBN, for both window sizes, throughput is same till loss = 0.2, but it decreases more for window size 50 than window size 10 as we increase the loss probability.

For **SR**, it only retransmits the selected packet which is lost or it's ACK is not received. Hence it is known as selective repeat. In graph, throughput is kind of similar for both window sizes except at loss = 0.4 and 0.8.
At loss = 0.4, I expected the throughput to be close to 0.019 but got 0.0163. However, when I checked the results in results.csv file, I see that majority of the packets received at B side are approximately 998. Due to particular seed, the result is 497, average throughput goes low.

Run	Messages	Loss	Corruption	Time_bw_messages	Application_A	Transport_A	Transport_B	Application_B	Total_time	0.0202445
1	1000	0.4	0.2	50	1000	4508	2694	997	50427.949219	0.019771
2	1000	0.4	0.2	50	1000	9855	4835	725	51066.28125	0.014197
3	1000	0.4	0.2	50	1000	4501	2677	995	49427.964844	0.02013
4	1000	0.4	0.2	50	1000	13859	7805	539	48973.457031	0.011006
5	1000	0.4	0.2	50	1000	4423	2640	999	50746.328125	0.019686
6	1000	0.4	0.2	50	1000	13005	7857	910	51291.347656	0.017742
7	1000	0.4	0.2	50	1000	4512	2674	998	48988.578125	0.020372
8	1000	0.4	0.2	50	1000	11030	5982	543	50514.09375	0.010749
9	1000	0.4	0.2	50	1000	4563	2762	999	50238.460938	0.019885
10	1000	0.4	0.2	50	1000	11186	6458	497	48107.539062	0.010331

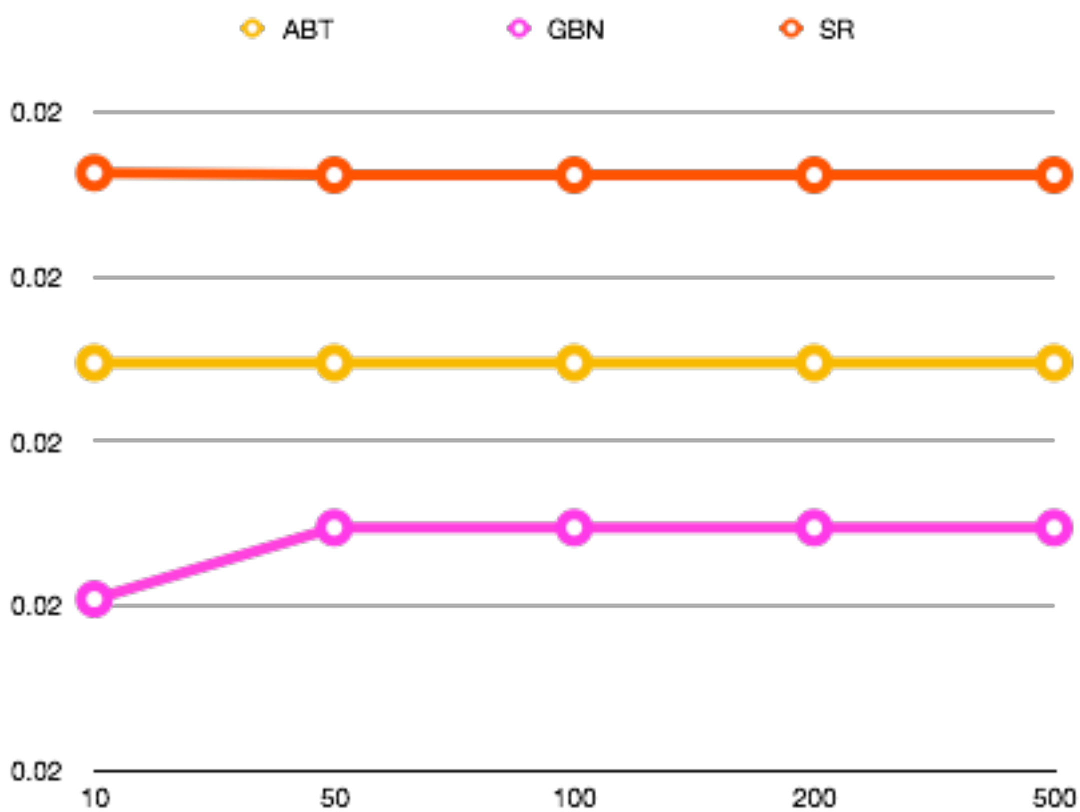
At loss = 0.8, there is a slight increase in throughput of window size 50 which is expected as well. Otherwise, things are similar.

4. Experiment 2

- **Loss – 0.2**

Loss = 0.2 [Throughput (y-axis) VS Window size (x-axis)]

Window Size\Protocols	ABT	GBN	SR
10	0.0200714	0.0198561	0.0202445
50	0.0200714	0.0199212	0.0202426
100	0.0200714	0.0199212	0.0202426
200	0.0200714	0.0199212	0.0202426
500	0.0200714	0.0199212	0.0202426

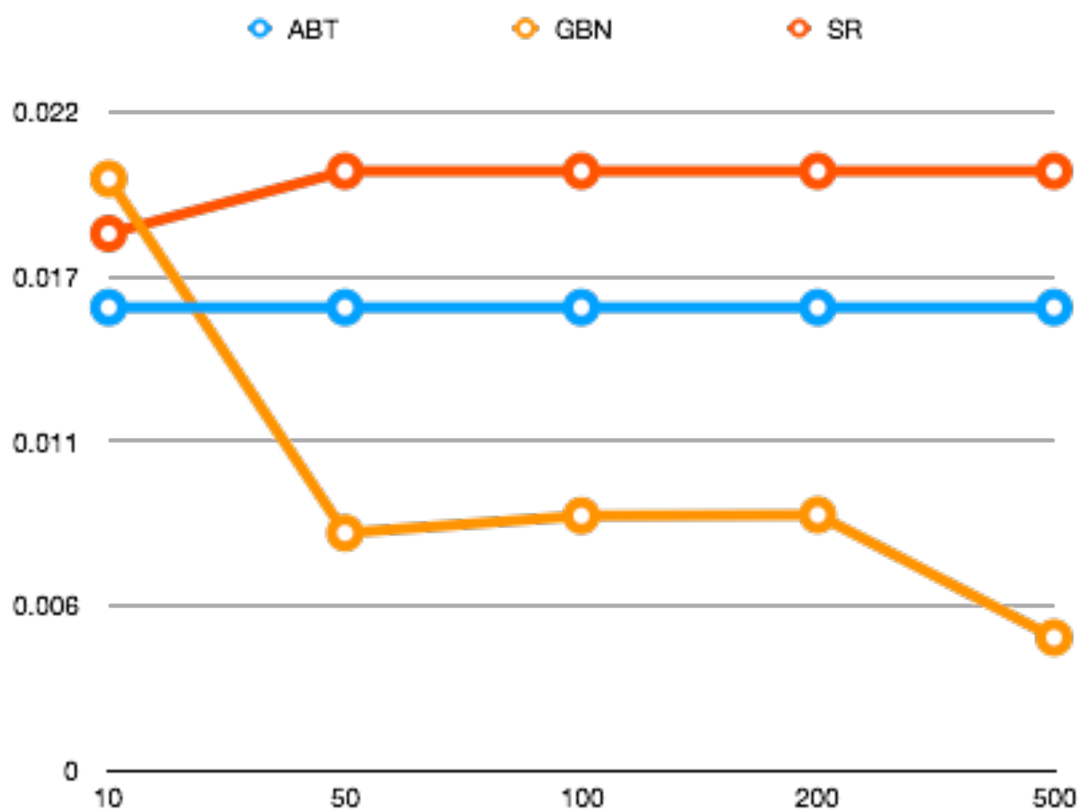


In this case:
ABT has same throughput since window size does not matter.
GBN has close to same throughput as loss probability is very low.
SR has same throughput as loss probability is very low.
Almost all the three protocols achieved close to same throughput.

- **Loss – 0.5**

Loss = 0.5 [Throughput (y-axis) VS Window size (x-axis)]

Window Size\Protocols	ABT	GBN	SR
10	0.0154697	0.0197655	0.0179378
50	0.0154697	0.0079359	0.0200242
100	0.0154697	0.0085116	0.0200242
200	0.0154697	0.0085389	0.0200242
500	0.0154697	0.00444	0.0200242



In this case:

ABT has same throughput in all the window sizes but lower than what we achieved from loss = 0.2. Since packets are getting lost more now hence we attained a decreased throughput value.

GBN has a decreasing throughput as we increase the window sizes. As window size increases and

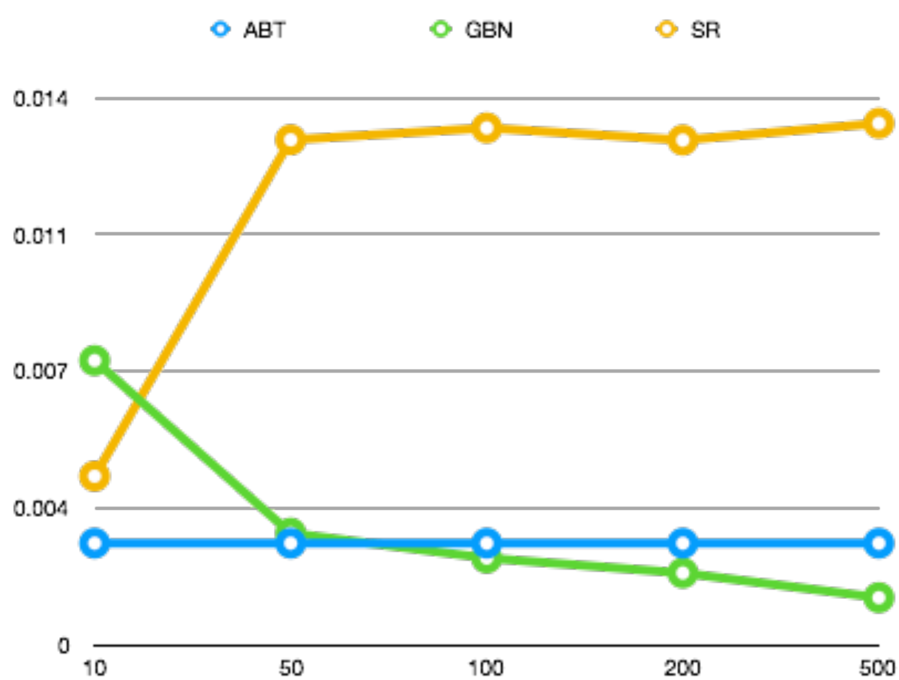
half of the total packets are getting lost which means it will try to send all packets in the window again and again for which ACK was lost and timeout happened since it is not selective. Therefore, throughput becomes very low as we approach window sizes > 100.
I increased timeout values later in my experiments as the window size increases. That is, I used timeout value to be 35 units with window size 200 and achieved a better throughput than received in the above case. We can change timeout value like if window size is high then, timeout value should also be at a higher level. However, in all the tests, I have used a constant timeout value which is 25 units.

SR is the leader here among all the protocols as we get the best throughput. It is doing so as it sends the selective packet once it does not receive ACK of the expected packet. Therefore, even if we increase the window sizes, results on throughput should not be effected drastically like it happened in the case of GBN.

- **Loss – 0.8**

Loss = 0.8 [Throughput (y-axis) VS Window size (x-axis)]

Window Size\Protocols	ABT	GBN	SR
10	0.0025986	0.007277	0.0043153
50	0.0025986	0.0028564	0.0129292
100	0.0025986	0.0022183	0.0132336
200	0.0025986	0.001836	0.0129213
500	0.0025986	0.0012045	0.01335222



In this case:

ABT gets the minimum throughput of all the loss probabilities taken into picture since probability of loss is very high hence throughput is decreased. This happens as it only sends one packet at a time and wait for the last sent packet ACK. If it is lost which is the picture here, it will keep on waiting at that state and come out of it only after getting a non corrupted expected sequence ACK.

GBN performs well than ABT here till window size is lesser than 100. Hence GBN leads ABT in the case of loss is high. At window sizes close to 200 and 500, throughput is very less since it will transmit all the packets in window when packet is lost which results in continuous loop running.

SR is the leader here as for more loss value and more window size, it achieves the best throughput. Hence, the higher window size gives a better performance.