CSE 589 PROJECT TWO-REPORT

Anand Tiwary

UNIVERSITY AT BUFFALO

Table of Contents

Introduction

This document contains a performance analysis of two Transport layer protocols: Go-Back-N and Selective Repeat. The below two experiments were performed on each of the two protocols and I have tried to provide a good explanation in later part of the document. A total of 1000 messages were sent with a corruption probability of 0.2 and message arrival interval set as 50. Rest of the details are as follows.

1. **Experiment 1:**

   In this experiment, I have analyzed the performance of each of the three protocols with varying loss probabilities- [0.1, 0.2, 0.4, 0.6, and 0.8]. The readings were taken for two window sizes [10 and 50].
2. **Experiment 2:**

   This experiment measures the effect of changing window size [10, 50, 100, 200, and 500] at different loss probabilities [0.2, 0.5, and 0.8] on the throughput.

Timeout Estimation: Since timeout is an important factor in protocol performance, I have implemented an adaptive timeout estimation. This gave good performance in some situation. However, there were cases when keeping a static timeout gave better performance. Similar performance variations were observed when different timeout estimation function was used. The details are included in the corresponding Protocol section.

Alternating Bit Protocol

In this protocol, the sender maintains only two sequence numbers for the packets (0 and 1). It sends the packet with one of the two sequence number and then waits till it is acknowledged and then sends the next packet with the other sequence number. While it is waiting for the acknowledgement, it does not send any more packets and hence the progress is very slow. It also maintains a timeout to deal with packet loss situation and retransmits the previously sent packet. The receiver just waits for a particular sequence number. Once it received the correct and non-corrupted packet, it sends an acknowledgement back. If it receives a packet with un-expected sequence number then it just sends back the last acknowledgement.

- **Implementation**:

  I have used a variable ‘**a_state’** to keep track of the state of the sender. It only sends the packet when it is in valid ‘sending’ state. Similarly, it receives the acknowledgment in the corresponding state. This makes sure that the state transition is proper and ensures the correctness of the protocol. Similarly, for receiver, another state variable ‘**b_state**’ is used to ensure its correct working according to this protocol.
- **Timeout**:

  Since this protocol sends only one packet at a time, network congestion due to multiple retransmission and high loss and corruption will not be a problem. So, there is no need of an adaptive timeout here. I have kept a static timeout of “20” for this protocol.
- **Analysis:**

  This protocol provided a very poor performance. Most of the time, sender and receiver were sitting idle for acknowledgement, next packets to arrive respectively. The throughput obtained was very less. There is no need for congestion control or flow control, because such a situation is completely avoided due to the Stop and wait nature. This wastes a lot of bandwidth and is definitely not a practical protocol for network communication.

Go Back N

In this protocol, the sender maintains a buffer for packets. A window is maintained, which is limits the maximum number unacknowledged packets at a given time. The protocol uses cumulative acknowledgment and in case of a timeout it retransmits all the packets from the base till the last packet inside the window. The receiver, on the other hand, works mostly like Alternating bit protocol. It just waits for the next expected packet and sends an acknowledgement. It discards out of order packets and just returns the previous acknowledgement for this and for any other old packet.

- Timeout:

  Since this protocol retransmits all the unacknowledged packets on timeout, choosing a good timeout value becomes an important performance measure. This is more important in cases where the loss probabilities and corruption probabilities are high.

  I have used an adaptive timeout because keeping a static timeout will not give consistent results for different combination of loss, corruption and window size.

  Choosing the correct estimation technique is important because it may result in performance improvement in one case while a decrease in some other.

  I tried with different estimation approach, but could not come up with an approach that gave very good results. Among the approaches I tried, the following one (Inspired by TCP estimation) gave overall better performance for all the different scenarios.

  1. Keep an initial static timeout 0f 15 units (a_static_timeout).
  2. Calculate the timeout :
     estimatedRTT=(1-a)*estimatedRTT+a*sampleRTT;
     devRTT=(1-b)*devRTT+ b*abs(sampleRTT-estimatedRTT);
     Timeout= 0.6* Timeout + 0.4*(estimatedRTT + 5*devRTT);
     We have used a weighted sum to smooth out the changes in timeout interval.
  3. Set a high timeout when the number of outstanding packets (original, retransmitted and ack packets) in the network become very large. Then re-estimate. This is used to avoid network congestion due to retransmission (I believe congestion may affect the RTT of packets).
     timeout=10*a_static_timeout;
  4. For timeout estimation, I am using only the packets which were not retransmitted. I am also discarding packets which were acknowledged using Cumulative acknowledgment. Only those packets RTT is considered for which acknowledgment is actually received.

I believe, for Go-Back-N timeout should also be related to the **window size** (Since all the packets gets retransmitted after a timeout and thus increasing the congestion). I tried few combination, but the results were poor.

Selective Repeat

In this protocol, both the sender and receiver maintains a buffer. Each packet is acknowledged separately. So a software timer is kept for each packet. When the base packet is acknowledged, the window gets shifted to the right by the number of consecutive acknowledged packets. The received works in a similar way.

Two important design challenges with this protocol:

- **Software Timer**:

  To implement multiple software timers using only one available timer routine, we used the following approach.

  1. Using time_local, calculate at what time the packet should be timed out. We stored this value with each packet as time_offset. It is calculated using below.
     Time_offset=time_local+ get estimated timeout interval
  2. Initially a static value is used to trigger the A_timerinterrupt routine.
  3. When A_timerinterrupt is triggered, it scans through each unacknowledged packet in the buffer to find a packet whose (time_offset – time_local) is within the range of [-1,1].
  4. If there is such a packet, then it retransmits it after setting the new value for time_offset as specified in step 1.
  5. It then, finds out which is the packet with the least value for (time_offset – time_local). It then uses this value as the next timeout interval and starts the timer again.
  6. If there is no more packets in the buffer right now, then A_timerinterrupt simply sets the default timeout interval and starts the timer again.

Using this approach, the interrupt routine knows exactly when to interrupt again. This avoids frequent checks for individual packet timeouts and lets the system give more time to packets sending part.

- **Timeout:**

  For timeout, we are using a TCP like estimation technique. This is similar to the one used for Go-Back-N with slight modification. The choice of estimation function has a high importance. Different estimation technique may result performance increase in one scenario while a decrease in another**. For this project, my focus was to improve the throughput in case of high loss while still give reasonable performance for other scenarios.** The below approach is used keeping this in mind.

  Below steps describe the timeout estimation logic.

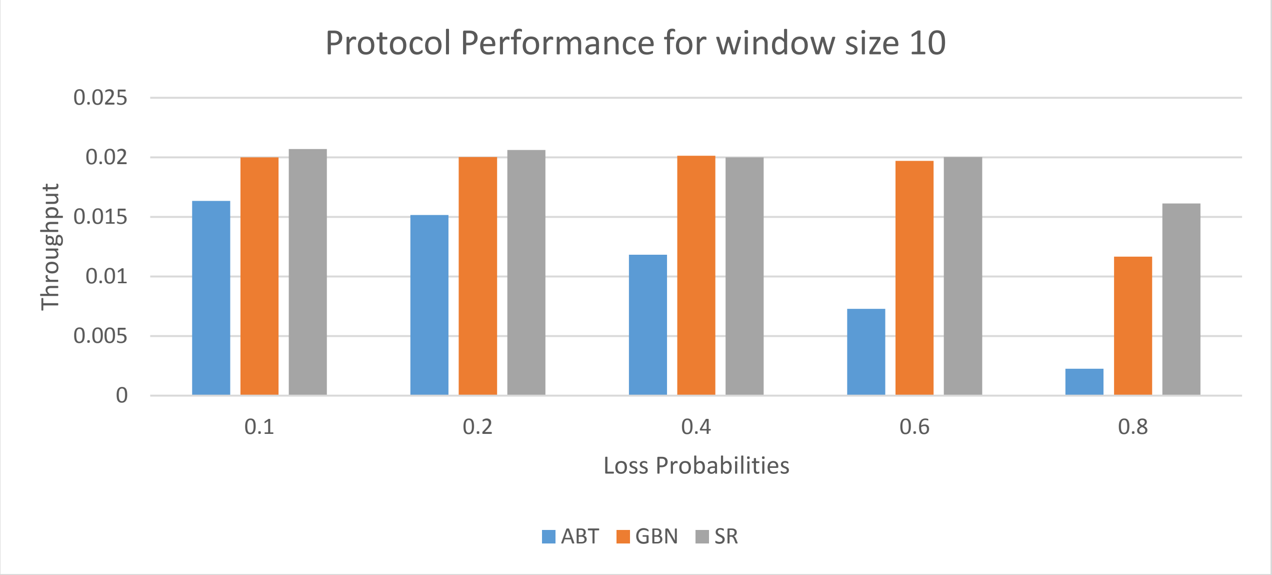
  1. Keep an initial static timeout of 8 units.
  2. Estimate timeout, every time a new packet is acknowledged:
     estimatedRTT=(1-a)*estimatedRTT+a*sampleRTT;
     devRTT=(1-b)*devRTT+ b*abs(sampleRTT-estimatedRTT);
     timeout=0.6*timeout + 0.4*(estimatedRTT + 5*devRTT);
  3. A weighted sum to smooth out the changes in timeout interval.

4. In case of retransmission of base packet, ‘timeout’ is set to static timeout and estimation is done again. After this, the new timeout is used to set the timer interrupt for base packet. This is done only for base packet.
5. For estimation, only non-retransmitted packets are considered.

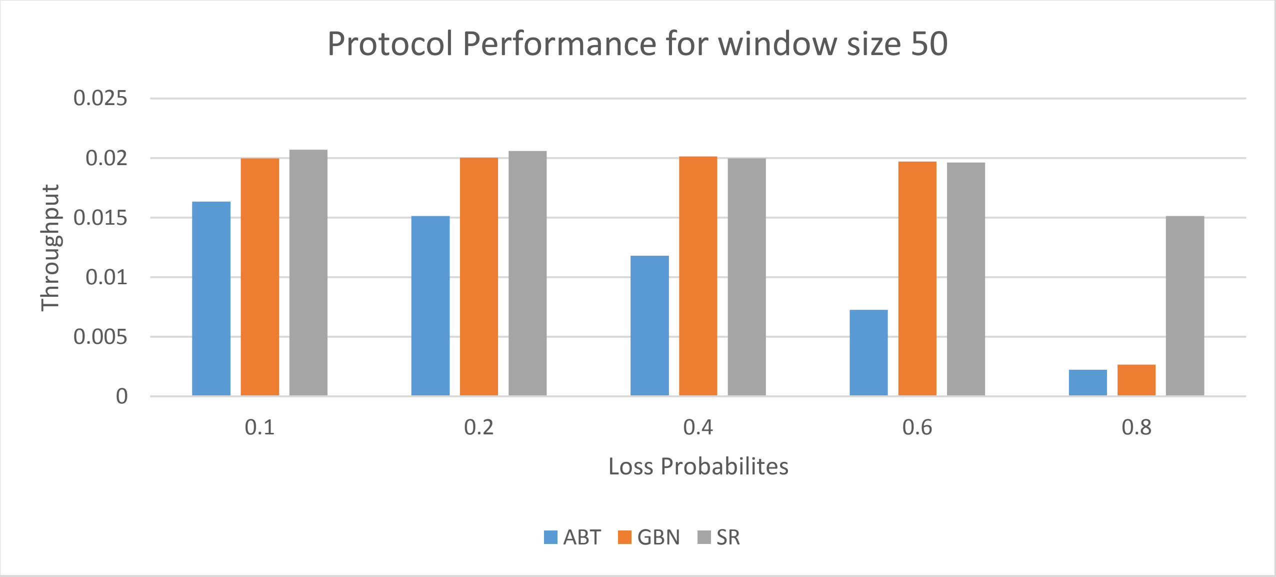
Experiment 1

Here, variation in throughput of protocols are studied for different loss probabilities. The readings are taken for two window size 10 and 50.

Window Size 10:



Window Size 50:



Observations and Analysis:

1. **ABT has the worst performance among all the three protocols.** This is because, in ABT only one packet is sent at a time. So once a packet is sent, the sender stays idle for 1 RTT. This decreases the throughput to a great extent.
2. **Performance of ABT decreases as loss probabilities decreases.** As the loss probability is increased, more and more packets will be dropped. The packet will eventually timeout and get retransmitted. The sender will stay idle (or no progress) until it receives a valid acknowledgement. This idle period is the cause of decreasing throughput.
3. **Performance of GBN is almost same as SR except for loss 0.8.** In GBN, window will move to the right when acknowledgment of any of the transmitted packet is received. This advantage is due to the **cumulative acknowledgement** feature. Because of this, in the presence of loss, the probability that one of the acknowledgment (for N transmitted packets, where N=window size) will reach the sender is more. Hence, the chances of shifting window to the right and sending new packets is more. This is why, it performs better for low to moderate loss rate.

   However, if timeout is set too low or loss probability is very high (i.e. >0.8), the end result will be a lot of retransmission which increases the network congestion. Now, window size becomes an important criteria. If loss is high, and window size is also high (>=50) then a single packet error (loss or corruption) will retransmit all the packets inside the window. This eats up a lot of network bandwidth and is the reason for very bad performance. We can observe this in the above graph for loss 0.8. There is a large decrease in throughput from the one at loss=0.6 to loss =0.8. This decrease is more for window size=50 than that for window size =10, because of the reasons explained above.
4. **SR has the best performance among the three protocols.** SR performs better because of packet buffering in the receiver side and the ability of acknowledging individual packets. Since, every packet has its own timer, on a timeout only one particular packet gets retransmitted. This keeps the congestion to a minimum. Since, packets gets buffered at the receiver, a correctly received packet is not discarded.

However, if the base packet or it's acknowledge keep getting lost or corrupted, then there will be no or little progress. Because it will not move the window. So even if all the other packets are acknowledged, the window will not shift because base packet is not acknowledged. This is why, when loss is high, the performance of SR is decreased a bit.
Timeout is also an important factor here. If it is set to a small value, then it will result in retransmission and decreasing the performance.

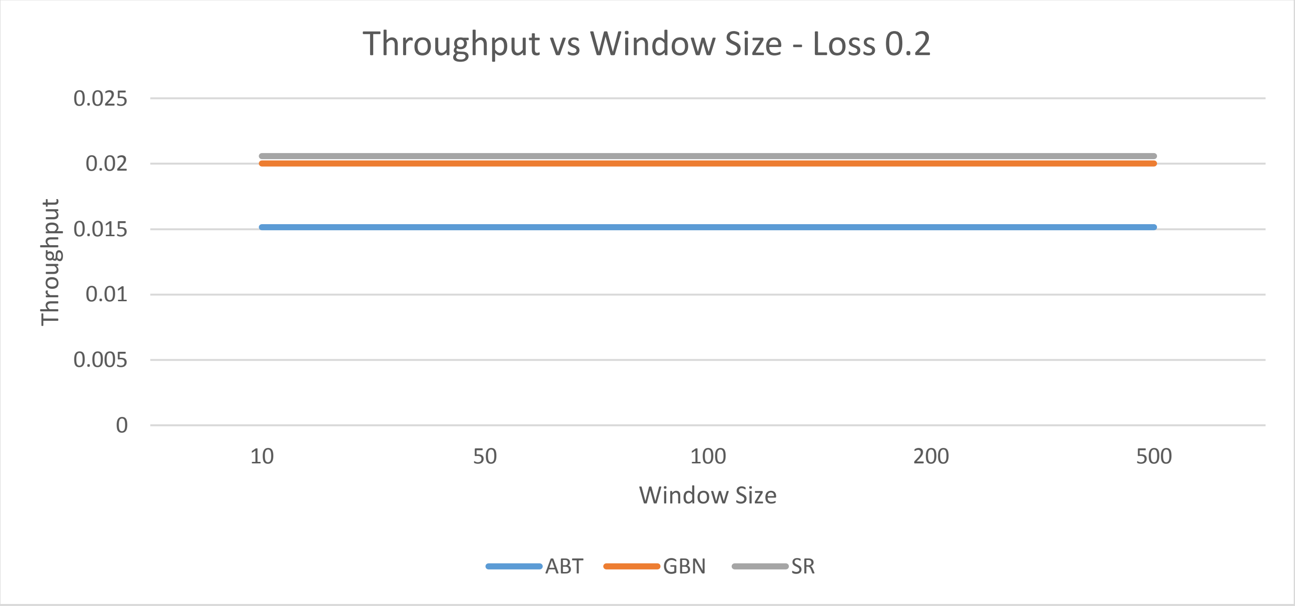
So in case of high loss, there will be lesser retransmission and hence improvement in throughput. Also, with higher window size, more number of packets can be sent and they can be individually acknowledged. So even if the base packet is not getting acknowledged, other packets in the buffer may get acknowledged and when the base packet gets acknowledged finally then the protocol will move the window quickly to the right by more number of packets.

This is why its performance is better.

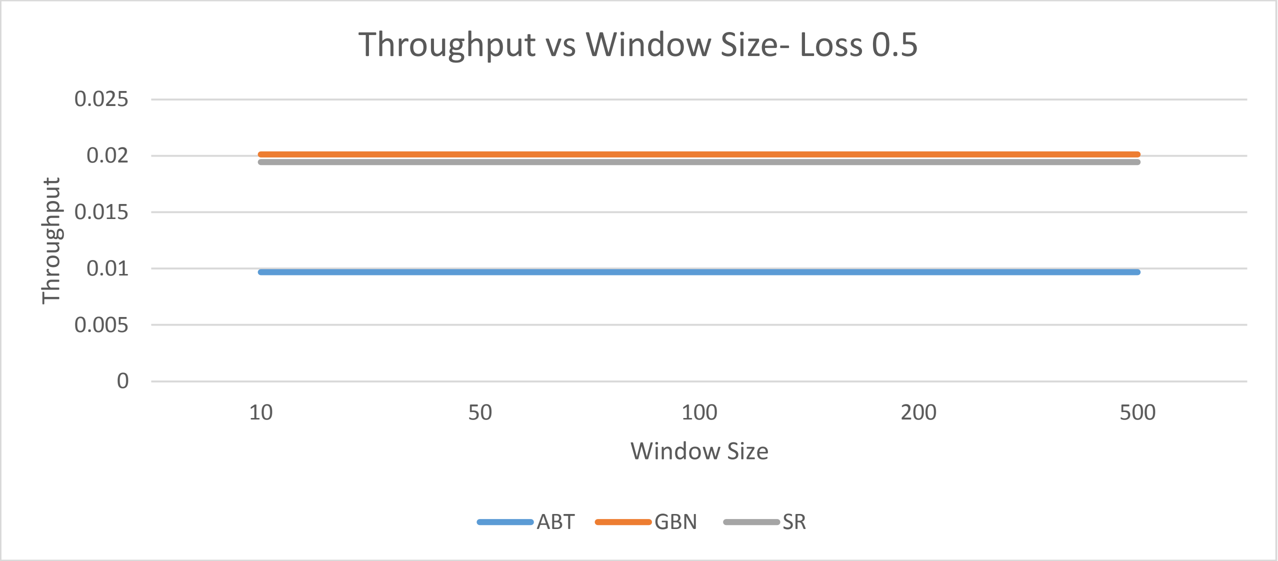
Experiment 2

In this experiment, we are comparing the performance of protocol at a given loss for varying window size. The value for ABT will remain constant as it does not have any window.

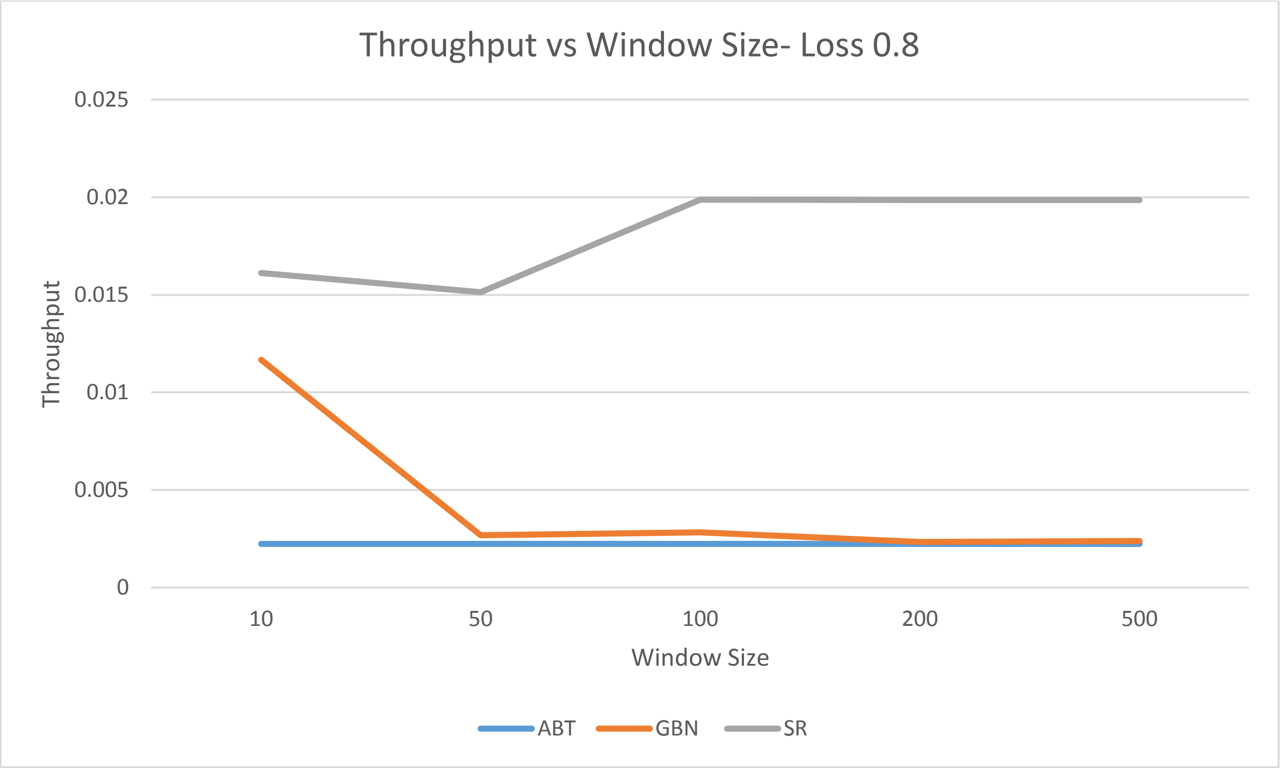
Loss 0.2:



Loss 0.5:



Loss 0.8:



Observation and Analysis:

1. **Performance of GBN decreases as window size increases.** As explained above, in case of retransmission, GBN will retransmit all the packets inside the window. This will increase the congestion to a great extent. The network will get flooded with retransmitted packets and the

protocol will spend most of the resources in resending the same packets (out of which many could have been successfully received at the receiver).

2. **Performance of SR increase as window size increases.** As explained above, if the window size is too high then many packets will be in transit. Since packets are buffered at both sender and receiver, out of order packets are not discarded. Every packet has its own timer so there transmission and retransmission will occur in parallel. Even if the base packet or the packets closes to the base are getting retransmitted again and again, there is a high possibility that during this time other packets inside the window will get eventually acknowledged. The protocol will take advantage of this and when the base packet finally gets acknowledged, then the window will shift to the right (by possibly more than one packets) allowing many new packets to be sent immediately.
3. **The decrease in SR and GBN for window size 50 might be due to the selection of timeout estimation function.**