Xingkun Yin

U44255956

Jingyu Su

U58115442

Compilation instruction:

To run GBN or SR, go to GBN or SR folder and run:

* javac \*.java
* java Project

Description of trace file header:

To get our trace, we piped the output of “java Project” to a txt files.

The header of the trace files are as below (from trace/sw/sw\_loss\_corruption.txt)

Inputs are :

Enter number of messages to simulate (> 0): [10]

Enter packet loss probability (0.0 for no loss): [0.0]

Enter packet corruption probability (0.0 for no corruption): [0.0]

Enter average time between messages from sender's layer 5 (> 0.0): [1000]

Enter window size (> 0): [8]

Enter retransmission timeout (>0.0) [15.0]

Enter trace level (>= 0): [0]

Enter random seed: [0]

100

0.2

0.2

1000

1

30

3

1234

Each input is listed below the requirement. This should be straight forward.

==============================================

Stop & Wait (window = 1) Working correctly

Score is out of 10 pts

==============================================

For stop and wait, we tested several cases by setting the window size to 1 in SR.

works for no loss and no corruption

* Our code work for this one and the trace is in trace/sw/sw\_noloss\_nocorruption.txt.

# Graphical user interface, application Description automatically generated

works for loss and no corruption

* Our code show recovery from loss
* Trace can be found in trace/sw/sw\_loss\_nocorruption.txt
* recovery from DATA loss, error detection by timeout
  + Graphical user interface, application, Word

    Description automatically generated
  + The screen shot above shows a data packet from A is lost , it is detected by timeout and resent. The packet is then successfully collected by B.
* recovery from ACK loss, error detection by timeout
  + Graphical user interface, application, Word

    Description automatically generated
  + The above is a screen of a loss in ACK packet and was detected by timeout.
* A computer screen capture

  Description automatically generated with medium confidence
* The above is the result for loss but no corruption, stop and wait protocol. It can be found in trace/sw/sw\_loss\_nocorruption.txt.

works for corruption and no loss

* Our code show recovery from corruption
* In our design, when receiving a corrupt packet the sender and receiver both send the previous packet, so it does not wait for time out
* Trace can be found in trace/sw/sw\_noloss\_corruption.txt
* recovery from DATA corruption
  + Graphical user interface, application, Word

    Description automatically generated
  + The screen shot above shows a data packet from A is corrupted and detected by B. The packet is resent and then successfully collected by B.
* recovery from ACK corruption
  + Graphical user interface, application, Word

    Description automatically generated
  + The above is a screen of a corruption in ACK packet and was detected by A.
* Graphical user interface, application

  Description automatically generated
* The above is the result for corruption but no lost, stop and wait protocol. It can be found in trace/sw/sw\_noloss\_corruption.txt.

works for both loss and corruption

* A computer screen capture

  Description automatically generated with medium confidence
* The above is the result for corruption andlost, stop and wait protocol. It can be found in trace/sw/sw\_loss\_corruption.txt.

==============================================

SR (window > 1)

Score is out of 90 pts

==============================================

C1: works for no loss + no corruption

* Text, letter

  Description automatically generated
* Our code work for this one and the trace is in trace/sr/ sr\_noloss\_nocorruption.txt

**For the following scenario, C2 to C5, the trace are in trace/sr/ sr\_loss\_corruption.txt**

C2: identify (on output trace) case where ack is lost/corrupted and a later cumulative ack moves sender window by more than 1

* Text

  Description automatically generated
* In the image above, the ACK packet is lost can cause the window in A to move from base at 10 to 12.

C3: identify (on output trace) case where when data packet is lost/corrupted, and data is retransmitted after RTO

* Text

  Description automatically generated
* In the screen shot above, the Data packet is dropped

C4: identify (on output trace) case where when data packet is lost/corrupted, and data is retransmitted after receiving duplicate ack

* Text

  Description automatically generated
* In the screen shot above, the data packet is lost and it is retransmitted after a duplicate ack is received.

C5: identify (on output trace) case where when data packet is lost/corrupted, and the retransmitted data is delivered and a cumulative ack moves the sender window by more than 1

* Text

  Description automatically generated
* In the screen shot above, a data packet is lost and cause the sender window to go from 12 to 14.

**Logistics**

* For our GBN Sack Protcol
  + Sender
    - We have an array list to hold all packets from layer 5.
    - On receiving the message from layer 5, we convert the message into a packet with appropriate sequence number and store it in to the buffer. We also check if the window is at its capacity, it not we send window until we reach the capacity.
    - On receiving ACK
      * If corrupt, we send the base packet
      * If not corrupt
        + If the seqnum is duplicated, we send the base
        + If it is in window, we move the base to the sequence number after ACK number and send more packets to fit the window capacity
  + Receiver
    - Receiver has a buffer of the same size of the window
    - On receiving the corrupted or duplicated packet, reply an ACK with the base
    - On receiving a packet with seqnum == base, move the base up until there are no packet with the sequence number, upload the newly received packet and the packets with seqnum base just looped through and delete them from base.
* Our checksum related functions include addChecksum(), calculateChecksum() and evaluateChecksum()
  + The way the checksum is calculated is:
    - Checksum = seqnum + seqnum + Character.getNumericVale() for evey character in the payload.
* Our code is readable as each state in the FSM is in a single clause of if else clauses. We have a lot of comments describe how the code works. The helper functions and attributes we added are in java regions, i.e.
  + //region helper function
  + //endregion
* Possible tradeoff:
  + We are storing all packet from layer 5 to A, and related time in a array list which is the same length as the amount of packets A receive from layer 5.
* Extension
  + We may reduce the usage of memory by using a constant array to store the information and delete the packets in the past that we no longer need.
* Compilation instruction is at the top of this report.

**Statistics**

Justification for retransmission timer:

Our retransmission timer is always 90. This is because under RTO of 90, our program is able to work correctly, receiving everything sent in order. (Doesn’t mean it can’t work correctly with other RTO value!)

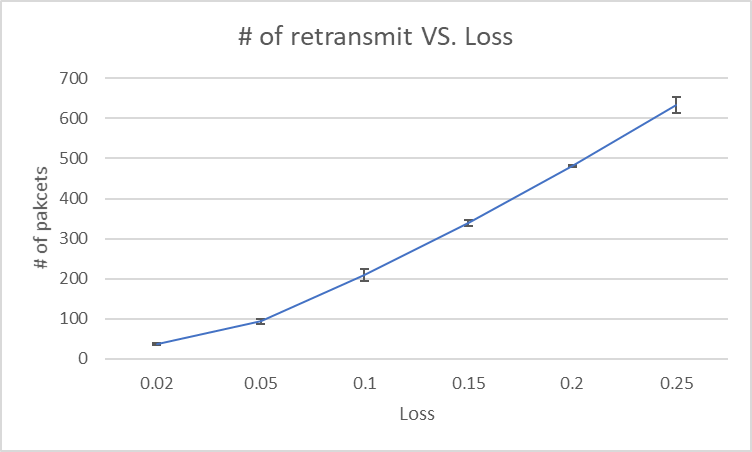
# retransmits under no loss & no corruption

Text, letter

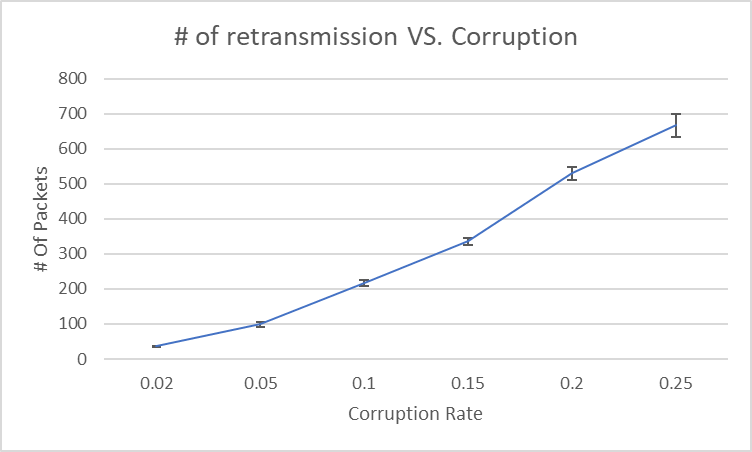
Description automatically generated

From previous sections, we can see that when there is no loss and corruption, the number of retransmissions is 0.

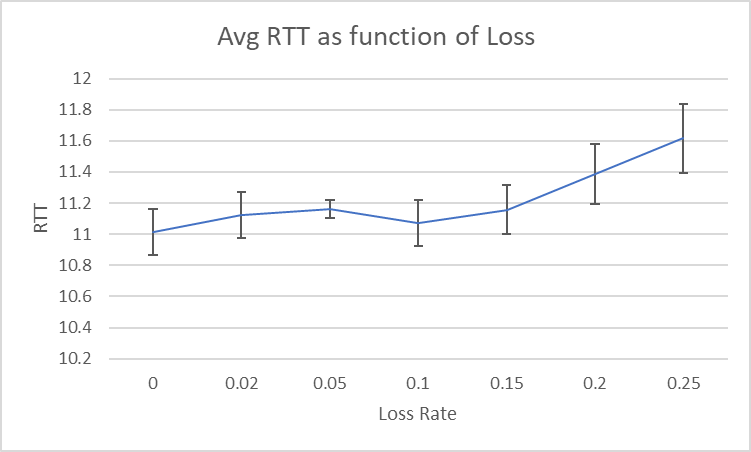
# retransmits as function of loss

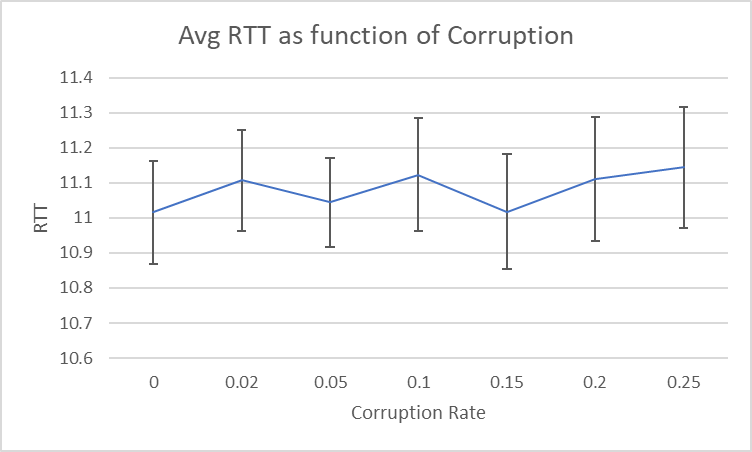


# retransmits as function of corruption

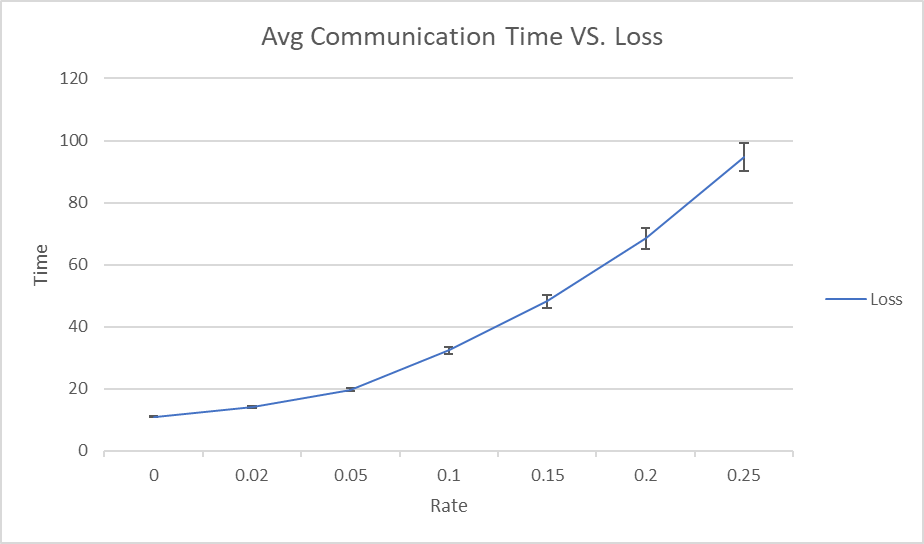


Average RTT as function of loss/corruption

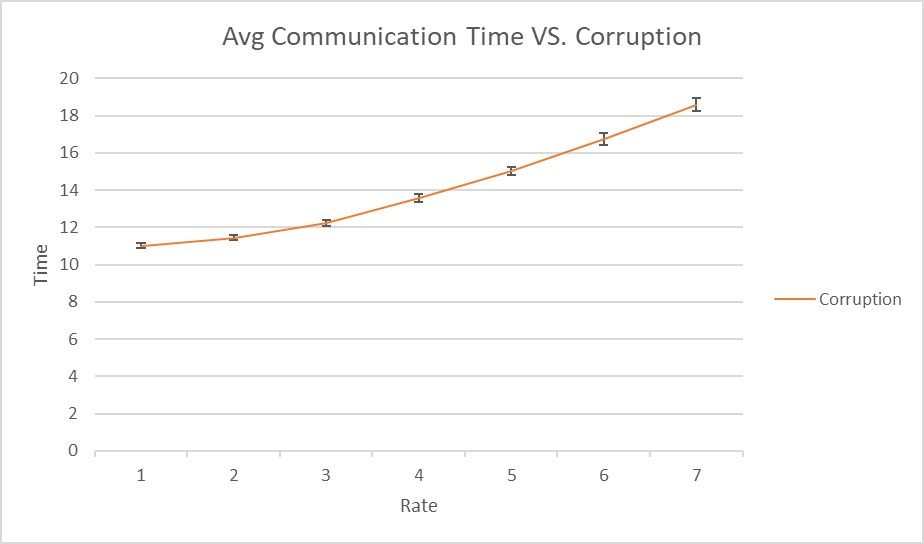




Average time to communicate packet vs. Loss



Average time to communicate packet vs. Corruption



**Evidence of several runs**

This can be found in the .csv and.excel files submitted on csa machines.

Our program is well tested, we go over the traces thoroughly.

================================================

GBN with SACK option (window >1)

Score is out 100 pts

================================================

same behavior as SR for no loss & no corruption

* Text, letter

  Description automatically generated
* From trace trace/gbn/gbn\_noloss\_nocorruption.txt

works for loss and no corruption

* Text, letter

  Description automatically generated
* From trace trace/gbn/gbn\_loss\_nocorruption.txt

works for no loss and corruption

* Text

  Description automatically generated
* From trace trace/gbn/gbn\_noloss\_corruption.txt

works for both loss and corruption

* Text, letter

  Description automatically generated
* From trace trace/gbn/gbn\_loss\_corruption.txt

Annotations on traces show difference between SN and GBN+SACK

* Text

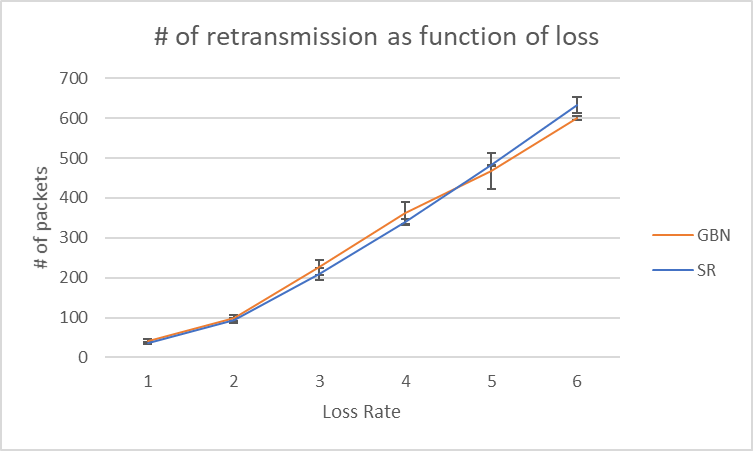
  Description automatically generated
* Notice GBN send a window, 2 packets here which does not happen in SR
* From trace trace/gbn/gbn\_loss\_corruption.txt

**Logistics**

* Our code is readable as each state in the FSM is in a single clause of if else clauses. We have a lot of comments describe how the code works. The helper functions and attributes we added are in java regions, i.e.
  + //region helper function
  + //endregion
* For our GBN Sack Protcol
  + Sender
    - As SR, we have an array list to hold all packets from layer 5.
    - On receiving the message from layer 5, we convert the message into a packet with appropriate sequence number and store it in to the buffer. We also check if the window is at its capacity, it not we send window until we reach the capacity.
    - On receiving ACK
      * If corrupt, we send the whole window
      * If not corrupt
        + If the seqnum is duplicated, we send the packets in window that is not in SACK
        + If it is in window, we move the base to the sequence number after ACK number and send more packets to fit the window capacity
  + Receiver
    - Receiver has a array list for SACK to record the 5 latest received packets that are in window
    - Receiver has a buffer of the same size of the window
    - On receiving the corrupted or duplicated packet, reply an ACK with the base and a SACK
    - On receiving a packet with seqnum == base, move the base up until there are no packet with the sequence number, upload the newly received packet and the packets with seqnum base just looped through and delete them from base.

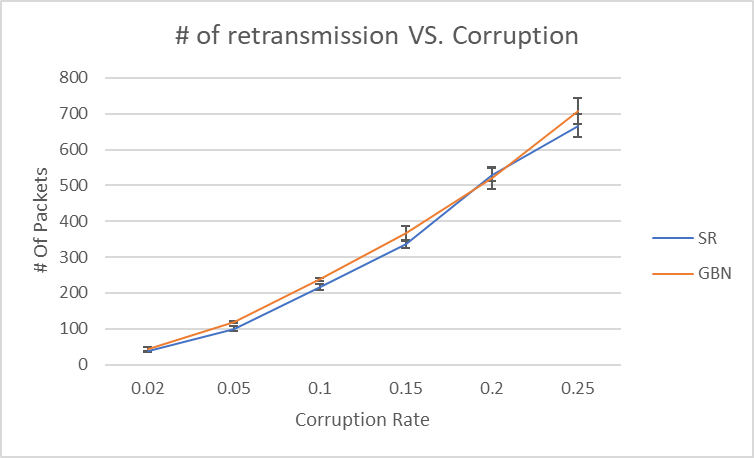
**Statistics**

**# of retransmission as function of loss:**



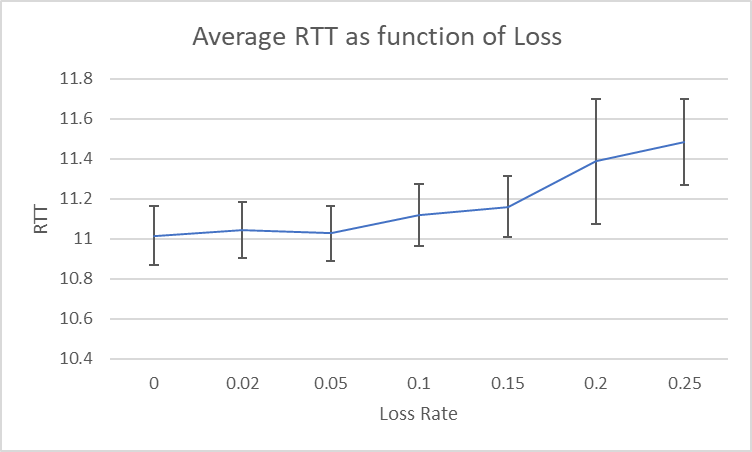
We can see that GBN makes more retransmissions when loss rate is low, but when loss rate is higher, SR tend to make more retransmissions.

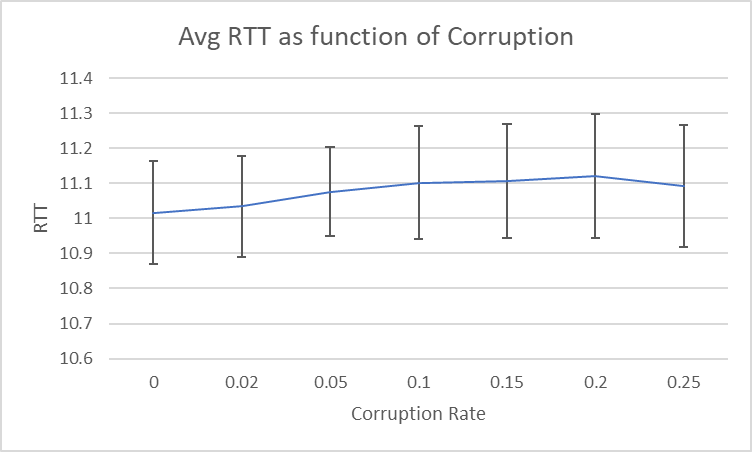
**# of retransmission as function of corruption:**



We can see that, with duplicate Ack transmitted by B upon receipt of corrupted packets, GBN in general do more retransmits compared to SR. However, since GBN has the potential to retransmit multiple corrupted packets at a time, as we will see later, the performance under corruption is very similar.

**Average RTT as function of loss/corruption**

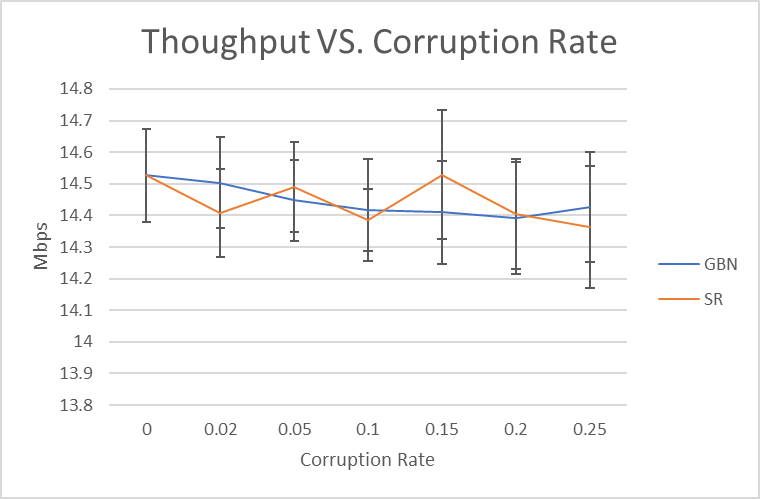


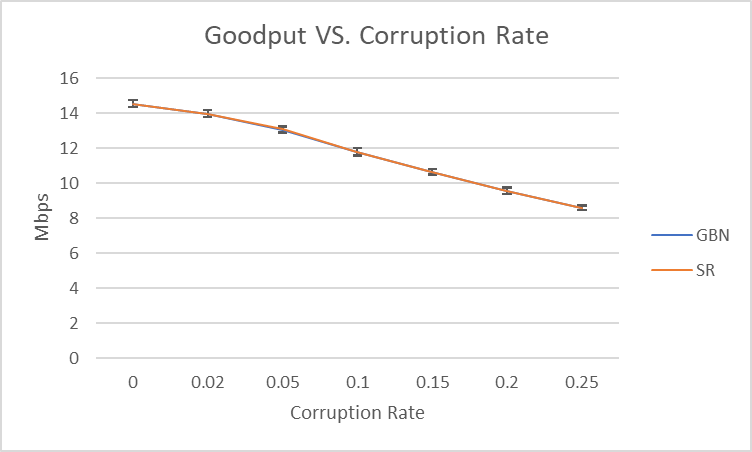


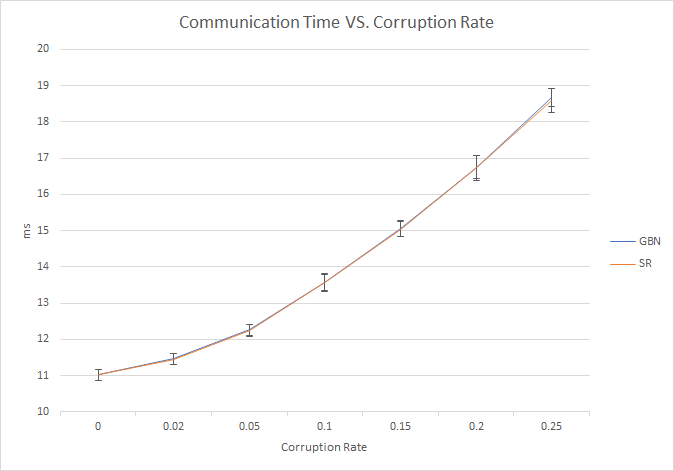
We can see that average RTT under loss for GBN is better compared to SR, and similar under corruption.

We plotted Throughput, Goodput, average packet delay of SR vs. GBN with ACK under corruption as loss, as follows:

**Throughput, Goodput, average packet delay, SR vs. GBN+SACK under corruption:**

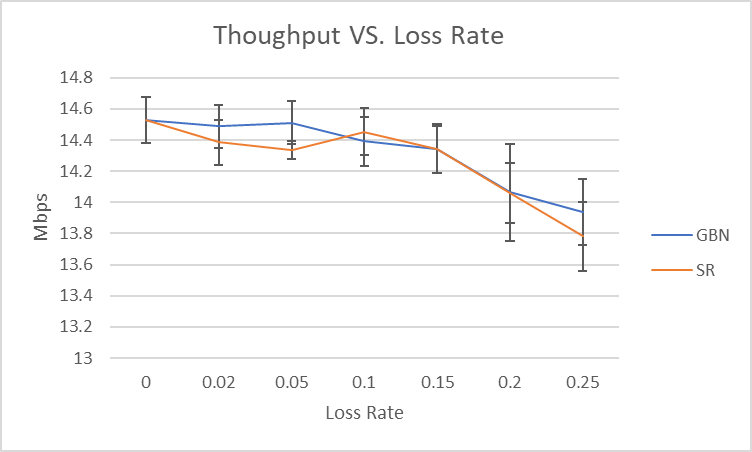




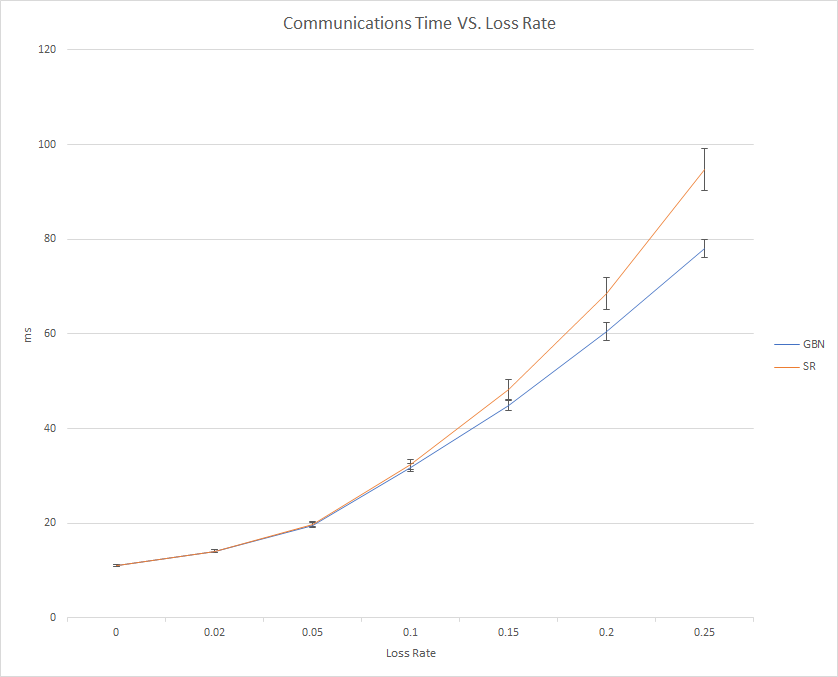
****

We can see that the two difference mechanisms show very similar performance under corruption, this is because in both mechanisms, both sender and receiver retransmit when corrupted packet is received.

**Throughput, Goodput, average packet delay, SR vs. GBN+SACK under losses：**





****

The difference of performance is much larger compared to corruption. This is because when packets are lost, it is likely to trigger timeout. Since higher loss rate will result in higher possibility of other lost packets in the window, applying GBN will speed up the retransmission of successive packets, therefore result in better performance.

**Evidence of several runs**

This can be found in the .csv and.excel files submitted on csa machines.

Our program is well tested, we go over the traces thoroughly.