Wassim Gharbi
Erik Laucks
Zurabi Mestiashvili

April 3rd, 2018
CS 305: Computer Networks
Submitted to Prof. Amir Sadovnik

Project #2: Transport Layer Implementation

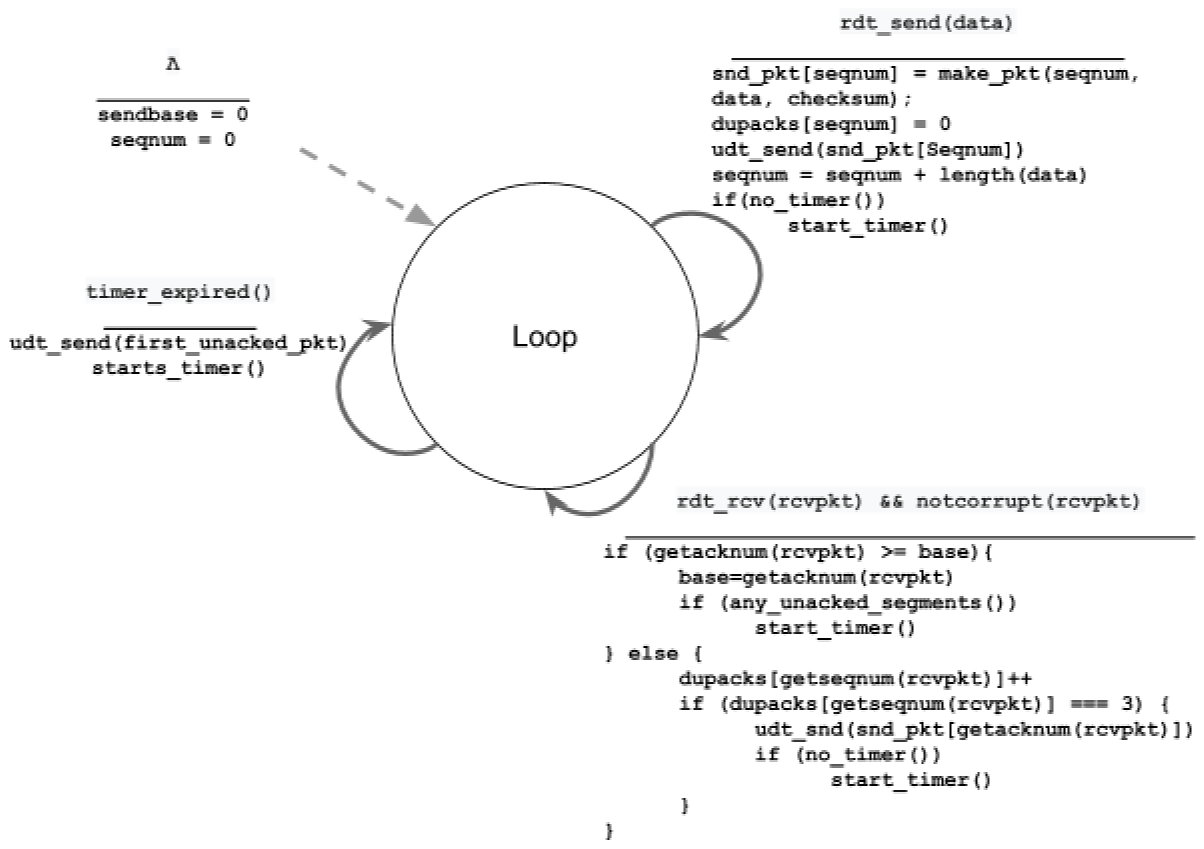
I. Introduction

Through this project, we attempted to simulate the functionalities of a simple transport layer protocol inspired by TCP. Our simulation encompasses both the sending and receiving ends of the transport protocol as well as the interaction between them in order to provide (unidirectional) reliable data transfer with added flow control. Our implementation relies on an abstraction of hardware and software structures within a simulated environment in order to emulate the behavior of a real network architecture. To do so, we also added simulated package loss as well as package corruption (with variable probabilities). In order to study the behavior of transport layer protocols, we implemented two variants of our network layer protocol: A variant that buffers out of order packets and another variant that ignores them (meaning that the sender will have to resend the out-of-order packages). As an extension to the project, we also experimented with the flow control mechanism by varying its parameters (buffer size and read speed of the application layer). In this report, we present the results of our findings as well as the methods we used to implement our simulation.

II. Simulated Protocol

We have designed our protocol to be analogous to the real-world TCP implementation with a few simplifications that make the simulation easier to understand. Our protocol is very similar to TCP in that it keeps all of its structural components such as the buffer, sending and receiving windows, packet headers (including checksums, sequence numbers, etc.), the use of acknowledgment packets (**`ACKs`**) and the accurate simulation of the unreliable data transfer layer (along with randomized packet loss and packet corruption as packets travel through the layer).

To simplify the simulation, we had to change some of the aspects of a real-world internet transport protocol by removing some of the features and changes some of the ways a real-life protocol would behave in certain situations. For example, in our simulation, packets are modeled through the **`Packet`** Java class and thus fields such as the sequence number, **`ACK`** number and checksum are modeled as instance variables of every **`Packet`** object. This implies that we do not use "TCP headers" per se (as prepended text data), although all the necessary information is carried along the packet object. Another minor discrepancy between our protocol and a real-world TCP protocol is the absence of the handshake component of a TCP connection (which implies the absence of the **`SYN`** flag on the TCP "header"). Although we implement a dynamic sliding window size (on the receiving end) in order to provide flow control, we do not maintain a congestion window. Therefore, out protocol lacks the ability to mitigate congestion on the network (assuming a large enough buffer size on the receiving end).



```
Λ
_______________
sendbase = 0
seqnum = 0
```

```
rdt_send(data)
_______________
snd_pkt[seqnum] = make_pkt(seqnum,
data, checksum);
dupacks[seqnum] = 0
udt_send(snd_pkt[Seqnum])
seqnum = seqnum + length(data)
if(no_timer())
      start_timer()
```

```
timer_expired()
_______________
udt_send(first_unacked_pkt)
starts_timer()
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_______________
if (getacknum(rcvpkt) >= base){
      base=getacknum(rcvpkt)
      if (any_unacked_segments())
            start_timer()
} else {
      dupacks[getseqnum(rcvpkt)]++
      if (dupacks[getseqnum(rcvpkt)] === 3) {
            udt_snd(snd_pkt[getacknum(rcvpkt)])
            if (no_timer())
                  start_timer()
      }
}
```

Figure 1. Finite State Machine description of the sender end-system

To simulate a real-world TCP implementation, our protocol handles loss and corruption the same way as TCP Reno (omitting the congestion control mechanisms): As demonstrated in *Figure 1.*, the sender end-system initializes a timer after the first packet is sent and resets it when a correct **`ACK`** is received. When a packet loss occurs at the unreliable data transfer layer, the timer will eventually expire, thus triggering the sender to resend the unacked packet with the smallest sequence number. In addition to the timeout mechanism, our protocol also implements re-sending packets on three successive out-of-order ACKs. On the receiving end, we implement checksum tests in order to protect against corrupt packets.

In summary, our simulation features a protocol that is highly inspired by TCP Reno and includes the following features:

- Unidirectional reliable data transfer between a sender and receiver end-systems
- Sender-side timer to keep track of lost/corrupt packets and resend them as necessary
- Fast retransmit on three duplicate acknowledgement packets
- Protection against corruption of packets (checksum)
- Flow control through the variable window size
- Randomized polling of the receiver buffer by the application layer (follows a Gaussian distribution)
- Parameterized loss and corruption probabilities, window sizes, MSSs and time between messages sent
- Data segmentation and reassembly for large messages over multiple packets

The header of our TCP Packet (modeled by the **`Packet`** class) is formatted as follows:

Instance variable (equivalent to a header field)	**Usage**
`int seqnum`	Sequence number of the packet to be transmitted or received, starts at 0
`int acknum`	Acknowledgement number, starts at 0 and denotes the initial sequence number added to the total size of data messages last acknowledged at the receiver

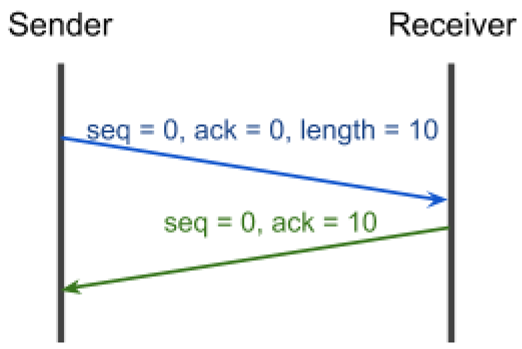
`int checksum`	Checksum computed by adding the sequence number to the acknowledgement number and to the ASCII representation of every character of the payload message
`int status`	One of **0** (not usable, outside of window scope), **1** (usable, not sent yet) , **2** (sent), **3** (sent and acknowledged)
`int rcvwnd`	Size of the receiver window (used to limit the sending window and thus provide flow control)

Figure 2. TCP Header model

Expected Exchanges

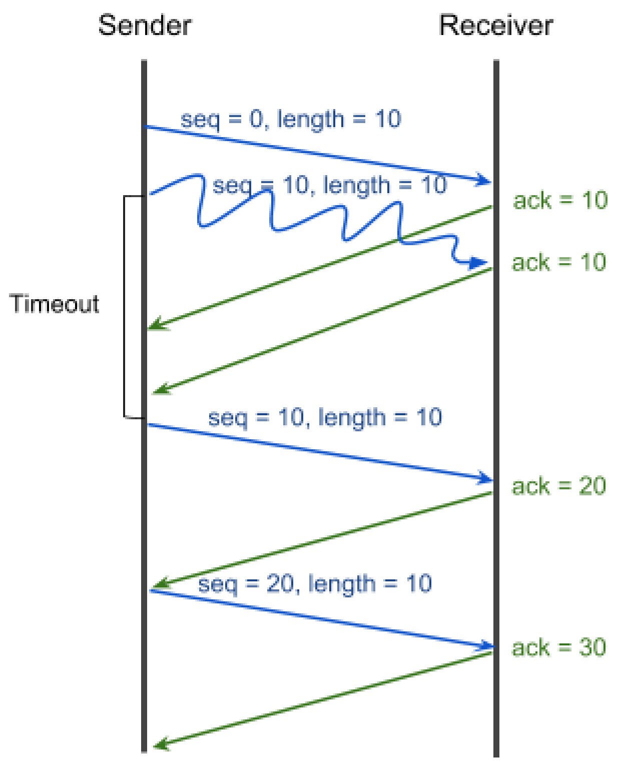
No Corruption or Loss

In the case of no corruption and no packet loss (on either side), we expect our protocol to show a normal exchange of packets with the sender packing data and sending it through the network layer followed by an acknowledgement from the receiving side :



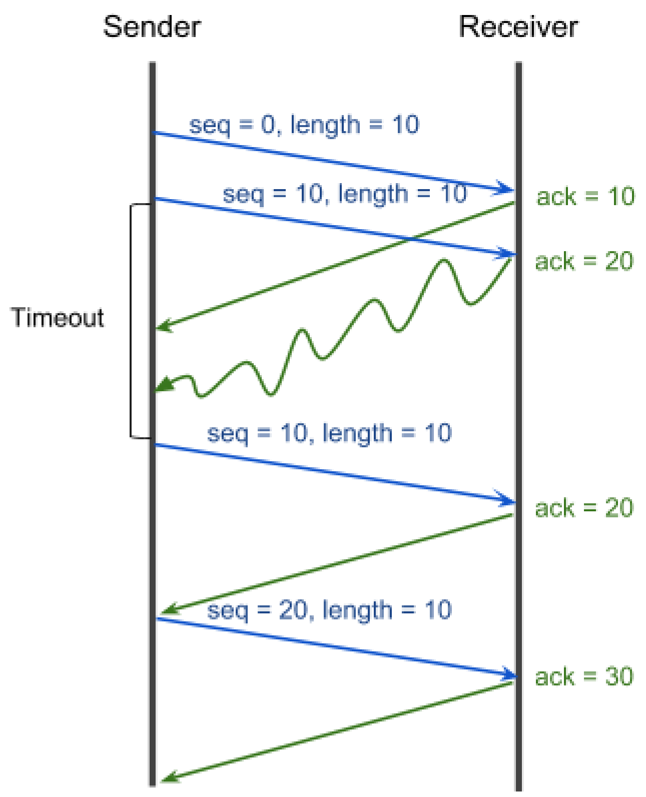
Corrupt Data Packet

In the case of a corrupt packet from the sender to the receiver, the sender will timeout and thus retransmit the packet that got corrupt:



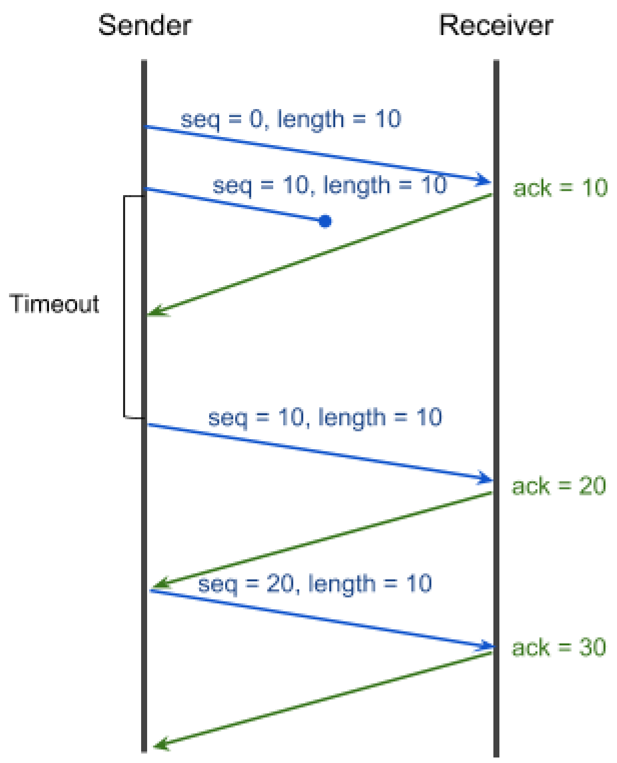
Corrupt ACK Packet

In the case of a corrupt packet from the receiver to the sender (ACK packet), the sender will eventually timeout and thus retransmit the packet that got corrupt:



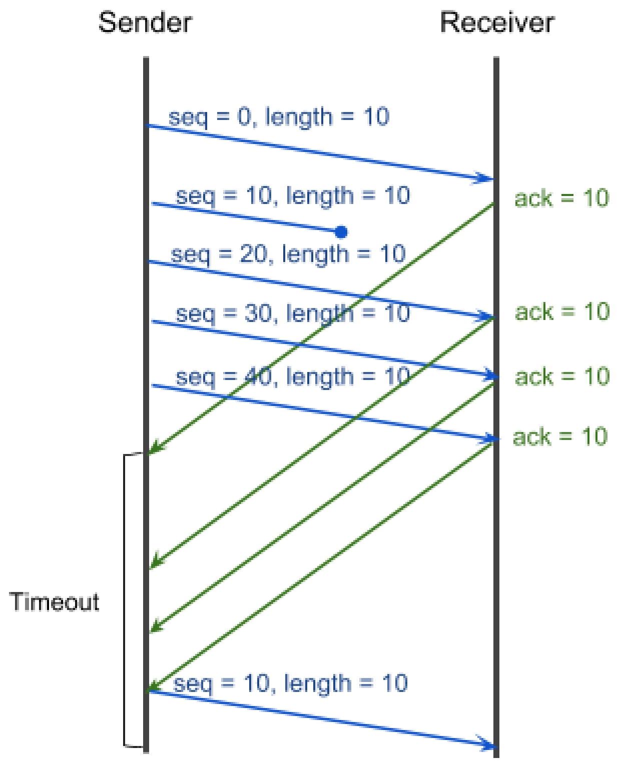
Lost Packet

In case of a lost packet, either a timeout occurs and the sender resends the timed out packet or (if other packets have been sent after it), a fast-retransmit occurs (see next case):



Fast Retransmit

Through our implementation of fast retransmit, our protocol does not need to wait for timer to timeout before resending a packet that was lost or got corrupted. After 3 out-of-order **ACK** packets, the protocol automatically re-sends the packet that the receiving end is expecting as follows:



Flow Control

To implement Flow Control, we added a new header field (**rcvwnd**) on the **`Packet`** class to enable the receiving side of the network to share the size of the receiving window with the sender side. On the sender end-system, we added a functionality that allows the sender to contract its sending window to match the given receiving window. This prevents the sender from sending messages at a faster rate than what the receiver can handle. On the receiving end, the rcvwnd variable is set as **`rcvwnd = maxBufferSize - totalBytesInBuffer`** (in other words, the space left on the buffer in bytes) and is then sent as part of the header of acknowledgement packets. In the event that the receiving window size goes down to 0 and all packets have been acknowledged before the receiver application starts reading from the buffer (in which case there is no way to notify the sender of the change in the receiving window size), we added an exception to the sender's sending condition that allows it to re-send the window base to the receiver on every timeout so that the receiver sends back updates about its receiving window size (through an **`ACK`** packet).

III. Code Design & Data Structures

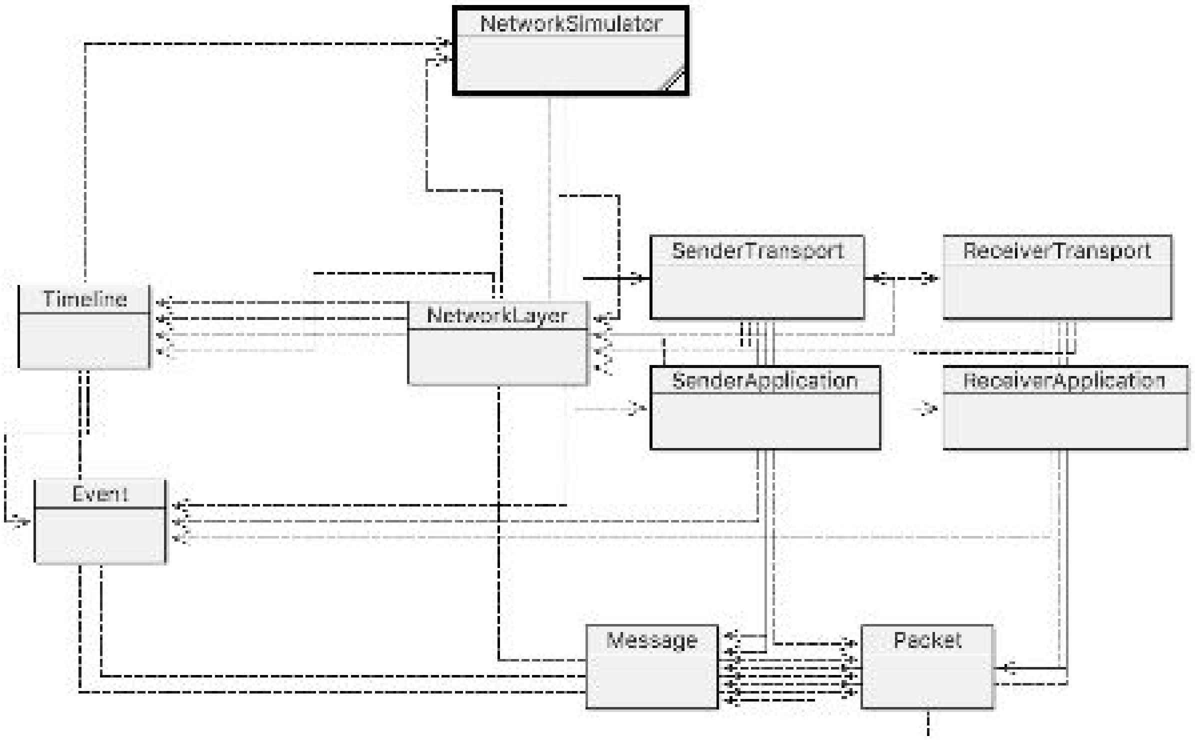


Figure 3. Project Structure

In order to implement all the features discussed in the previous section, we designed multiple abstractions of the various components of the network and transport layers as well as the sending and receiving end-points as follows:

- The **`NetworkSimulator`** class takes the parameters for the simulation, sets up the transport and application layers and runs the simulation based on the given parameters, the definition of the parameters will be detailed in the next section.

- The **`Timeline`** is the main time-keeping component of the simulation, it manages the creation and destruction of events and it also keeps the timer that the sending end-systems uses to keep track of lost and corrupt packets.
- The **`NetworkLayer`** is a simple abstraction of the unreliable transfer layer which given probabilities of corruption and loss, either delivers, fails to deliver or corrupts the packet
- The **`Packet`** class models a single TCP packet that is transferred over the network. In addition to the given initial class, we added multiple header fields as show on *Figure 2.* in order to account for the various functionalities (such as buffering and variable window size for flow control) as well as the checksum generation and verification methods as described in *Figure 2* (**`setChecksum()`** and **`isCorrupt()`**).
- The **`SenderTransport`** and **`ReceiverTransport`** classes model the TCP sending and receiving ends. In addition to setting all the parameters necessary for the transport layer, these classes package the message passed from above (packet creation), send the packet to the network layer, handle the reception, verification (checksum) and buffering of packets and manage the internal states of the receiving buffer, sequence numbers and acknowledgement numbers. The **`SenderTransport`** class also handles timer interrupts to resend packages when a timeout occurs.
- The **`SenderApplication`** and **`ReceiverApplication`** classes model the application layer of the simulation, although in this case their interaction with the other components is minimal (the **`SenderApplication`** reads a message from the given text file and passes it to the sender transport layer whenever instructed to by the **`NetworkSimulator`** while the **`ReceiverApplication`** simply pops a message

from the buffer and prints it out whenever instructed by the **`NetworkSimulator`** class based on a random Gaussian distribution).

IV. Simulator Usage & Parameters

Building the simulation

Build the project using the included Makefile (do not use BlueJ) as follows :

```
$ make cc
```

Running the simulation

The simulation runs in three different debugging modes, and takes multiple parameters that control the simulator's behaviors. **Do not run the simulation inside BlueJ** since we have added ANSI color codes which are not compatible with BlueJ's terminal and will therefore render the output illegible. In order to run the simulation, use the command

```
$ make run
```

to run a predefined test or use

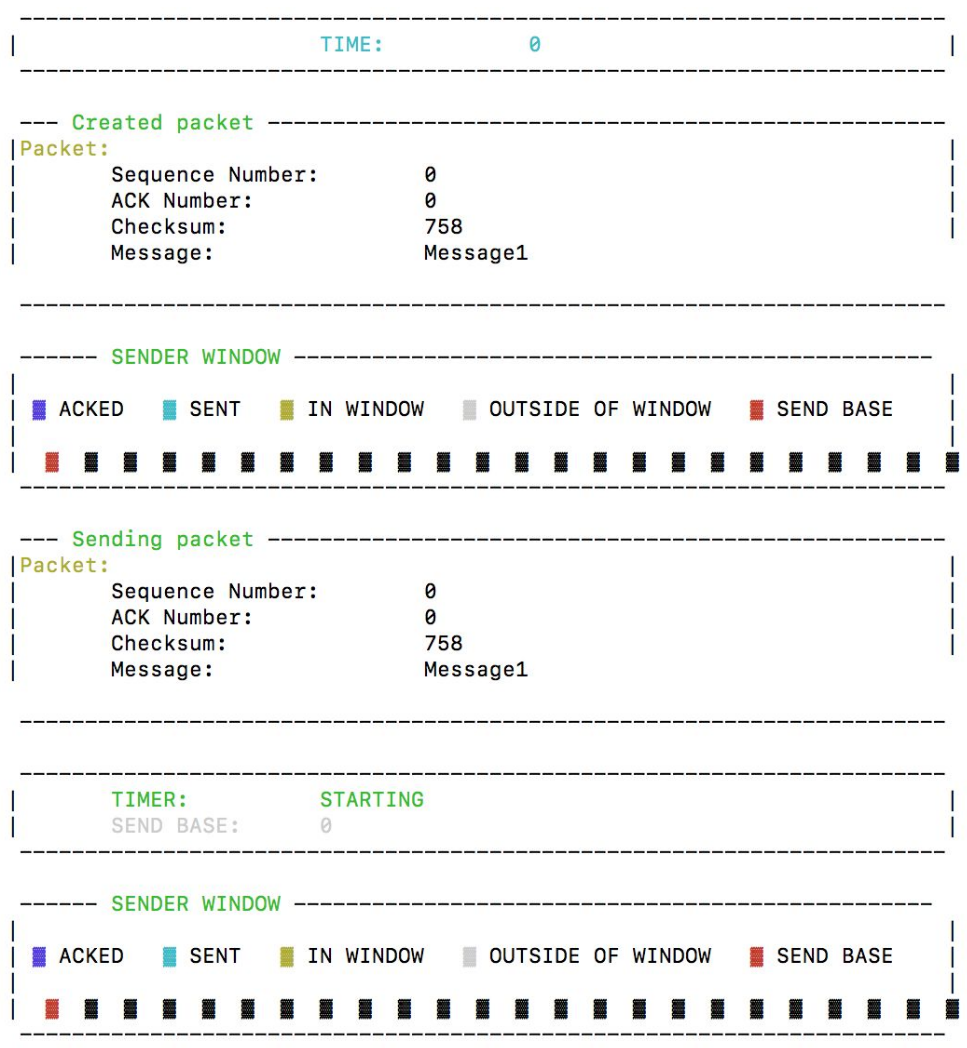
```
$ java NetworkSimulator [file] [time-between-msgs] [loss%] [corruption%] [window] [mss]
[protocol] [debug] [rcv-buffer-length] [timeout-length]
```

to customize the parameters of the simulation. The parameters are as follows:

Parameter	**Usage**
`String file`	Input file containing messages to be sent
`String timeBetweenMsgs`	The time (in timesteps) before messages are sent from the application layer to the transport layer
`String lossPercent`	Probability of a packet loss (for both data packets and ACK packets)
`String corruptionPercent`	Probability of a packet corruption (for both data packets and ACK packets)
`String window`	Size of the receiver window
`String mss`	Maximum size of data contained in a single packet
`String protocol`	Protocol (0 to disable buffering out of order packets and 1 to enable it)
`String debugging`	Tracing level (0: no debugging, 1: timeline events only, 2: timeline events as well as corruption and loss events)
`String maxBufferSize`	The maximum buffer size of the receiver in bytes
`String timeout`	The timeout in timeline "ticks"

Output

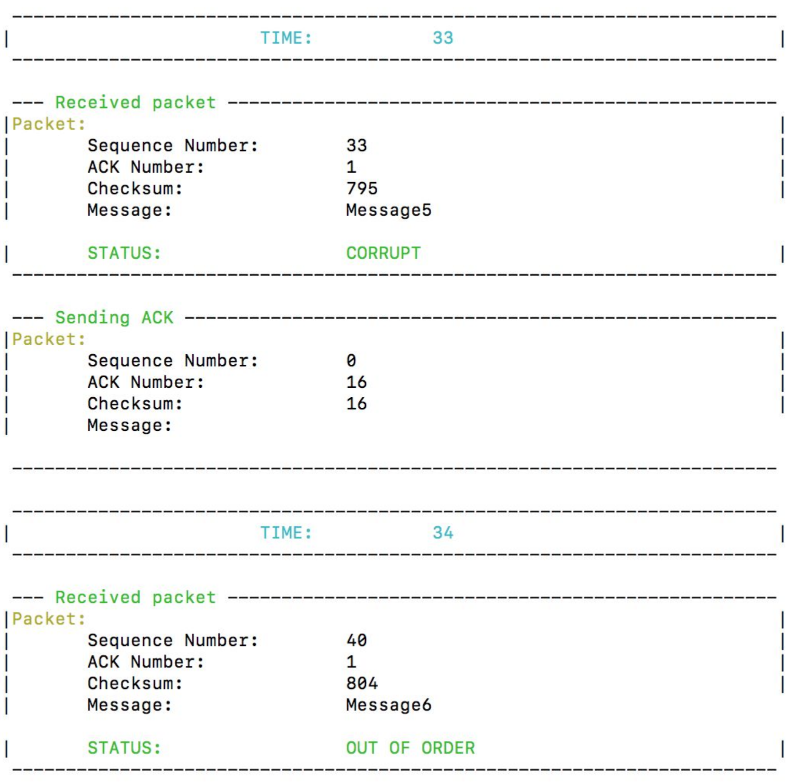
The output of the simulation consists of a timeline of events, the time associated with them as well as the status of the sender buffer/window and the statuses of packets (whether corrupt, lost or delivered. An example output looks as follows:



```
 -----------------------------------------------------------------------
|                        TIME:           0                              |
 -----------------------------------------------------------------------

 --- Created packet ----------------------------------------------------
|Packet:                                                                |
|       Sequence Number:        0                                       |
|       ACK Number:             0                                       |
|       Checksum:               758                                     |
|       Message:                Message1

 -----------------------------------------------------------------------

 ------ SENDER WINDOW --------------------------------------------------
|                                                                       |
| 🟪 ACKED   🟦 SENT   🟨 IN WINDOW   ⬜ OUTSIDE OF WINDOW   🟥 SEND BASE  |
|                                                                       |
|  🟥 ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛
 -----------------------------------------------------------------------

 --- Sending packet ----------------------------------------------------
|Packet:                                                                |
|       Sequence Number:        0                                       |
|       ACK Number:             0                                       |
|       Checksum:               758                                     |
|       Message:                Message1

 -----------------------------------------------------------------------

 -----------------------------------------------------------------------
|       TIMER:          STARTING                                        |
|       SEND BASE:      0                                               |
 -----------------------------------------------------------------------

 ------ SENDER WINDOW --------------------------------------------------
|                                                                       |
| 🟪 ACKED   🟦 SENT   🟨 IN WINDOW   ⬜ OUTSIDE OF WINDOW   🟥 SEND BASE  |
|                                                                       |
|  🟥 ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛ ⬛
 -----------------------------------------------------------------------
```

Figure 4. Example output from the simulator

Figure 4. shows events happening in a single time step (time step 0), where data ("Message1") was received from the application layer (and thus a packet was created for it) then since the send window is currently empty, the packet is immediately sent with a checksum of 758, a timer was also started for the packet to check whether it was lost or not later on.



```
 ----------------------------------------------------------------------
|                       TIME:           33                             |
 ----------------------------------------------------------------------

 --- Received packet --------------------------------------------------
|Packet:                                                               |
|       Sequence Number:        33                                     |
|       ACK Number:             1                                      |
|       Checksum:               795                                    |
|       Message:                Message5

|       STATUS:                 CORRUPT                                |
 ----------------------------------------------------------------------

 --- Sending ACK ------------------------------------------------------
|Packet:                                                               |
|       Sequence Number:        0                                      |
|       ACK Number:             16                                     |
|       Checksum:               16                                     |
|       Message:

 ----------------------------------------------------------------------

 ----------------------------------------------------------------------
|                       TIME:           34                             |
 ----------------------------------------------------------------------

 --- Received packet --------------------------------------------------
|Packet:                                                               |
|       Sequence Number:        40                                     |
|       ACK Number:             1                                      |
|       Checksum:               804                                    |
|       Message:                Message6

|       STATUS:                 OUT OF ORDER                           |
 ----------------------------------------------------------------------
```

Figure 5. Corrupt and out of order packets

As packets are received, the simulator will also show whether the packet was corrupt or out of order (*Figure 5.*). To show whether a packet was lost or not, the user needs to activate a higher level of debugging/tracing.

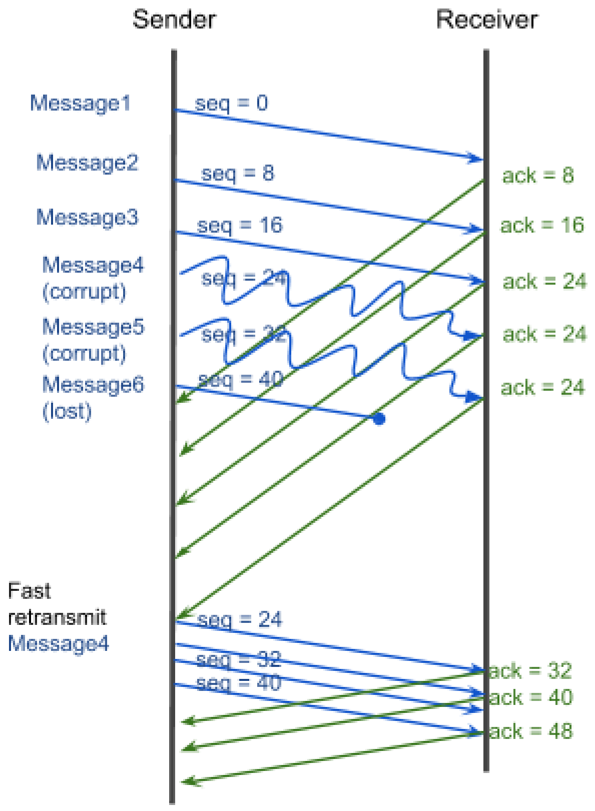
V. Correctness Results

Since our simulation makes use of rich output, we attached the correctness results as separate PDF files and we will proceed to analyze them here. Please refer to the references file for each result.

No Buffering of Out of Order Packets

(Output-1Lost-2Corrupt-NoBuffer.pdf)

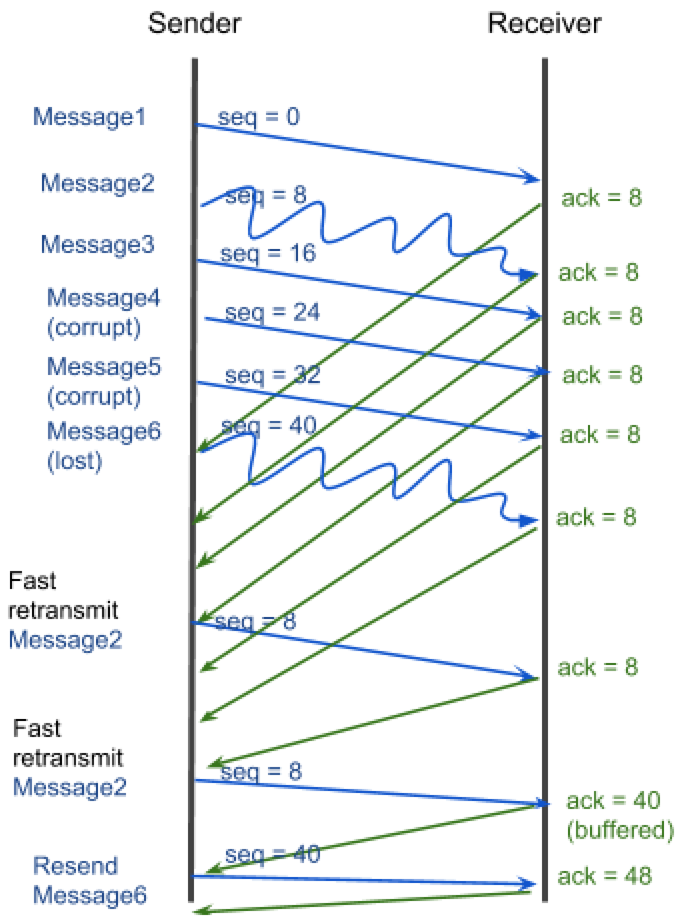
This case presents the transmission of 6 messages over a transport layer with no buffering of out of order packets. In this case, there was a corruption of the 4th and 5th message as well as a lost of the 6th Message, in which case a fast retransmit occured. The diagram for this situation looks as follows:



With Buffering of Out of Order Packets

(Output-WithBuffering1Lost2Corrupt.pdf)

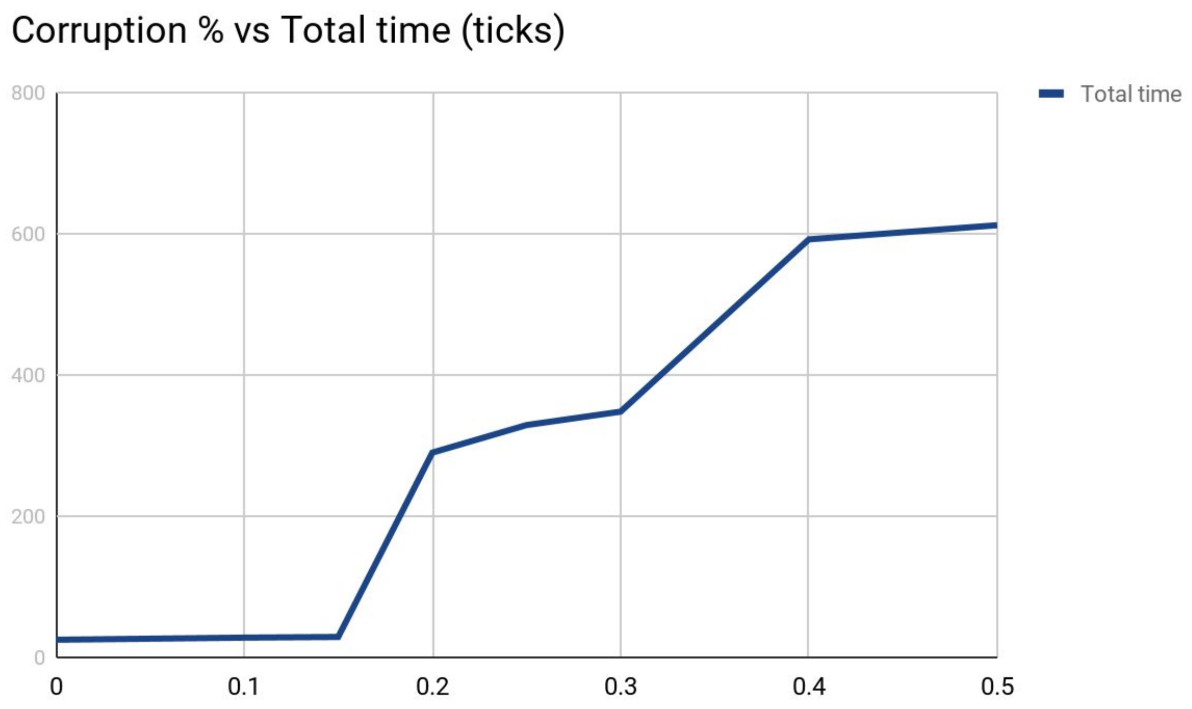
In the case of buffering of out of order packets, the subsequent packets (after a lost or corrupt one) do not need to be retransmitted. This is the case with this experiment where Message2 and Message6 were corrupt then on retransmit, Message2 was lost :



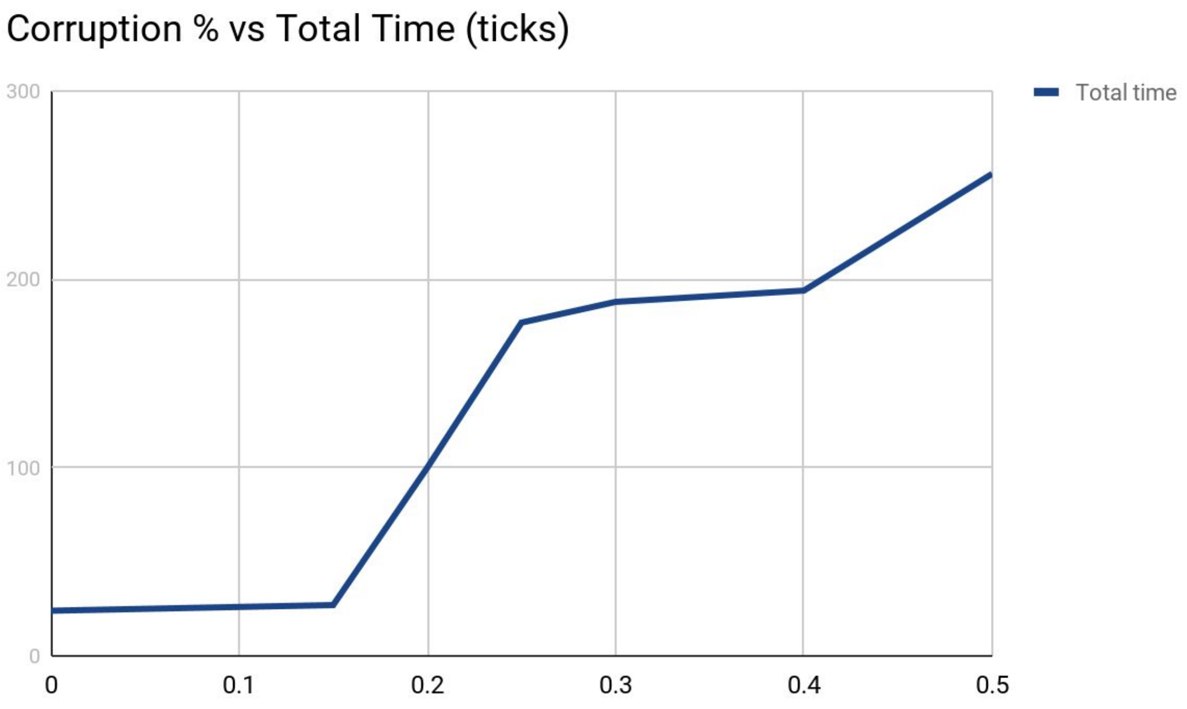
VI. Result Analysis

All experiments have been performed using 6 Messages and a window size of 50 averaged over 3 experiments per parameter combination. All curves have the same overall aspect (as corruption or loss increase, the total time the simulation takes to deliver the packets is obviously increased), however the interesting behavior lies in the maximum time taken (where buffering out of order packets seems to have a greater advantage over no buffering of out of order packets).

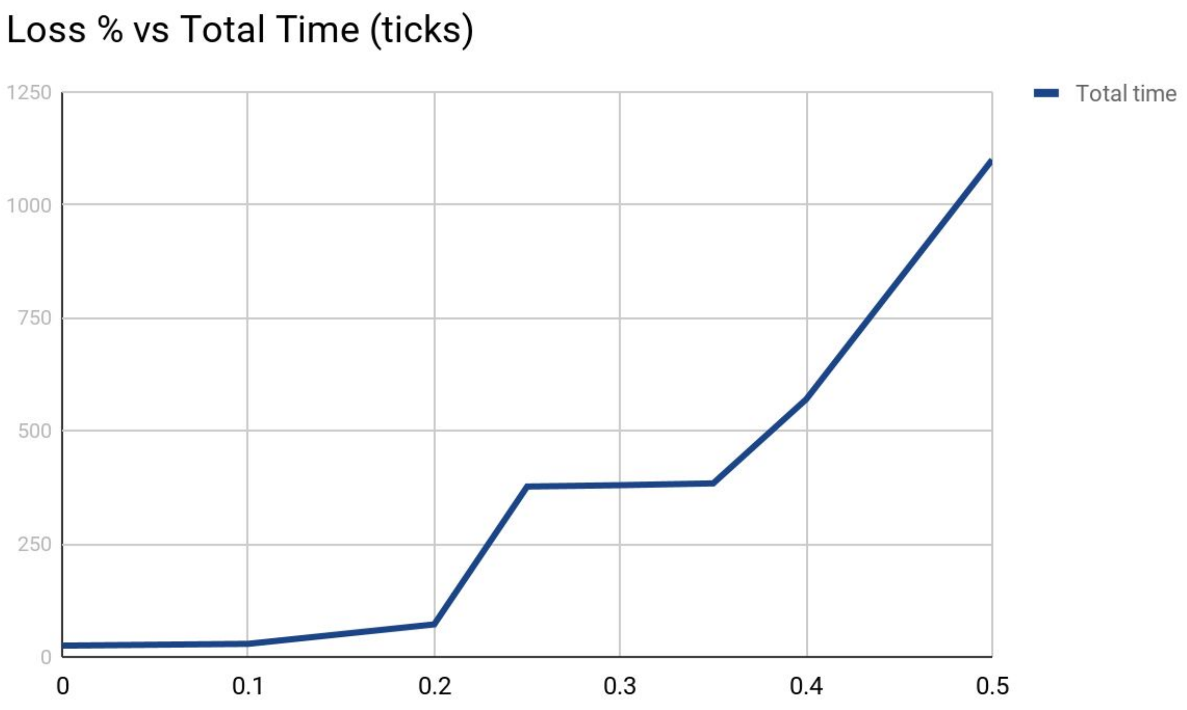
No buffering, No loss, Increasing Packet Corruption



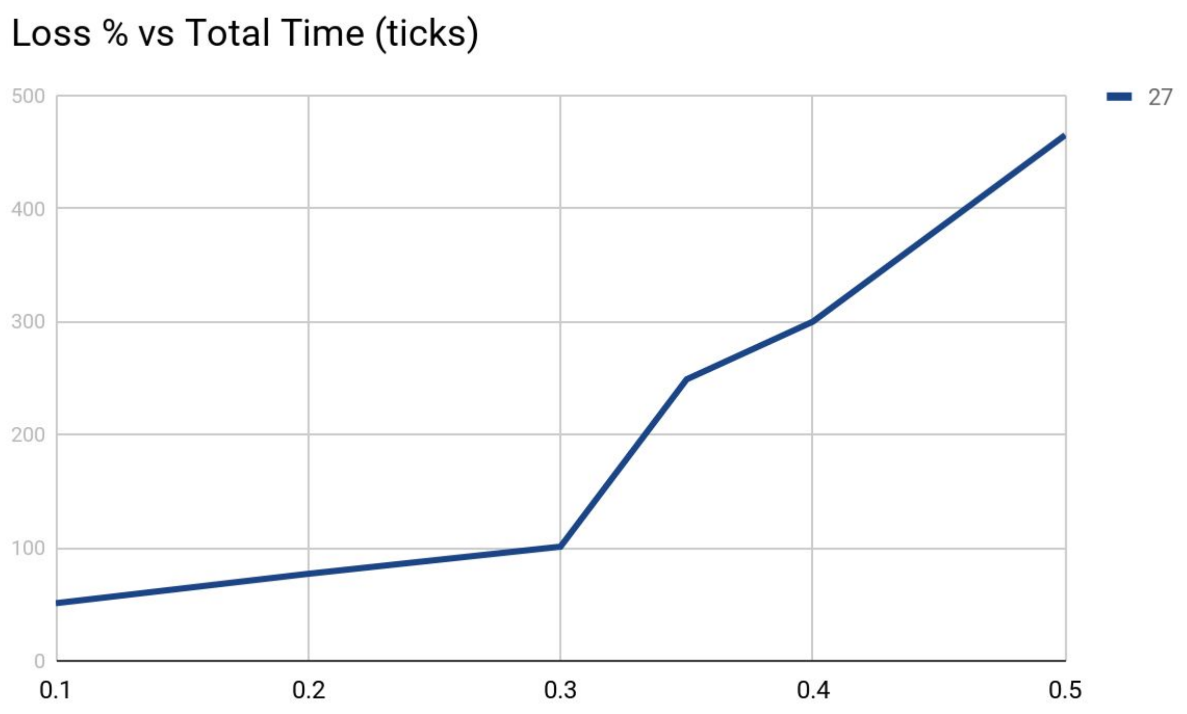
With buffering, No loss, Increasing Packet Corruption



No buffering, No corruption, Increasing Packet Loss



With buffering, No corruption, Increasing Packet Loss



VII. Conclusion

After examining our correctness results and our experimental data and analysis, we have found that the best circumstances for our protocol to run in are those that are very intuitive. As the rate of packet corruption goes up, the time it takes to transmit the same amount of data increases. In the same vein, when packet loss increases, total time rises greatly. In all situations and experiments, buffering out of order packets was a beneficial choice. Even if there were no out of order packets (in a case with no loss or corruption), buffering certainly didn't hurt to have implemented. As the rate of loss and corruption grows, however, we really start to see the benefits of buffering. Our simulation isn't completely accurate, given that this is unidirectional, in a network with just two hosts, without routers or links, and without congestion control, but it gives an analogous representation of real-life implementations of TCP.

VIII. Member Contributions

Erik

Contributed to various parts of the project, such as the **`ReceiverTransport`** class, the buffering on the receiver side, flow control, commenting, tracing, as well as the **`receiveMessage()`**, **`initialize()`**, **`setChecksum()`**, and **`isCorrupt()`** message. Helped with TCP diagrams in the report, as well as conclusion.

Wassim

Wrote a majority of the report and carried out the testing and experimentation. Formalized the software design and created diagrams to represent the program flow. Calculated the theoretical correctness results and verified them. Designed and executed the experiments for the results analysis.

Zura

Contributed to various parts of the project, such as the **`SenderTransport`** class, the dynamic window size for the sender, the logging, the timers and event loop, the fast retransmission, and splitting packets that are too large, as well as the **`sendMessage()`**, **`receiveMessage()`**, **`timerExpired()`**, and **`initialize()`** methods.

IX. References

Kurose, James F., and Keith W. Ross. Computer Networking: A Top-Down Approach. Pearson, 2013.