COMP3331 Assignment 2
Abanob Tawfik
Z5075490

Table of Contents

Implementation of STP protocol, and features successfully implemented. What I would have wanted to Implement

The STP protocol that was implemented followed a similar procedure to TCP. A UDP socket was used to transmit and receive packets on both sender and receiver end. Either sender/receiver can be initiated first as the sender SYN packet is re-transmitted if it does not receive a SYNACK. Log files for each unique file is placed into a new directory.

On the STPSender side, on execution, the file requested is checked for existence within the current working directory. If the file does not exist, the sender exits the program, otherwise the sender will prepare the file for sending. The STPSender will convert the file into a list of packets to send. After the file is prepared the sender want to initialise a decrementing counter of the packets left to send, which is initialised to the size of the list of packets. After the file is prepared, the handshake procedure begins. The sender will send the SYN packet, and wait for the SYNACK response. Upon receiving the SYNACK it will send an ACK to finish the handshake procedure. After the handshake procedure, the STPSender will begin to send data, and receive acknowledgments, on separate threads. The window was a thread safe blocking queue that would add packets to the window if there is room, and upon receiving an ack for that packet it would then finally remove it from the blocking queue. This ensures that any packet once placed inside the window will be guaranteed to be delivered to the receiver before being removed. Packets are sent through the socket after being processed by the PLD module, as per specification. If the receiving thread receives a false ack (NAK), it will re-transmit the packet which was corrupted. If the sender does not receive a response within the socket timeout limit (calculated from the estimatedRTT + gamma*devRTT), the first packet in the window will be re-transmitted. All re-transmissions are handled through the PLD module. Since any packet placed in the window will be guaranteed to be received by the sender, our exit condition will be if all packets have been placed into the window and the window is empty implicating that all packets have been received. After all the data is sent the sender begin the four-way closure, by sending a FIN, waiting for the ACK and then the FIN from the receiver and finally sending an ACK back to the receiver before closing the socket.

On the STPReceiver side, on execution the receiver creates a new directory to place the file the receiver is receiving. The receiver will then begin its side of the handshake, waiting for a SYN from any sender. Upon receiving the SYN it will instantly send back a SYNACK, and then wait for the final ACK from the sender. After the handshake has been completed, the receiver now sits in a while true loop to receive data segments from the sender; this is done on a single thread. Out of order packets received are temporarily stored in the buffer, whilst in order packets are stored in the list of payloads assuming there is no corruption. The acknowledgement number sent back to the sender is the sequence number of the last packet in the payloads list. The list storing the payloads is an ordered set which means it will not store duplicates, and it will always maintain order whenever a packet is added, so re-ordering has no effect. The packet received has a checksum calculation performed and compared to the checksum value in the packet header, if the checksums don't match the receiver send back a false ACK indicating data corruption and requiring of re-transmission. Upon receiving a packet with the FIN flag set to true, which indicates termination on sender side, the receiver will begin termination. First the receiver will send back an ACK to the initial FIN, and then also send its own FIN. Finally, the receiver will wait for an ACK before closing the socket and writing the file. Once the socket is closed, the receiver will do a safety sort on the list of packets, and then write the payloads of every packet in the list, into the output stream for the requested file.

On both sender and receiver side there are two versions of a packet used, one is the processing packet which uses values such as integers, IP's, Booleans to allow for simple processing, and the other version of the packet is the one used by UDP datagrams which is purely byte oriented. A byte buffer is used to convert between the two versions of the packet to allow for simple processing and transmissions.

System feedback is constantly output to standard output, so the programs don't just look frozen and it can be identified when the program is frozen.

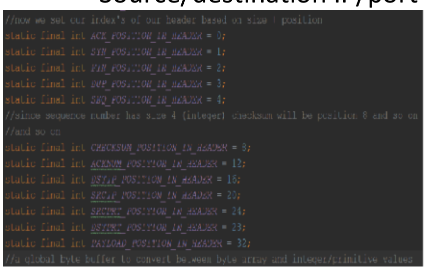
The features that were successfully implemented are
- Sliding window
- PLD module
- Timeout slightly modified to work (put limits)
- Packet transmission
- Cumulative acknowledgement, in a more efficient manner
- Correct logging
- Connection overhead (handshake and termination)
- Correct file writing + guarantee to receive the correct file
- This was also tested from my machine to a friend's machine who lives around 3 hours away (file was received the exact same)
- Buffer implementation
- Maintain packet order with no duplicates or corruption

I have also implemented other features such as, creating files in a new directory and log files are created in a directory based on file requested to not delete old logs, extensive error checking.

Something I would have liked to better implement is the cumulative ack in the manner described, however I couldn't wrap my head around the logic to make it work that way, and the case with the timer, on timeouts since it decreases rapidly to 0, and increases rapidly on receives I had put restraint to reset the timeout value and slightly modify devRTT and estimatedRTT to a more sensible value. This is slightly different however, it is to ensure that my code does not set timeout to 0 when dropping packets, causing full window that will never timeout or to ensure that timeout doesn't reach an insane value due to constant transmission.

A detailed diagram of your STP header and a quick explanation of all fields

The STP header is similar to the TCP header, below in figure 1 will display the header values with their positions.
The following explanation for each field is as follows
- SYN flag – Boolean (1 = true, 0 = false in byte) 1-byte checks if we start connection
- ACK flag – Boolean (1 = true, 0 = false in byte) 1-byte checks if we acknowledged a packet
- FIN flag – Boolean (1 = true, 0 = false in byte) 1-byte checks if we are closing connection
- DUP flag – made in early development was not used (hard to re-factor packets)
- Sequence number – refers to current byte we are sending
- Checksum – validation of payload, performed on both ends
- Acknowledgement number – acknowledging the last successfully received packet
- Source/destination IP/port – already in UDP header, was not used



```
//now we set our index's of our header based on size + position
static final int ACK_POSITION_IN_HEADER = 0;
static final int SYN_POSITION_IN_HEADER = 1;
static final int FIN_POSITION_IN_HEADER = 2;
static final int DUP_POSITION_IN_HEADER = 3;
static final int SEQ_POSITION_IN_HEADER = 4;
//since sequence number has size 4 (integer) checksum will be position 8 and so on
//and so on
static final int CHECKSUM_POSITION_IN_HEADER = 8;
static final int ACKNUM_POSITION_IN_HEADER = 12;
static final int DSTIP_POSITION_IN_HEADER = 16;
static final int SRCIP_POSITION_IN_HEADER = 20;
static final int SRCPRT_POSITION_IN_HEADER = 24;
static final int DSTPRT_POSITION_IN_HEADER = 28;
static final int PAYLOAD_POSITION_IN_HEADER = 32;
//a global byte buffer to convert between byte array and integer/primitive values
```

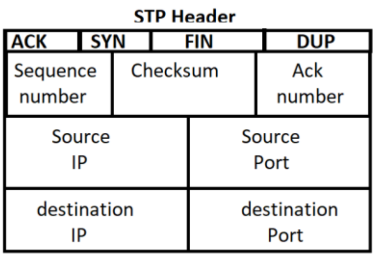


Figure 1 STP header and positioning of header

design trade-offs considered and made, possible improvements and extensions to the program and how I could realise them

some design trade-offs I made was modifying the receiver side buffer, and packet handling, instead of instantly writing a packet to the file if it is in order and non-corrupt, I instead stored each packet received into a list of writing packets. This list was sorted and contained no duplicates, so file writing was performed at the end of receiving rather than during receiving. This while it does use less CPU usage comes with a higher memory usage. Due to this as well, it allows for delayed ACK's where the window will not stop transmitting just because of out of order packets, instead it will keep putting files in the window if there is space, take for instance, if we send out packet 1 2 3 4, and receive ack for 2 3 4 and not 1, instead of halting till we receive ack for packet 1, we would make our window 1 5 6 7, as any packet inside the window will be guaranteed to be sent and maintain the packet order on the receiver side. This allows for a faster transmission, however as mentioned above this comes with a higher memory cost, and a larger MSS has a MUCH faster transmission (on the CSE machine, on my machine it is very fast). Something else differently I did to simulate cumulative ACK to an extent was I stored out of order packets in the buffer, and got my ACK number from the last packet in my list of correct packets. After this I would insert the buffer packets into the list as the list will maintain order. Something to note especially in testing, is that the larger number of packets to be transmitted (small MSS), this will have a huge effect due to the amount of memory used. Note despite how long it takes, it will send the file across by the end, and it will not be corrupted. If in testing it is taking too long, increasing MSS will fix this issue. Something I could have done to improve the overall send/receiver, is creating the packets whilst sending and writing to the pdf file as receiving, this would increase the overall memory usage, with the trade-off of a slower transmission depending on the machine, and higher CPU usage. If I were to do this I would have possibly attempted an implementation and as done with every section, through the use of debugging I would slowly work towards a finished solution. One extension I did make was creating the files, and logs in a separate directory in order to avoid naming conflictions, losing data and or losing previous log files. I was considering using a hash map for the packets, in order to maintain the packets, and allow for efficient search compared to array list, however the use of a hash-map increases the complexity of the code, and would require a large-scale refactoring of many aspects, so I opted to use a set implementation using an array list as a wrapper. Despite all these considerations, the program does work as intended, the time spent varies depending on the MSS and memory on the machine (lower memory means less available heap which will require a larger amount of garbage collection, to avoid this increasing MSS will drastically increase performance.

Segments of code that were borrowed

Most of the segments were borrowed from stack-exchange from learning from the answers, and performing my implementation from my understanding, this was only for the byte buffer operations to understand how to create packets and convert them to a readable form.
https://stackoverflow.com/questions/4841340/what-is-the-use-of-bytebuffer-in-java
https://stackoverflow.com/questions/6206336/java-nio-and-bytebuffer-problem
https://stackoverflow.com/questions/1936857/convert-integer-into-byte-array-java
https://stackoverflow.com/questions/7619058/convert-a-byte-array-to-integer-in-java-and-vice-versa
Also, another reference for writing bytes to a file output stream.
https://stackoverflow.com/questions/4350084/byte-to-file-in-java

Tests on output

(please note multi-threading was used in both sending/receiving acknowledgements, and this testing was done on my machine (32gb ram, intel i7 4790k processor), as it is hard to view log files on vlab, however I have tested this on vlab too and the results are very similar, however time might be different, but it works!)

(a) Run your protocol using pDrop = 0.1, MWS = 500 bytes, MSS = 100 bytes, seed = 100, gamma = 4, and pDuplicate, pCorrupt, pOrder, MaxOrder, pDelay, MaxDelay all set to 0. Transfer the file test0.pdf. Run an additional experiment with pdrop = 0.3, transferring the same file (test0.pdf). In your report, discuss the resulting packet sequences of both experiments indicating where dropping occurred.

On the first packet sequence, dropping occurred on packet 700 and 900 and as can be observed, they were later received as they are waiting in the window for re-transmission due timeout before they are re-transmitted.

For part 2 of this experiment

12 segments were dropped and these segments on the sender log were 500, 600, 800, 1100, 500 again, 1500, 1600, 1700, 1800, 1100 again, 2000, 2100, 2900, and 2100 again. These can be seen from the receiver log as arriving late as they are still in window waiting for timeout.

For experiment one there are 31 packets to send to the receiver, with a 1/10 probability to drop 2 packets dropped (albeit nearly back to back for part a) which shows working behaviour of pdrop, similarly for experiment 2 with a 3/10 probability to drop we dropped 12 segments which similarly shows working behaviour of pdrop. The file is received correctly in the directory created files under "test0.pdf". with no missing packets and maintaining the exact same file size.

(b) The timeout for STP is given by: TimeoutInterval = EstimatedRTT + gamma * DevRTT where gamma will be supplied to the program as an input argument. Set pdrop = 0.5, MWS = 500 bytes, MSS = 50 bytes, seed = 300, pdelay = 0.2, MaxDelay = 1000 and pDuplicate, pCorrupt, pOrder, MaxOrder all set to 0. Run three experiments with the following different gamma values: I. gamma = 2 ii. gamma = 4 iii. gamma = 6 and transfer the file test1.pdf using STP. Show a table that indicates how many STP packets were transmitted in total and how long the overall transfer took. Discuss the results

From the following test it was observed that the value gamma had very little to no effect on number of packets transmitted (they fell around the same amount from 19158 – 19532) this could be because of multi-threading, or the change in the timeout causing extra re-transmissions. However, with the same number of packets sent, the total time for transmission was increasing at a linear rate, when we added 2 to gamma, it took an extra 6 minutes, when we added 2 again it took an extra 5 minutes. This is because when we increase the timeout interval, we increase the time before we re-send the dropped packets, in turn causing a longer time of overall transfer. This can be seen in table 2, when we increase gamma, number of packets transmitted stays the same, but time increases.

Table 2.

Experiment b time taken, and number of packets sent

(see appendix for validation screenshots from summary)

gamma	*Total time for transmission* (seconds)	*Total number of packets transmitted* (number of packets)
2	1363 (22 minutes)	19158
4	1665 (28 minutes)	19532
6	1982 (33 minutes)	19286

(c) Use the following values and run STP to transfer test2.pdf. MWS=500bytes MSS=50 gamma=4 pDrop=0.1 pDuplicate=0.1 pCorrupt=0.1 pOrder=0.1 maxOrder=4 pDelay=0 maxDelay=0 seed=300 Has the file been successfully transferred? How long the overall transfer took? For this experiment, which of the factor (out of pDrop, pDuplicate, pCorrupt and pOrder) is the most critical contributing most in the overall transfer time? How have you determined this?

The file has successfully been transferred, the overall transfer took 1493 seconds (25 minutes). The most critical factor that contributes to the transfer time, would be pDrop. pDuplicate will send over a duplicate packet which ends up being ignored and having a very small effect on the transfer time, this is due to the implementation of my receiver, however if you observe the log files, it will simply send back a duplicate ACK instantly for duplicate packets. Porder will have almost no effect on the transmission time, as the receiver side handles ordering of the packet, and it will still send and maintain the window despite re-ordering. This can be seen in the log files, when a packet is re-ordered, we are still putting packets into the window and receiving acks for those packets, and when the re-orded packet finally is sent, we instantly receive an ack back, effectively pOrder has no effect on the window/receiver side even though it sends out of order packets. pCorrupt has a small effect on the total transfer time, as this will require instant re-transmission upon receiving a NAK for the corrupt segment, in turn requiring more transmissions through the PLD, however none of these have as much of an effect as pDrop. pDrop will cause packets to drop, which in turn will decrease timeout and cause more re-transmissions due the decrease in timeout, in some cases we get a window full of dropped packets, as all the sent packets have been sent, and what's remaining are packets that are dropped waiting to be re-transmitted. This causes a section in time where we are simply waiting to re-transmit. Upon observing the log files, there are sections where our window has all dropped packets, and is simply waiting on the timeout before it re-transmits. as seen also above we had a small file almost six times as small as this one, and with pDrop being 0.5, it had a longer transmission time than above (28 minutes for a 308kb file vs 1.6mb file). This in turns displays the impact of pDrop, as with all other error checks on, and with a larger file, and simply only modifying pDrop down to 0.1, we get a drastic change in transmission time.

Appendix

Receiver sequences for test (a)

snd/rcv	time	type	sequence	payload size	ack
rcv	1.0	S	0	0	0
snd	1.0	SA	0	0	1
rcv	1.0	A	1	0	1
rcv	1.0	D	0	100	0
snd	1.0	A	1	0	0
rcv	1.0	D	100	100	0
snd	1.0	A	1	0	100
rcv	1.0	D	200	100	0
snd	1.0	A	1	0	200
rcv	1.0	D	300	100	0
snd	1.0	A	1	0	300
rcv	1.0	D	400	100	0
snd	1.0	A	1	0	400
rcv	1.0	D	500	100	0
snd	1.0	A	1	0	500
rcv	1.0	D	600	100	0
snd	1.0	A	1	0	600
rcv	1.0	D	700	100	0
snd	1.0	A	1	0	700
rcv	1.0	D	900	100	0
snd	1.0	A	1	0	700
rcv	1.0	D	1100	100	0
snd	1.0	A	1	0	900
rcv	1.0	D	1200	100	0
snd	1.0	A	1	0	1200
rcv	1.0	D	1300	100	0
snd	1.0	A	1	0	1300
rcv	1.0	D	1400	100	0
snd	1.0	A	1	0	1400
rcv	1.0	D	1500	100	0
snd	1.0	A	1	0	1500
rcv	1.0	D	1600	100	0
snd	1.0	A	1	0	1600
rcv	1.0	D	1700	100	0
snd	1.0	A	1	0	1700
rcv	1.0	D	1800	100	0
snd	1.0	A	1	0	1800
rcv	1.0	D	1900	100	0
snd	1.0	A	1	0	1900
rcv	1.0	D	2000	100	0
snd	1.0	A	1	0	2000
rcv	1.0	D	2100	100	0
snd	1.0	A	1	0	2100
rcv	1.0	D	2200	100	0
snd	1.0	A	1	0	2200
rcv	1.0	D	2300	100	0
snd	1.0	A	1	0	2300
rcv	1.0	D	2400	100	0
snd	1.0	A	1	0	2400
rcv	1.0	D	2500	100	0
snd	1.0	A	1	0	2500
rcv	1.0	D	2600	100	0
snd	1.0	A	1	0	2600
rcv	1.0	D	2700	100	0
snd	1.0	A	1	0	2700
rcv	1.0	D	2800	100	0
snd	1.0	A	1	0	2800
rcv	1.0	D	2900	100	0
snd	1.0	A	1	0	2900
rcv	1.0	D	2928	28	0
snd	1.0	A	1	0	3000
rcv	1.0	D	728	100	0
snd	1.0	A	1	0	3000
snd/DA	1.0	A	1	0	3000
rcv	1.0	D	928	100	0
snd	1.0	A	1	0	3000
snd/DA	1.0	A	1	0	3000
snd/DA	1.0	A	1	0	3000
rcv	2.0	F	3028	0	1
snd	2.0	A	1	0	3029
snd	2.0	F	1	0	3029
rcv	2.0	A	3029	0	2

Amount of data received (bytes)	3328
Total Segments Received	39
Data segments received	35
Data segments with Bit Errors	0
Duplicate data segments received	3
Duplicate ACKs sent	3

Sender sequence for test (a)

snd/rcv	time	type	sequence	payload size	ack
snd	0.0	S	0	0	0
snd	1.0	S	0	0	0
snd	3.0	S	0	0	0
rcv	3.0	SA	0	0	1
snd	3.0	A	1	0	1
snd	3.0	D	0	100	1
snd	3.0	D	100	100	1
snd	3.0	D	200	100	1
snd	3.0	D	300	100	1
snd	3.0	D	400	100	1
rcv/DA	3.0	A	1	0	0
snd	3.0	D	500	100	1
rcv	3.0	A	1	0	100
snd	3.0	D	600	100	1
rcv	3.0	A	1	0	200
snd	3.0	D	700	100	1
rcv	3.0	A	1	0	300
drop	3.0	D	800	100	1
rcv	3.0	A	1	0	400
snd	3.0	D	900	100	1
rcv	3.0	A	1	0	500
drop	3.0	D	1000	100	1
rcv	3.0	A	1	0	600
snd	3.0	D	1100	100	1
rcv	3.0	A	1	0	700
snd	3.0	D	1200	100	1
rcv	3.0	A	1	0	900
snd	3.0	D	1300	100	1
rcv	3.0	A	1	0	1200
snd	3.0	D	1400	100	1
rcv	3.0	A	1	0	1300
snd	3.0	D	1500	100	1
rcv	3.0	A	1	0	1400
snd	3.0	D	1600	100	1
rcv	3.0	A	1	0	1500
snd	3.0	D	1700	100	1
rcv	3.0	A	1	0	1600
snd	3.0	D	1800	100	1
rcv	3.0	A	1	0	1700
snd	3.0	D	1900	100	1
rcv	3.0	A	1	0	1800
snd	3.0	D	2000	100	1
rcv	3.0	A	1	0	1900
snd	3.0	D	2100	100	1
rcv	3.0	A	1	0	2000
snd	3.0	D	2200	100	1
rcv	3.0	A	1	0	2100
snd	3.0	D	2300	100	1
rcv	3.0	A	1	0	2200
snd	3.0	D	2400	100	1
rcv	3.0	A	1	0	2300
snd	3.0	D	2500	100	1
rcv	3.0	A	1	0	2400
snd	3.0	D	2600	100	1
rcv	3.0	A	1	0	2500
snd	3.0	D	2700	100	1
rcv	3.0	A	1	0	2600
snd	3.0	D	2800	100	1
rcv	3.0	A	1	0	2700
snd	3.0	D	2900	100	1
rcv	3.0	A	1	0	2800
snd	3.0	D	3000	28	1
rcv	3.0	A	1	0	2900
rcv	3.0	A	1	0	3000
snd	3.0	D	800	100	1
snd/RXT	3.0	D	800	100	1
snd	3.0	D	800	100	1
snd/RXT	3.0	D	800	100	1
rcv	3.0	A	1	0	800
snd	3.0	D	1000	100	1
snd/RXT	3.0	D	1000	100	1
snd	3.0	D	1000	100	1
snd/RXT	3.0	D	1000	100	1
rcv	3.0	A	1	0	1000
snd	3.0	D	1100	100	1
snd/RXT	3.0	D	1100	100	1
rcv	3.0	A	1	0	1100
snd	4.0	F	3028	0	1
rcv	4.0	A	1	0	3029
rcv	4.0	F	1	0	3029
snd	4.0	A	3029	0	2

Size of the file (in Bytes)	3028
Segments transmitted (including drop & RXT)	42
Number of Segments handled by PLD	36
Number of Segments dropped	2
Number of Segments Corrupted	0
Number of Segments Re-ordered	0
Number of Segments Duplicated	0
Number of Segments Delayed	0
Number of Retransmissions due to TIMEOUT	5
Number of FAST RETRANSMISSION	0
Number of DUP ACKS received	3

Receiver sequence for 2nd part of (a)

snd/rcv	time	type	sequence	payload size	ack
rcv	1.0	S	0	0	0
snd	1.0	SA	0	0	1
rcv	1.0	A	1	0	1
rcv	1.0	D	0	100	1
snd	1.0	A	1	0	0
rcv	1.0	D	100	100	1
snd	1.0	A	1	0	100
rcv	1.0	D	200	100	1
snd	1.0	A	1	0	200
rcv	1.0	D	300	100	1
snd	1.0	A	1	0	300
rcv	1.0	D	400	100	1
snd	1.0	A	1	0	400
rcv	1.0	D	700	100	1
snd	1.0	A	1	0	400
rcv	1.0	D	900	100	1
snd	1.0	A	1	0	700
rcv	1.0	D	1000	100	1
snd	1.0	A	1	0	1000
rcv	2.0	D	500	100	1
snd	2.0	A	1	0	1000
snd/DA	3.0	A	1	0	1000
rcv	3.0	D	1200	100	1
snd	3.0	A	1	0	1000
rcv	3.0	D	600	100	1
snd	3.0	A	1	0	1200
rcv	3.0	D	1300	100	1
snd	3.0	A	1	0	1300
rcv	3.0	D	1400	100	1
snd	3.0	A	1	0	1400
snd/DA	4.0	A	1	0	1400
rcv	4.0	D	800	100	1
snd	4.0	A	1	0	1400
snd/DA	4.0	A	1	0	1400
snd/DA	4.0	A	1	0	1400
rcv	4.0	D	1100	100	1
snd	4.0	A	1	0	1400
snd/DA	5.0	A	1	0	1400
rcv	5.0	D	1900	100	1
snd	5.0	A	1	0	1400
rcv	5.0	D	1500	100	1
snd	5.0	A	1	0	1900
snd/DA	5.0	A	1	0	1900
rcv	5.0	D	1600	100	1
snd	5.0	A	1	0	1900
snd/DA	5.0	A	1	0	1900
rcv	5.0	D	2200	100	1
snd	5.0	A	1	0	1900
rcv	5.0	D	1700	100	1
snd	5.0	A	1	0	2200
rcv	5.0	D	2300	100	1
snd	5.0	A	1	0	2300
rcv	5.0	D	2400	100	1
snd	5.0	A	1	0	2400
rcv	5.0	D	2500	100	1
snd	5.0	A	1	0	2500
rcv	5.0	D	2600	100	1
snd	5.0	A	1	0	2600
rcv	5.0	D	2700	100	1
snd	5.0	A	1	0	2700
rcv	5.0	D	2800	100	1
snd	5.0	A	1	0	2800
snd/DA	5.0	A	1	0	2800
rcv	5.0	D	2928	28	1
snd	5.0	A	1	0	2800
rcv	5.0	D	1728	100	1
snd	5.0	A	1	0	3000
snd/DA	5.0	A	1	0	3000
rcv	5.0	D	1928	100	1
snd	5.0	A	1	0	3000
snd/DA	5.0	A	1	0	3000
rcv	5.0	D	2028	100	1
snd	5.0	A	1	0	3000
snd/DA	5.0	A	1	0	3000
rcv	6.0	D	2828	100	1
snd	6.0	A	1	0	3000
snd/DA	6.0	A	1	0	3000
rcv	6.0	F	3028	0	1
snd	6.0	A	1	0	3029
snd	6.0	F	1	0	3029
rcv	6.0	A	3029	0	2

Amount of data received (bytes)	4228
Total Segments Received	48
Data segments received	44
Data segments with Bit Errors	0
Duplicate data segments received	12
Duplicate ACKs sent	12

Sender sequence for 2nd part of (a)

snd/rcv	time	type	sequence	payload size	ack
snd	0.0	S	0	0	0
snd	1.0	S	0	0	0
snd	3.0	S	0	0	0
rcv	3.0	SA	0	0	1
snd	3.0	A	1	0	1
snd	3.0	D	0	100	1
snd	3.0	D	100	100	1
snd	3.0	D	200	100	1
snd	3.0	D	300	100	1
snd	3.0	D	400	100	1
rcv/DA	3.0	A	1	0	0
drop	3.0	D	500	100	1
rcv	3.0	A	1	0	100
drop	3.0	D	600	100	1
rcv	3.0	A	1	0	200
snd	3.0	D	700	100	1
rcv	3.0	A	1	0	300
drop	3.0	D	800	100	1
rcv	3.0	A	1	0	400
snd	3.0	D	900	100	1
rcv	3.0	A	1	0	700
snd	3.0	D	1000	100	1
rcv	3.0	A	1	0	1000
drop	3.0	D	1100	100	1
snd	3.0	D	500	100	1
snd/RXT	3.0	D	500	100	1
drop	4.0	D	500	100	1
snd/RXT	4.0	D	500	100	1
snd	4.0	D	500	100	1
snd/RXT	4.0	D	500	100	1
rcv	4.0	A	1	0	500
snd	4.0	D	1200	100	1
snd	5.0	D	600	100	1
snd/RXT	5.0	D	600	100	1
rcv	5.0	A	1	0	1200
snd	5.0	D	1300	100	1
rcv	5.0	A	1	0	1300
snd	5.0	D	1400	100	1
rcv	5.0	A	1	0	1400
drop	5.0	D	1500	100	1
snd	5.0	D	600	100	1
snd/RXT	5.0	D	600	100	1
rcv	5.0	A	1	0	600
drop	5.0	D	1600	100	1
snd	5.0	D	800	100	1
snd/RXT	5.0	D	800	100	1
snd	6.0	D	800	100	1
snd/RXT	6.0	D	800	100	1
rcv	6.0	A	1	0	800
drop	6.0	D	1700	100	1
snd	6.0	D	900	100	1
snd/RXT	6.0	D	900	100	1
rcv	6.0	A	1	0	900
drop	6.0	D	1800	100	1
snd	6.0	D	1100	100	1
snd/RXT	6.0	D	1100	100	1
drop	6.0	D	1100	100	1
snd/RXT	6.0	D	1100	100	1
snd	6.0	D	1100	100	1
snd/RXT	6.0	D	1100	100	1
rcv	6.0	A	1	0	1100
snd	6.0	D	1900	100	1
snd	6.0	D	1500	100	1
snd/RXT	6.0	D	1500	100	1
rcv	6.0	A	1	0	1900
drop	6.0	D	2000	100	1
snd	6.0	D	1500	100	1
snd/RXT	6.0	D	1500	100	1
rcv	6.0	A	1	0	1500
snd	6.0	D	1600	100	1
drop	6.0	D	2100	100	1
snd/RXT	6.0	D	1600	100	1
snd	7.0	D	1600	100	1
snd/RXT	7.0	D	1600	100	1
rcv	7.0	A	1	0	1600
snd	7.0	D	2200	100	1
snd	7.0	D	1700	100	1
snd/RXT	7.0	D	1700	100	1
rcv	7.0	A	1	0	2200
snd	7.0	D	2300	100	1
rcv	7.0	A	1	0	2300
snd	7.0	D	2400	100	1
rcv	7.0	A	1	0	2400
snd	7.0	D	2500	100	1
rcv	7.0	A	1	0	2500
snd	7.0	D	2600	100	1
rcv	7.0	A	1	0	2600
snd	7.0	D	2700	100	1
rcv	7.0	A	1	0	2700
snd	7.0	D	2800	100	1
rcv	7.0	A	1	0	2800
snd	7.0	D	1700	100	1
drop	7.0	D	2900	100	1
snd/RXT	7.0	D	1700	100	1
rcv	7.0	A	1	0	1700
snd	7.0	D	3000	28	1
snd	7.0	D	1800	100	1
snd/RXT	7.0	D	1800	100	1
rcv	7.0	A	1	0	3000
snd	7.0	D	1800	100	1
snd/RXT	7.0	D	1800	100	1
rcv	7.0	A	1	0	1800
snd	7.0	D	2000	100	1
snd/RXT	7.0	D	2000	100	1
drop	7.0	D	2000	100	1
snd/RXT	7.0	D	2000	100	1
snd	7.0	D	2000	100	1
snd/RXT	7.0	D	2000	100	1
rcv	7.0	A	1	0	2000
drop	7.0	D	2100	100	1
snd/RXT	7.0	D	2100	100	1
snd	7.0	D	2100	100	1
snd/RXT	7.0	D	2100	100	1
drop	7.0	D	2100	100	1
snd/RXT	7.0	D	2100	100	1
snd	7.0	D	2100	100	1
snd/RXT	7.0	D	2100	100	1
rcv	7.0	A	1	0	2100
snd	7.0	D	2900	100	1
snd/RXT	7.0	D	2900	100	1
drop	7.0	D	2900	100	1
snd/RXT	7.0	D	2900	100	1
snd	7.0	D	2900	100	1
snd/RXT	7.0	D	2900	100	1
rcv	7.0	A	1	0	2900
snd	8.0	F	3028	0	1
rcv	8.0	A	1	0	3029
rcv	8.0	F	1	0	3029
snd	8.0	A	3029	0	2

Size of the file (in Bytes)	3028
Segments transmitted (including drop & RXT)	66
Number of Segments handled by PLD	60
Number of Segments dropped	17
Number of Segments Corrupted	0
Number of Segments Re-ordered	0
Number of Segments Duplicated	0
Number of Segments Delayed	0
Number of Retransmissions due to TIMEOUT	29
Number of FAST RETRANSMISSION	0
Number of DUP ACKS received	3

Summary for test 2 (b) in experiment for

Gamma = 2

Below in Figure 2 shows a summary of results with gamma = 2

```
snd               1363.0     A          308204          0               2
--------------------------------------------------------------------------
Size of the file (in Bytes)                         308203
Segments transmitted (including drop & RXT)         19518
Number of Segments handled by PLD                   19512
Number of Segments dropped                          9745
Number of Segments Corrupted                        0
Number of Segments Re-ordered                       0
Number of Segments Duplicated                       0
Number of Segments Delayed                          1988
Number of Retransmissions due to TIMEOUT            13007
Number of FAST RETRANSMISSION                       41
Number of  DUP ACKS received                        805
--------------------------------------------------------------------------
```

Figure 2 summary for test with gamma = 2

Gamma = 4

Below in Figure 3 shows a summary of results when gamma = 4

```
snd               1665.0     A          308204          0               2
--------------------------------------------------------------------------
Size of the file (in Bytes)                         308203
Segments transmitted (including drop & RXT)         19532
Number of Segments handled by PLD                   19526
Number of Segments dropped                          9764
Number of Segments Corrupted                        0
Number of Segments Re-ordered                       0
Number of Segments Duplicated                       0
Number of Segments Delayed                          1962
Number of Retransmissions due to TIMEOUT            13026
Number of FAST RETRANSMISSION                       34
Number of  DUP ACKS received                        739
--------------------------------------------------------------------------
```

Figure 3 summary for test with gamma = 4

Gamma = 6

Below in Figure 4 shows a summary of results when gamma = 4

```
snd               1982.0     A          308204          0               2
--------------------------------------------------------------------------
Size of the file (in Bytes)                         308203
Segments transmitted (including drop & RXT)         19286
Number of Segments handled by PLD                   19281
Number of Segments dropped                          9682
Number of Segments Corrupted                        0
Number of Segments Re-ordered                       0
Number of Segments Duplicated                       0
Number of Segments Delayed                          1923
Number of Retransmissions due to TIMEOUT            12881
Number of FAST RETRANSMISSION                       24
Number of  DUP ACKS received                        645
--------------------------------------------------------------------------
```

Figure 4 summary for test with gamma = 6

First and Last 20 entries plus overhead for the sender (part c)

Figure 5 shows the first 20 entries of the sender including the 3-way handshake connection (SYN SYNACK ACK), an extra 2 syns was sent because I started sender first and started receiver a second later

```
snd/rcv        time     type    sequence       payload size   ack
-------------------------------------------------------------------------
snd            0.0      S       0              0              0
snd            2.0      S       0              0              0
snd            3.0      S       0              0              0
rcv            3.0      SA      0              0              1
snd            3.0      A       1              0              1
snd            3.0      D       0              50             1
snd            3.0      D       50             50             1
snd            3.0      D       100            50             1
drop           3.0      D       150            50             1
snd            3.0      D       200            50             1
drop           3.0      D       250            50             1
rcv/DA         3.0      A       1              0              0
snd            3.0      D       300            50             1
snd            3.0      D       350            50             1
rcv            3.0      A       1              0              50
drop           3.0      D       400            50             1
snd            3.0      D       450            50             1
rcv            3.0      A       1              0              100
snd            3.0      D       500            50             1
snd            3.0      D       550            50             1
snd            3.0      D       600            50             1
drop           3.0      D       650            50             1
rcv            3.0      A       1              0              200
drop           3.0      D       700            50             1
```

Figure 5 first 20 entries of the transmissions and handshake for the sender

Figure 6 shows the final 20 entries of the sender including the 4-way closure (FIN ACK FIN ACK) and the overall summary

```
rcv            1492.0     A       1              0              1604700
snd            1492.0     D       1604800        50             1
snd/RXT        1492.0     D       1604800        50             1
snd            1493.0     D       1604800        50             1
snd/RXT        1493.0     D       1604800        50             1
rcv            1493.0     A       1              0              1604800
snd            1493.0     D       1605150        50             1
snd/RXT        1493.0     D       1605150        50             1
rcv            1493.0     A       1              0              1605150
rcv            1493.0     A       1              0              1605150
snd            1493.0     D       1605400        50             1
snd/RXT        1493.0     D       1605400        50             1
snd/corr       1493.0     D       1605400        50             1
snd            1493.0     D       1605400        50             1
snd/RXT        1493.0     D       1605400        50             1
drop           1493.0     D       1605400        50             1
snd/RXT        1493.0     D       1605400        50             1
snd            1493.0     D       1605400        50             1
snd/RXT        1493.0     D       1605400        50             1
rcv            1493.0     A       1              0              1605400
snd            1493.0     F       1605585        0              1
rcv            1493.0     A       1              0              1605586
rcv            1493.0     F       1              0              1605586
snd            1493.0     A       1605586        0              2
---------------------------------------------------------------------------
Size of the file (in Bytes)                        1605585
Segments transmitted (including drop & RXT)        63853
Number of Segments handled by PLD                  61291
Number of Segments dropped                         5654
Number of Segments Corrupted                       4549
Number of Segments Re-ordered                      2578
Number of Segments Duplicated                      5091
Number of Segments Delayed                         0
Number of Retransmissions due to TIMEOUT           16114
Number of FAST RETRANSMISSION                      624
Number of  DUP ACKS received                       8873
---------------------------------------------------------------------------
```

Figure 6 the last 20 entries + statistics + closure for sender

First and Last 20 entries plus overhead for the receiver (part c)

Figure 7 shows the first 20 entries of the receiver including the 3-way handshake connection (SYN SYNACK ACK).

snd/rcv	time	type	sequence	payload size	ack
rcv	1.0	S	0	0	0
snd	1.0	SA	0	0	1
rcv	1.0	A	1	0	1
rcv	1.0	D	0	50	1
snd	1.0	A	1	0	0
rcv	1.0	D	50	50	1
snd	1.0	A	1	0	50
rcv	1.0	D	100	50	1
snd	1.0	A	1	0	100
rcv	1.0	D	200	50	1
snd	1.0	A	1	0	100
rcv	1.0	D	300	50	1
snd	1.0	A	1	0	200
rcv	1.0	D	350	50	1
snd	1.0	A	1	0	350
rcv	1.0	D	450	50	1
snd	1.0	A	1	0	350
rcv	1.0	D	500	50	1
snd	1.0	A	1	0	500
rcv	1.0	D	550	50	1
snd	1.0	A	1	0	550
rcv	1.0	D	600	50	1
snd	1.0	A	1	0	600

Figure 7 first 20 entries of the transmissions and handshake for the receiver

Figure 8 shows the final 20 entries of the receiver including the 4-way closure (FIN ACK FIN ACK) and the overall summary

snd	1490.0	A	1	0	1605450
snd/DA	1490.0	A	1	0	1605450
rcv	1490.0	D	1605500	50	1
snd	1490.0	A	1	0	1605500
rcv	1490.0	D	1605535	35	1
snd	1490.0	A	1	0	1605550
snd/DA	1490.0	A	1	0	1605550
snd/DA	1490.0	A	1	0	1605550
rcv	1490.0	D	1604635	50	1
snd	1490.0	A	1	0	1605550
snd/DA	1490.0	A	1	0	1605550
snd/DA	1490.0	A	1	0	1605550
rcv	1490.0	D	1604785	50	1
snd	1490.0	A	1	0	1605550
snd/DA	1490.0	A	1	0	1605550
snd/DA	1491.0	A	1	0	1605550
rcv	1491.0	D	1605385	50	1
snd	1491.0	A	1	0	1605550
snd/DA	1491.0	A	1	0	1605550
snd/DA	1491.0	A	1	0	1605550
rcv	1491.0	F	1605585	0	1
snd	1491.0	A	1	0	1605586
snd	1491.0	F	1	0	1605586
rcv	1491.0	A	1605586	0	2

Amount of data received (bytes)	2780735
Total Segments Received	55620
Data segments received	55616
Data segments with Bit Errors	4544
Duplicate data segments received	20282
Duplicate ACKs sent	13030

Figure 8 the last 20 entries + statistics + closure for receiver