TVNARGUN and HRAMACH2

Project #2 Report - Point-to-Multipoint File Transfer Protocol (P2MP-FTP)

Objective
To implement a point-to-multipoint reliable data transfer protocol using the Stop-and-Wait automatic repeat request (ARQ) scheme over UDP. And carry our number of experiments to evaluate the impact of number of peers, MSS size and packet loss on the performance of data transfer.

Point-to-Multipoint File Transfer Protocol (P2MP-FTP)
The protocol is designed to transfer data reliably from one sender to multiple receivers. The protocol we have designed provides a simple service to transfer a files from one host to multiple receivers. The protocol uses UDP to send packet from host to destination. And to achieve reliable transfer the protocol uses stop-and-wait ARQ scheme. Using the unreliable UDP
protocol allows us to implement a “transport layer” service such as reliable data transfer in user space.

Client-Server Architecture of P2MP-FTP
The client-server architecture consists of a P2MP-FTP client that will act as the sender and that connects to a set of P2MP-FTP servers that act as receivers. All data transfer occurs from client to server side and only acknowledgements travel from receivers to senders.

The P2MP-FTP Client (Sender)
The sender implements the P2MP-FTP client in a reliable fashion and carries out the data transfer. When the client starts, it reads a file provided over command line argument and the contents of the file are read in a set of bytes. And the main functions calls send_udp_packet() function to transfer the data to receivers in a reliable fashion. The client also implements stop-and-wait protocol, this logic is part of the main() function in our sender code, which takes care of buffering the data read from the file while ensuring the sent data is received correctly at the receiver side.

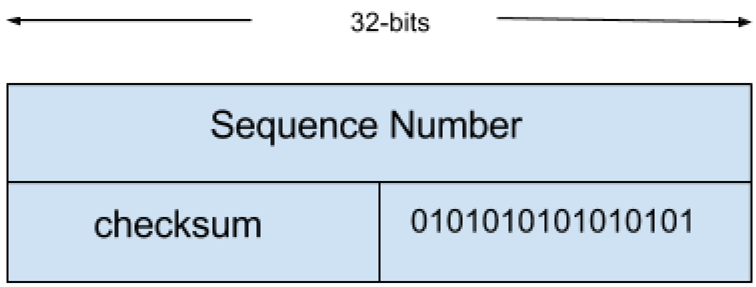
The client also reads the value of MSS from the command line. The stop-and-wait protocol buffers the data received from fread() until it has received at least 1 MSS worth of bytes. And once these bytes are available, it forms the header segment. Since we add a header at user-space, the MSS contains (MSS - user Header) bytes of file data. And the protocol ensures that all the packets sent contain exactly 1 MSS bytes of data, except possibly the last packet.

The client transmits each segment separately to each receiver and waits until it has received ACK from all the receivers before it proceeds to transmit the next segment. This mechanism is achieved through list_for_each(), going over all the receivers, passed through command line argument, for each segment. Every time a segment is transmitted, the sender sets a timeout counter. If counter expires before ACK is received, then the sender re-transmits the segment to only those receivers from which it has not received an ack yet. This process runs until all the ACKs have been received.

The header segment of the sender contains these three fields plus Payload:

1. a 32-bit sequence number (packet_sequence_number variable)
2. a 16-bit checksum of the data part, computed in the same way as the UDP checksum - function generate_checksum() takes care of this
3. a 16-bit field that has the value 0101010101010101, indicating that this is a data packet

Hence the header will occupy 8 bytes in the MSS sent to UDP layer,



Header Format - Sender

The sequence number is initialized to 0 in the beginning and increments by one for each segment sent. The function send_udp_packet() transmits the bytes to each receiver and after time-out retransmits the bytes to receivers who have not yet replied with an ACK.

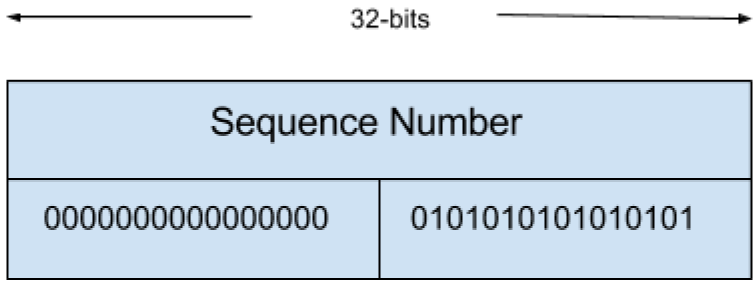
The P2MP-FTP Server (Receiver)

The server listens on the well-known port 7735. It implements the receive side of the Stop-and-Wait protocol. Specifically, when it receives a data packet, it computes the checksum and checks whether it is in-sequence, and if so, it sends an ACK segment (using UDP) to the client; it then writes the received data into a file whose name is provided in the command line. If the packet received is out-of-sequence, an ACK for the last received in-sequence packet is sent, if the checksum is incorrect, the receiver does nothing.

The ACK segment consists of three fields and no data:

1. a 32-bit sequence number that is being ACKed
2. a 16-bit field that is all 0's
3. a 16-bit field that has the value 1010101010101010, indicating that this is an ACK packet.

The receiver side header also takes up 8-bytes:



Header Format - Receiver

Generating Errors

To simulate the practical network conditions where the sender might fail to receive a packet due to packet drops in the network, we implement a mechanism to generate packet loss errors.

We have implemented a probabilistic loss service at the server. We read a probability value P, between 0 and 1, from the command line. When a data segment is received before generating the ACK, the server generates a random number r between 0 and 1. If r is greater than P, the receiver will not send an ACK for that packet and drop the received packet. The random number is generated using rand() function.

*Raw data from all the tests is shared at the end of the report

Task 1: Effect of Receiver Set Size n

For this first task, we set the MSS to 500 bytes and the loss probability p = 0.05. We ran the P2MP-FTP protocol to transfer a file of 1MB size, and vary the number of receivers n = 1, 2, 3, 4, 5. For each value of n, file transfer was ran 5 times, time of the data transfer (i.e., delay) was averaged over the five transmissions to arrive at the average delay. The below table lists the values generated by the test:

Receivers	**MSS**	**Error P**	**Delay - Average (sec)**	**Delay - Average (min)**
1	500	0.05	119.298	1.988
2	500	0.05	188.122	3.135
3	500	0.05	286.053	4.768
4	500	0.05	357.244	5.954
5	500	0.05	444.521	7.409

Table 1.1 - Receiver Set Test Data

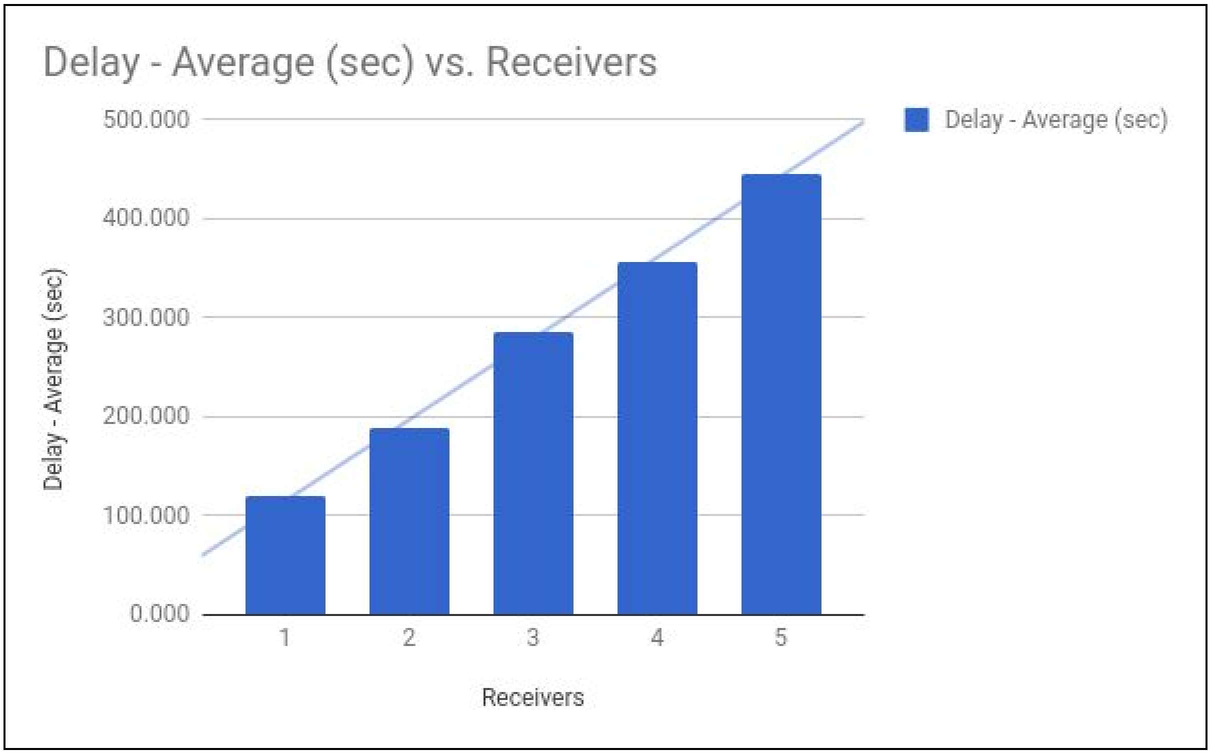


Fig 1.1 - Average Delay vs. Number of Receivers Graph

Task 1: Observation and Conclusion

Please refer the Fig.1.1 for the plot of the average delay against n. As we can see from the graph, the time required to transfer the file increases almost linearly with increase in the number of receivers for a given value of MSS and P. With each additional receiver, the sender has will incur an additional RTT and the probability of retransmission also increases by a factor of 0.05 (Independent events).

Task 2: Effect of MSS

In this experiment, the number of receivers was fixed at n = 3 and the loss probability at p = 0.05. We ran the P2MP-FTP protocol to transfer a 1Mb file, and the MSS value was varied from 100 bytes to 1000 bytes in increments of 100 bytes. For each value of MSS, file transfer was repeated 5 times, and the average delay over five transactions was calculated. The below table lists the values generated during the task,

Receivers	**MSS**	**Error P**	**Delay - Average (sec)**	**Delay - Average (min)**
3	100	0.05	1644.260	27.404
3	200	0.05	988.532	16.476
3	300	0.05	393.393	6.557
3	400	0.05	307.073	5.118

3	500	0.05	254.221	4.237
3	600	0.05	214.718	3.579
3	700	0.05	165.818	2.764
3	800	0.05	129.769	2.163
3	900	0.05	241.260	4.021
3	1000	0.05	209.603	3.493

Table 2.1 - MSS value set test data

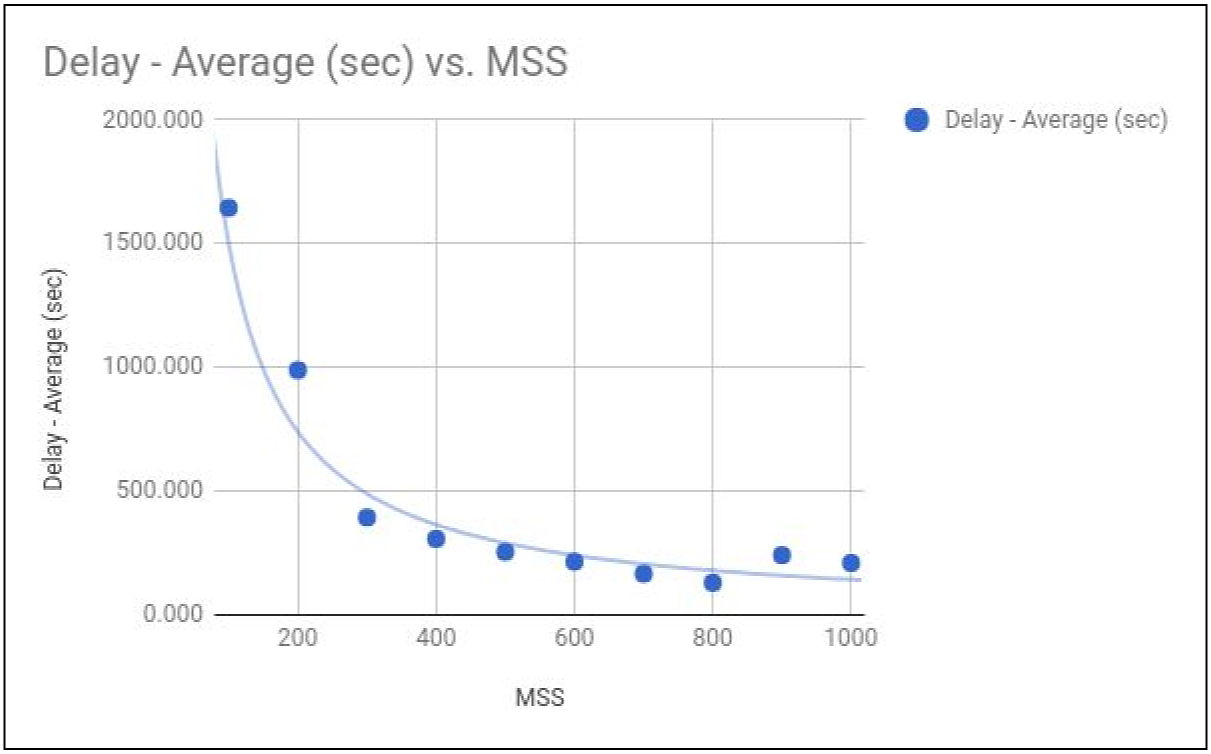


Fig 2.1 - Average Delay vs. increasing MSS value Graph

Task 2: Observation and Conclusion

Please refer the Fig 2.1 for the plot of the average delay against the MSS value. The average delay drops dramatically, almost in thirds, with each increase in MSS value. The second value of MSS = 200 , is double its previous value of 100 and we see a ~40% drop in file transfer delay. But as the MSS value increases, we see marginal improvements in the average delay time, this behavior can be explained by:

- When MSS is lower, we need a large number of segments to transfer a file of same size, this results in increased propagation delay and processing delay and as well as the delay in receiving ACK.
- Since Loss Probability, P, is a function of number of packets, we experience a large number of packet loss with lower MSS values. This adds additional delay due to Timeouts and retransmissions

We can conclude that the gains seen in faster file transfer times, starts to saturate as MSS value increases beyond 800 bytes. With a fixed packet loss, delay and given network conditions, we cannot continuously increase the MSS value to speed up file data transmissions. And this explains the shape of the curve in the graph, and yes, this trend is expected.

Task 3: Effect of Loss Probability, P

For this task, we set the MSS to 500 bytes and the number of receivers are fixed at n = 3. We tested the P2MP-FTP protocol transfer using the same 1Mb file, and varied the loss probability from p = 0.01 to p = 0.10 in increments of 0.01. For each value of p, file transmission was repeated 5 times, and computed the average delay over the five transfers. The results are tabulated below:

Receivers	**MSS**	**Error P**	**Delay - Average (sec)**	**Delay - Average (min)**
3	500	0.01	97.257	1.621
3	500	0.02	101.791	1.697
3	500	0.03	157.926	2.632
3	500	0.04	212.647	3.544
3	500	0.05	254.221	4.237
3	500	0.06	298.151	4.969
3	500	0.07	348.898	5.815
3	500	0.08	438.206	7.303
3	500	0.09	437.573	7.293
3	500	0.1	486.380	8.106

Table 3.1 - Loss Probability set test data

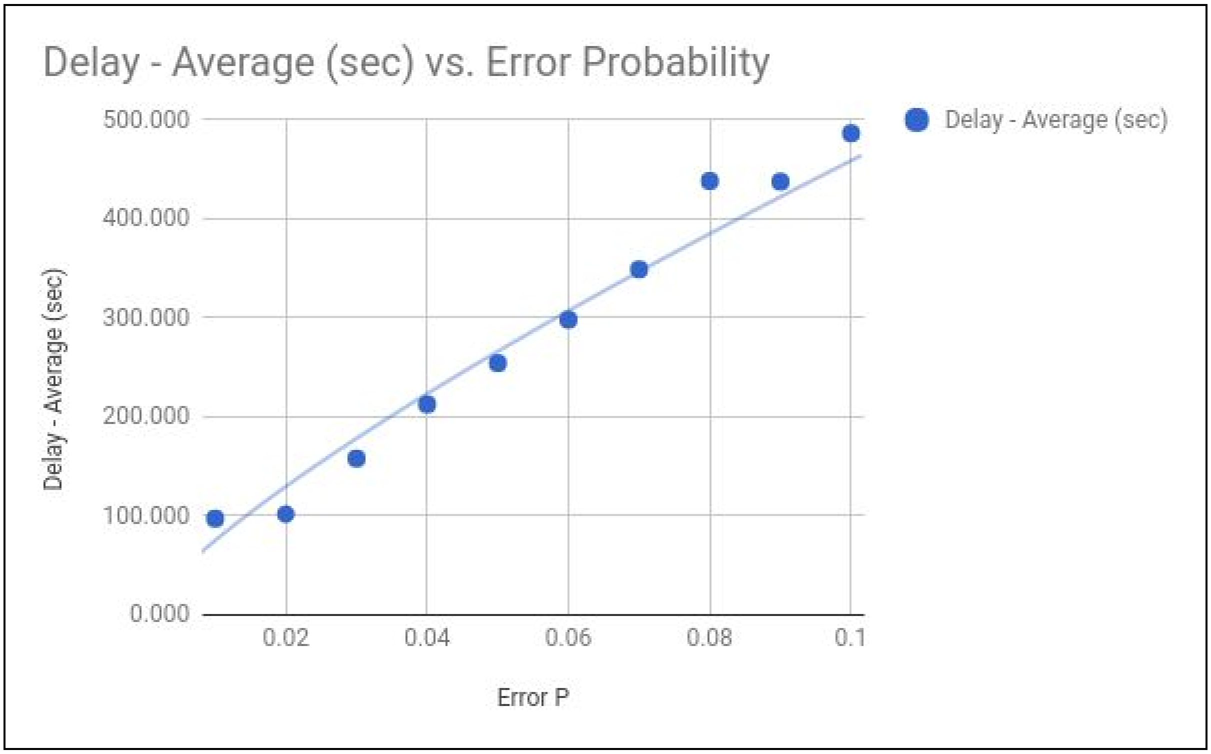


Fig 3.1 - Average Delay vs. increasing Loss Probability Graph

Task 3: Observation and Conclusion

Please refer the Fig 3.1 for the plot of the average delay against increasing loss probability. From the graph we can clearly see a linear increase in delay with increase in loss probability. We have 3 receivers that are dropping packets independently of each other. We can determine the combined probability by using general multiplication rule.

By design, the sender will not proceed unless and until everyone in the receiver set has acknowledged the reception of a segment. This mechanism will keep the other receivers waiting even if a single receiver drops a segment. Due to this behavior, the file transfer delay increases linearly with each step increase in loss probability.

Command Line Arguments to run server and client

The P2MP-FTP server is invoked as follows:

./p2mp <port#> <mss> <filename> <P> <server-ip>

Example: ./p2mp 7735 500 b.txt 0.05 10.139.60.228

Where,

Port# - is the port number to which the server is listening

mss - is the value of the mss bytes,

filename - is the name of the destination file to which the received data will be saved,

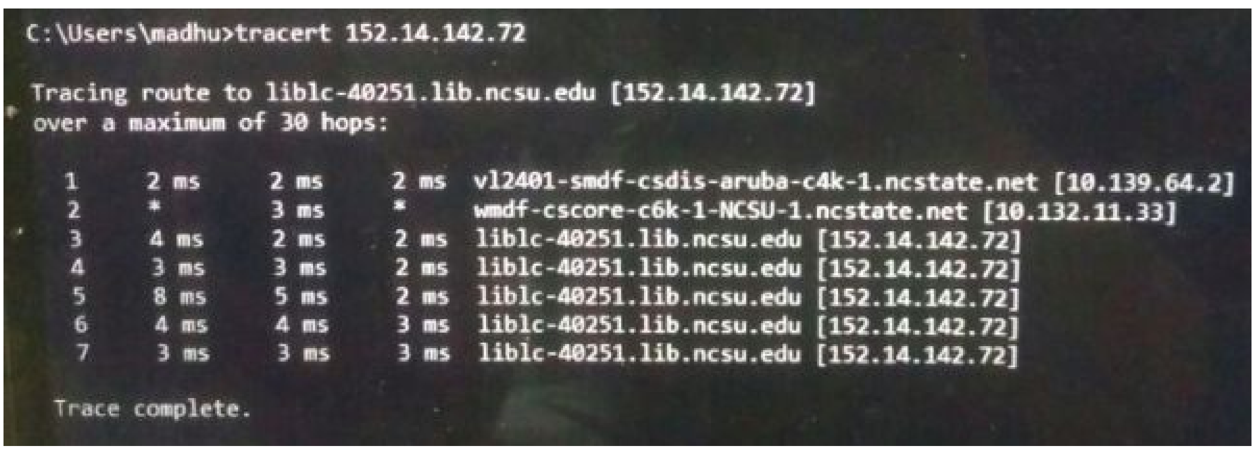
P - packet loss probability and

Server-ip - IP on which the server is running
For this project, this port number is always 7735.

The P2MP-FTP client is invoked as follows:
./sender <port#> <mss> <filename> <server-1 server-2 …. server-n>
Example: ./sender 7735 500 socket_client.c 10.139.60.228 152.14.142.107 152.14.142.72

Where,
Port# - is the port number to which the server is listening
mss - is the mss byte length
Filename - name of the file to be transferred
Server-i - is the ith server to which the file has to be transferred
For this project, this port number is always 7735.

Test Environment
The test was ran over NCSU campus network. We borrowed 5 MACs from the library and ran the receivers (P2MP-FTP server) on them. And the sender (P2MP-FTP client) was ran on our laptop which was connected to NCSU wifi. The route from sender to receiver had multiple-hops as shown below:



```
C:\Users\madhu>tracert 152.14.142.72

Tracing route to liblc-40251.lib.ncsu.edu [152.14.142.72]
over a maximum of 30 hops:

  1     2 ms     2 ms     2 ms  vl2401-smdf-csdis-aruba-c4k-1.ncstate.net [10.139.64.2]
  2     *        3 ms     *     wmdf-cscore-c6k-1-NCSU-1.ncstate.net [10.132.11.33]
  3     4 ms     2 ms     2 ms  liblc-40251.lib.ncsu.edu [152.14.142.72]
  4     3 ms     3 ms     2 ms  liblc-40251.lib.ncsu.edu [152.14.142.72]
  5     8 ms     5 ms     2 ms  liblc-40251.lib.ncsu.edu [152.14.142.72]
  6     4 ms     4 ms     3 ms  liblc-40251.lib.ncsu.edu [152.14.142.72]
  7     3 ms     3 ms     3 ms  liblc-40251.lib.ncsu.edu [152.14.142.72]

Trace complete.
```

FIg.4 Traceroute from Sender to Receiver

Based on this output and the ping from sender to receiver which was taking less than 15 ms. We set the timeout counter for 75 ms. This meant that the sender will wait for a time of 75 ms before sending any retransmissions.

Output
The snapshots below show a sample of the sender and client output.

```
tushar@server:~/ncsu/ip/project/2/src$ make
rm -f p2mp
rm -f sender
gcc reliable_udp.c -o p2mp
gcc sender.c linklist.c -o sender
tushar@server:~/ncsu/ip/project/2/src$ ./sender 7735 500 socket_client.c 192.168.1.6
 Client  IP :192.168.1.6
Timeout, Sequence Number        :       27
Timeout, Sequence Number        :       32
Timeout, Sequence Number        :       105
Timeout, Sequence Number        :       105
Timeout, Sequence Number        :       117
^C
tushar@server:~/ncsu/ip/project/2/src$
tushar@server:~/ncsu/ip/project/2/src$
```

Fig.5 Output format of the Receiver (P2MP-FTP Server)

```
tushar@server:~/ncsu/ip/project/2/src$ make
rm -f p2mp
rm -f sender
gcc reliable_udp.c -o p2mp
gcc sender.c linklist.c -o sender
tushar@server:~/ncsu/ip/project/2/src$ ./p2mp 7735 500 b.txt 0.05 192.168.1.6
Packet loss,     :       27
Packet loss,     :       32
Packet loss,     :       105
Packet loss,     :       105
Packet loss,     :       117
^CClient is shutting down.
tushar@server:~/ncsu/ip/project/2/src$
```

Fig.6 Output format of the Sender (P2MP-FTP Client)

Conclusion

We were able to successfully transfer the file from sender to receivers using our program. And built our skills related to writing transport layer services like encapsulating application data into transport layer segments by including transport headers, buffering and managing data received from, or to be delivered to, multiple destinations, managing the window size at the sender, computing checksums, and using the UDP socket interface.

Instructions to run the programs

The code for sender, receiver and a Makefile is provided in separate files.

1. The code was developed on Ubuntu environment in C Language
2. Copy the attached folder to a location on the test machines (both sender and receiver machines)
3. It contains two subdirectories “src” and “include”
4. Change directory to “src”
5. Run ./make >> to trigger makefile
6. Start the receivers first with appropriate arguments for each fields as explained above

    ./p2mp <port#> <mss> <filename> <P> <server-ip>

Example: ./p2mp 7735 500 b.txt 0.05 10.139.60.228

7. Start the sender program with appropriate arguments as explained earlier

./sender <port#> <mss> <filename> <server-1 server-2 …. server-n>

Example: ./sender 7735 500 socket_client.c 10.139.60.228

Raw Data From all the runs

TASK1

Clients	MSS	Error P	Run1 (ns)	Run2 (ns)	Run3 (ns)	Run4 (ns)	Run5 (ns)	Run - Average (sec)	Run - Average (min)
1	500	0.05	126830519412	114054570284	115719408301	127830619513	112054470183	119.2979175	1.988298626
2	500	0.05	187167248929	195609830172	176056137818	188167349030	193609730071	188.1220592	3.135367653
3	500	0.05	301315089137	288218394274	252199055537	302315189238	286218294173	286.0532045	4.767553408
4	500	0.05	355221346375	366332457486	344110235264	356221446476	364332357385	357.2435686	5.954059477
5	500	0.05	442498468077	453609579188	431387356966	443498568178	451609479087	444.5206903	7.408678172

TASK 2

Clients	MSS	Error P	Run1 (ns)	Run2 (ns)	Run3 (ns)	Run4 (ns)	Run5 (ns)	Run - Average (sec)	Run - Average (min)
3	100	0.05	1602238053683	1713349164794	1591126942572	1603238153784	1711349064693	1644.260276	27.40433793
3	200	0.05	986510276446	997621387557	975399165335	987510376547	995621287456	988.5324987	16.47554164
3	300	0.05	391370853612	402481964723	380259742501	392370953713	400481864622	393.3930758	6.556551264
3	400	0.05	305050430112	316161541223	293939319001	306050530213	314161441122	307.0726523	5.117877539
3	500	0.05	252199055537	263310166648	241087944426	253199155638	261310066547	254.2212778	4.237021296
3	600	0.05	212695955	223807066	201584844	213696055	221806966	214.7181778	3.578636297

			586	697	475	687	596		
3	700	0.05	163795843756	174906954867	152684732645	164795943857	172906854766	165.818066	2.763634433
3	800	0.05	127746396685	138857507796	116635285574	128746496786	136857407695	129.7686189	2.162810315
3	900	0.05	239237592105	250348703216	228126480994	240237692206	248348603115	241.2598143	4.020996905
3	1000	0.05	207581019599	218692130710	196469908488	208581119700	216692030609	209.6032418	3.493387364

TASK 3

Clients	**MSS**	**Error P**	**Run1 (ns)**	**Run2 (ns)**	**Run3 (ns)**	**Run4 (ns)**	**Run5 (ns)**	**Run - Average (sec)**	**Run - Average (min)**
3	500	0.01	163795843756	52214611801	55261622068	164795943857	50214511700	97.25650664	1.620941777
3	500	0.02	106880395438	100212364093	95769284327	107880495539	98212263992	101.7909607	1.696516011
3	500	0.03	155025480543	168333577743	143914369432	156025580644	166333477642	157.9264972	2.632108287
3	500	0.04	210624973713	221736084824	199513862602	211625073814	219735984723	212.6471959	3.544119932
3	500	0.05	252199055537	263310166648	241087944426	253199155638	261310066547	254.2212778	4.237021296
3	500	0.06	296128797281	307239908392	285017686170	297128897382	305239808291	298.1510195	4.969183658
3	500	0.07	346876020406	357987131517	335764909295	347876120507	355987031416	348.8982426	5.81497071
3	500	0.08	436183902603	447295013714	425072791492	437184002704	445294913613	438.2061248	7.303435414
3	500	0.09	431105913597	442217024708	442217024708	432106013698	440216924607	437.5725803	7.292876338
3	500	0.1	484357664938	495468776049	473246553827	485357765039	493468675948	486.3798872	8.106331453