CSC 573 Internet Protocols
Project 2

n-dent v1.0

Reliable Multicast over UDP

AdithSudhakar (asudhak)
DinojaPadmanabhan(dpadman)
RaghavendranNedunchezhian (rnedunc)
Srinath Krishna Ananthakrishnan (sananth5)

1. Introduction

TCP offers a connection oriented service over the best effort, no frills base of IP, in today's Internet. However, TCP brings with itself the constraint of congestion control and the related complexities owing to which the latency of file transfer is drastically reduced. Also, current implementation of multicast over IP doesn't provide reliability. In this report, we present n-dent, apoint-to-multipoint reliable file transfer protocol over UDP. The report is organized in the following manner. Section 2 deals with the design goals of the protocol which is followed by the implementation in Section 3. We present the results of the offline experiments in Section 4 followed by conclusion and future work.

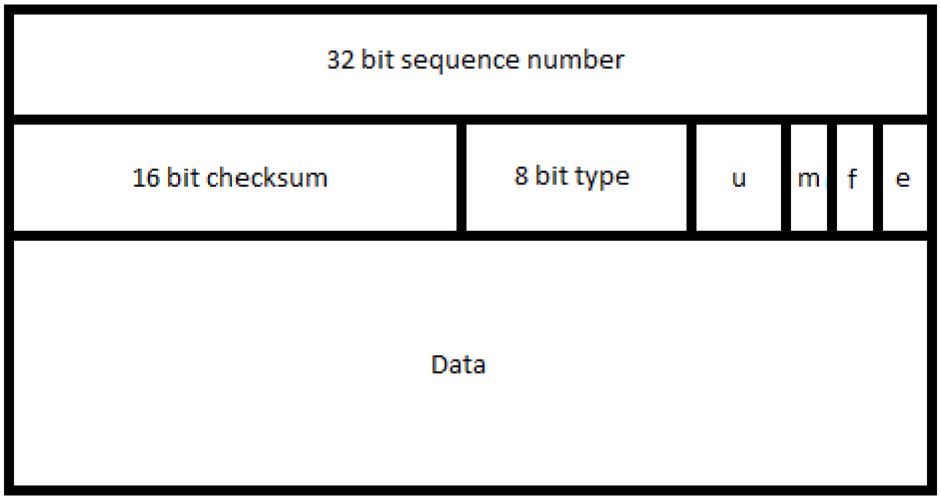
2. Design of n-dent

It seems apt to first explain why the name n-dent was chosen for the protocol. With 1 sender and 3 receivers, the flow of data from the sender to the receivers looks like the forks of a trident. Thus, for a generic number of receivers 'n', we chose the name n-dent.

The design goals for the protocol are as follows.

1. The client or the sender should transfer a file to multiple servers or receivers that are online listening on specified ports.
2. The protocol should support reliability over UDP using an automatic repeat request (ARQ) scheme.
3. This, intended to be a transport layer protocol, should seamlessly receive data from the application, which can send data at will.

In addition to these, there might be some additional support from the protocol, but the above goals must be certainly met. To combat these design goals, the protocol specifies a header of 8 bytes which is as follows.



The first 4 bytes contain the sequence number of the packet sent, followed by 2 bytes containing the internet checksum as computed in TCP/UDP. This is followed by the 8 bit type field which currently has two values – 0x55, denoting a data message and 0xAA, denoting an acknowledgement from the receiver. The next 8 bits are special flags, of which the first 5 bits are currently unused. The 6th bit is set when the packet contains an md5 checksum of the file and the 7th bit is sent when the packet contains the filename sent.The sender can send the md5 checksum as part of the first packet along with a filename.The 8th bit of the flags is set for the last packet to signal the end of the transmission. The acknowledgement currently doesn't make use of the flags and the data portion of the message.

The acknowledgements are cumulative and denote the last received packet. A receiver can send several acknowledgements for the same packet if it received out of sequence packets. On receiving two such duplicate acknowledgements, the sender shall send the expected packet to the receivers who haven't acknowledged the packet, yet.The sender shall maintain a transmission window worth of packets which are sent to multiple receivers. The sender also shallmaintain a timer for the oldest outstanding packet. The sender shall slide the window when the acknowledgement for the oldest outstanding packet arrives. The receivers are relatively simple wherein they shall buffer, out of sequence packets within the reception window and write data to the file as packets are received in sequence.

3. Implementation

The implementation is done in C. There are two programs, one each for the server and the client whose usage is as follows.

```
p2mpclient server-1 server-1-port [server-2 server-2-port ... server-r server-r-port]
file-name N MSS max-rtt
        server-n      - server-n's ip address (at least 1 server should be specified,
max 10
        server-n-port - server-n's port (at least 1 server should be specified, max 10
        file-name     - file to be transferred
        N             - window size
        MSS           - maximum segment size of each segment (< 1500)
        max-rtt       - maximum rtt (in ms) of all receivers (found using rttserver
and rttclient programs)

p2mpserver port# file-name N p
        port# - port number to which the server is listening
        file-name - file where the data will be written
        N - window size
        p - probability of packet loss, ranges between 0 and 1
```

The p2mpclient takes the server(s) IP addresses and port, the file to be sent, the window size, the MSS (the size of the data bytes in the message format shown above) and the maximum RTT of the connections with the receivers for setting the timeout value. The p2mpserver takes the port on which it binds, the filename to be stored, the window size and a probability p of losing packets to simulate packet loss.

The p2mpclient has three threads – rdt_send, sender and the receiver. rdt_send simulates the application layer protocol which feeds data constantly by reading the file specified. This thread puts in data in the buffer and stalls when full. The sender thread sends data from the buffer to all the receivers and starts a timer for the oldest outstanding packet. The receiver thread receives acknowledgements and appropriately slides the window. The timer, on firing, spawns a new thread, which then retransmits the packets. We also specify the maximum RTTof different connections to determine the timeout. The timeout is set as 2*RTT. We have used POSIX timers to implement timer functionality.

The p2mpserver is a single threaded application which waits for packets from the sender and acknowledges them as per the protocol demands. The p2mpserver takes in a filename, but if a filename is sent as part of the first packet and a file doesn't exist with that name, then it is made use of. If the file exists, the command line filename is used.

To test the implementation, we chose to use port forwarding. The different servers are housed in different machines and ports on the routers are opened. We used Cisco E2000 and E1000 routers which support port forwarding. However, traceroute cannot be used to obtain RTT between hosts behind NATs. So, we wrote a small program that could send a burst of UDP packets and measure the round trip time for them. We take the average of these measurements as the rtt of that connection. The maximum of such rtts is specified to the p2mpclient program to determine the timeout value. The file transferred between the hosts is an mp3 file of size 2.2 MB.Please note the protocol implementation works for both binary and text files. We also collect a bunch of statistics, which are presented after the file transfer is complete. The README file along with source code has detailed information on executing this project.

Source code location at GITHUB - https://github.com/borncrusader/n-dent

4. Results for offline Experiments

I. Effect of Window size N

MSS=500 bytes, R =3, P =0.05

N=1

Trial	1	2	3	4	5	Average
Time	719.3	698.36	701.5	655.6	729.3	700.812

N=2

Trial	1	2	3	4	5	Average
Time	223.36	191.86	177.18	170.15	438.23	240.156

N=4

Trial	1	2	3	4	5	Average
Time	352.18	198.274	241.93	329.14	289.32	282.1688

N=8

Trial	1	2	3	4	5	Average
Time	410.56	653.9	588.1	507.28	579.26	547.82

N=16

Trial	1	2	3	4	5	Average
Time	940.23	1023.5	973.5	901.33	1002.41	968.194

N=32

Trial	1	2	3	4	5	Average
Time	209	196.3	177.4	201.1	129.3	182.62

N=64

Trial	1	2	3	4	5	Average
Time	136	258.6	149.6	179.6	215.6	187.88

N=128

Trial	1	2	3	4	5	Average
Time	192	278	177.2	302	223.3	234.5

N=256

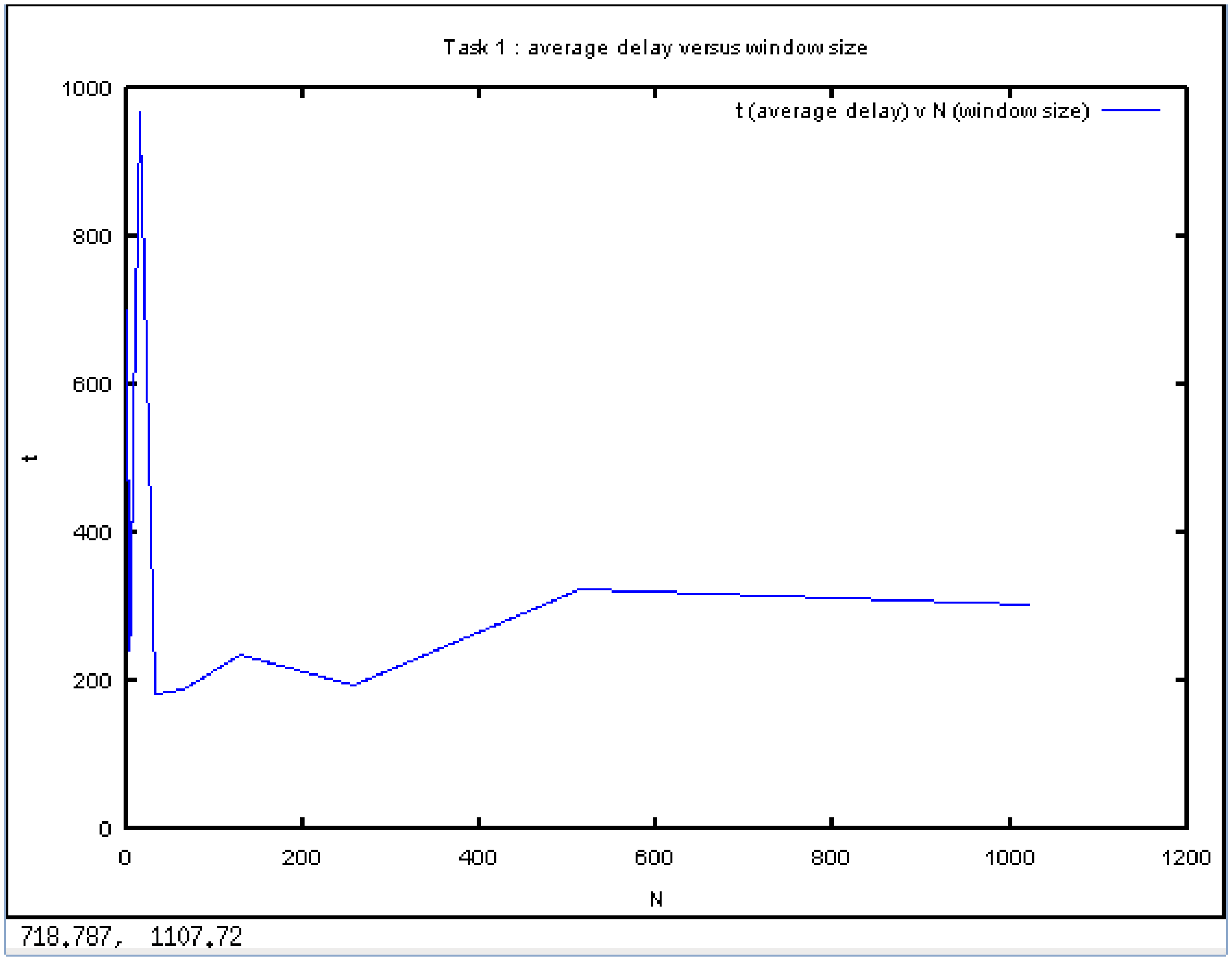
Trial	1	2	3	4	5	Average
Time	162	147.6	199.3	174.5	287.6	194.2

N = 512

Trial	1	2	3	4	5	Average
Time	308	369.2	402.1	287.6	247.3	322.84

N=1024

Trial	1	2	3	4	5	Average
Time	312	398	278	269	266.6	304.72



The graph suggests that as the window size increases, the average delay decreases. But this depends on factors such as the probability, real time internet traffic and the number of measurements. The graph shows a couple of anomalies owing to the above mentioned factors. However the size of the window increases in higher order (from N >= 32) the graph depicts the expected behavior.

II. Effect of Receiver Set size r

N=16, MSS=500 Bytes, P = 0.05

R = 1

Trial	**1**	**2**	**3**	**4**	**5**	**Average**
Time	28.59	29.32	23.02	30.14	25.39	27.292

R = 2

Trial	**1**	**2**	**3**	**4**	**5**	**Average**
Time	150.08	104.28	118.26	81.23	124.32	115.634

R = 3

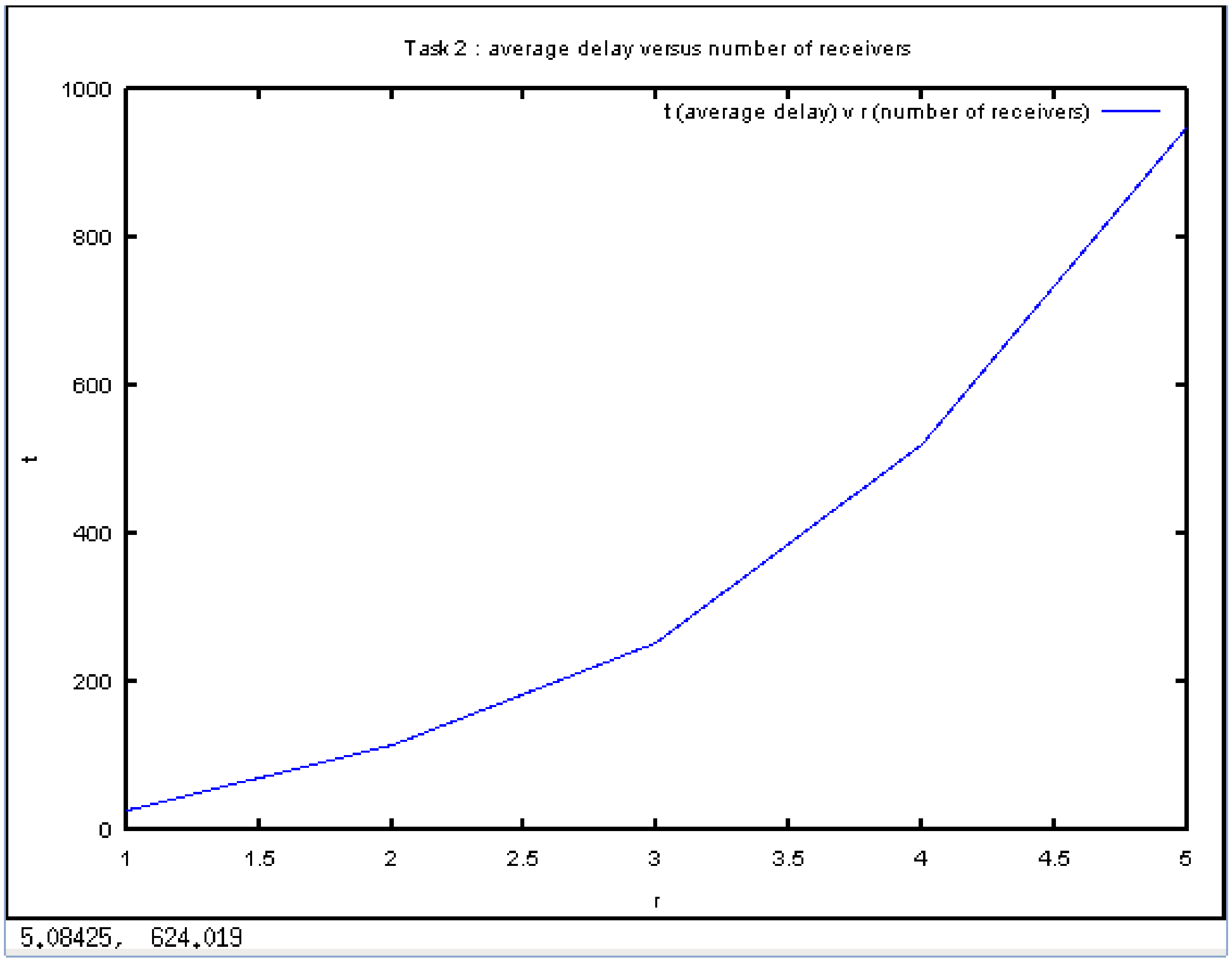
Trial	1	2	3	4	5	Average
Time	240	252.3	267.5	244.23	256.9	252.186

R = 4

Trial	1	2	3	4	5	Average
Time	521.24	598.32	511.11	498.36	478.36	521.478

R = 5

Trial	1	2	3	4	5	Average
Time	922	845.6	1003.57	1120.03	865.6	951.36



The graph suggests that the average delay increases as the number of receivers increase. We see a monotonically increasing curve which can be reasoned as follows. As the number of receivers increase, each receiver can drop packets independently of each other and thus can slide their windows independently. However, the sender has to maintain a window that should cater to all the receivers, i.e. the oldest unacknowledged packet would be the oldest across all the receivers. Thus, even though a receiver might have received packets in its window, it would still have to wait for other receivers to acknowledge them.

III. Effect of MSS

N=16, R=3 and P =0.05

MSS=100

Trial	1	2	3	4	5	Average
Time	976	658.2	867.2	778.2	897.65	835.45

MSS=200

Trial	1	2	3	4	5	Average
Time	305	298.7	369.2	256.12	301.2	306.044

MSS=300

Trial	1	2	3	4	5	Average
Time	270	269.2	198.2	247	215.3	239.94

MSS=400

Trial	1	2	3	4	5	Average
Time	198	221	210.11	200.1	231.5	212.142

MSS=500

Trial	1	2	3	4	5	Average
Time	240	252.3	267.5	244.23	256.9	252.186

MSS=600

Trial	1	2	3	4	5	Average
Time	200.12	211.32	204.36	199.6	193.5	201.78

MSS=700

Trial	1	2	3	4	5	Average
Time	131.34	125.22	147.21	135.23	172.3	142.26

MSS=800

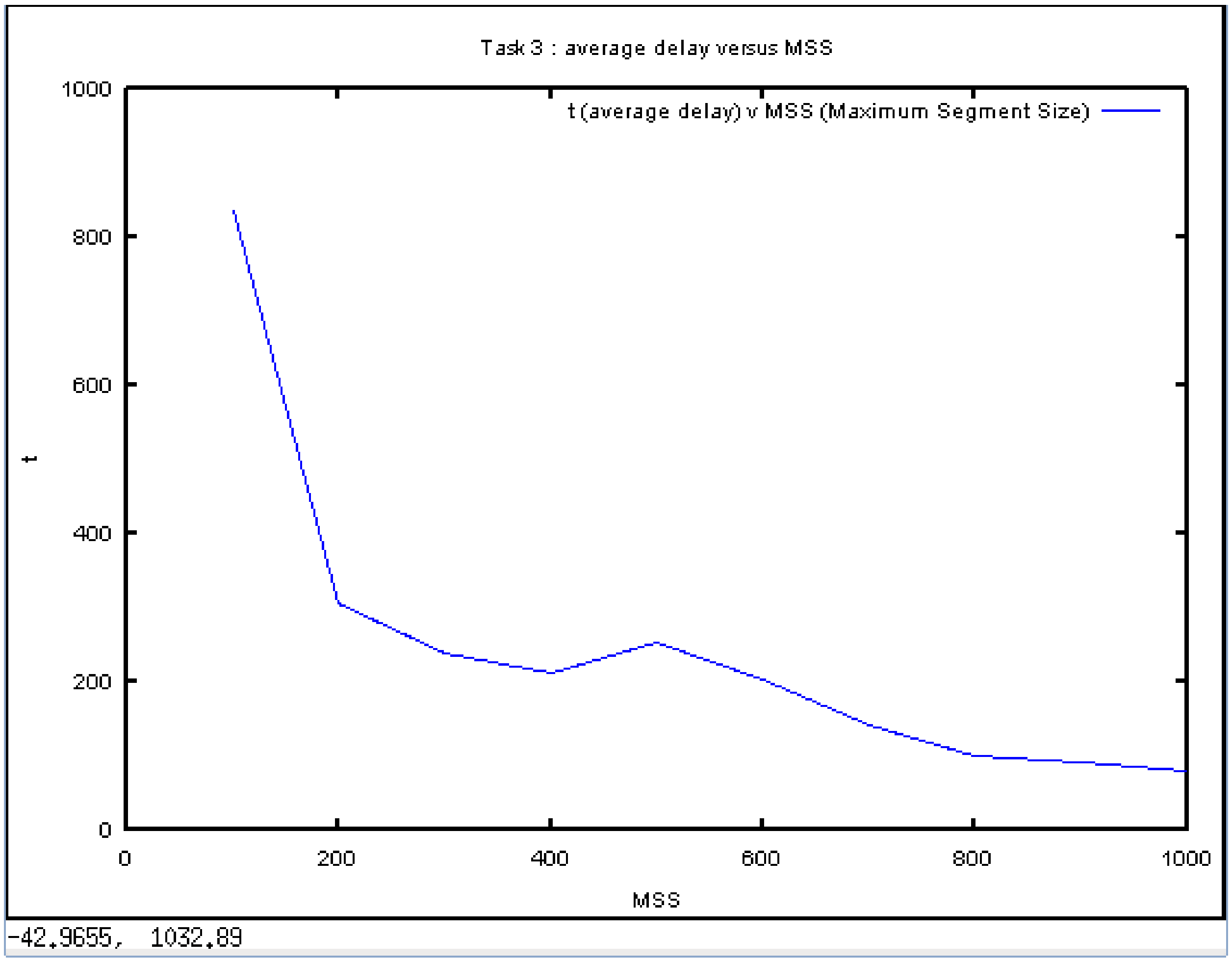
Trial	1	2	3	4	5	Average
Time	89	98.3	99	105.3	104	99.12

MSS=900

Trial	1	2	3	4	5	Average
Time	92.4	73.2	132.3	91.3	66.5	91.14

MSS=1000

Trial	1	2	3	4	5	Average
Time	87	75	63	91.2	88.3	80.9



The graph suggests that as the MSS increases, we see a decrease in the average delay. This is because as the MSS increases, the number of packets decreases. Even though the transmission delay for a packet increases, the number of retransmissions is less, thereby reducing the time to transfer. We encounter a small increase in average delay with MSS = 500, for 5 measurements. If the number of measurements were high then we could have got a smooth curve.

IV. Effect of Loss probability p

N=16, MSS=500 and R=3

P =0.01

Trial	1	2	3	4	5	Average
Time	220	215.6	238.3	214.22	136.8	204.984

P =0.02

Trial	1	2	3	4	5	Average
Time	252.3	148.3	116.5	127.3	174.6	163.8

P =0.03

Trial	1	2	3	4	5	Average
Time	138.03	269.4	197.65	342.21	321.33	253.724

P =0.04

Trial	1	2	3	4	5	Average
Time	132.23	174	187.2	203.1	166.6	172.626

P =0.05

Trial	1	2	3	4	5	Average
Time	240	252.3	267.5	244.23	256.9	252.186

P =0.06

Trial	1	2	3	4	5	Average
Time	137	132.3	99.3	147.3	153.47	133.874

P =0.07

Trial	1	2	3	4	5	Average
Time	150	169.3	174.2	133.3	108	146.96

P =0.08

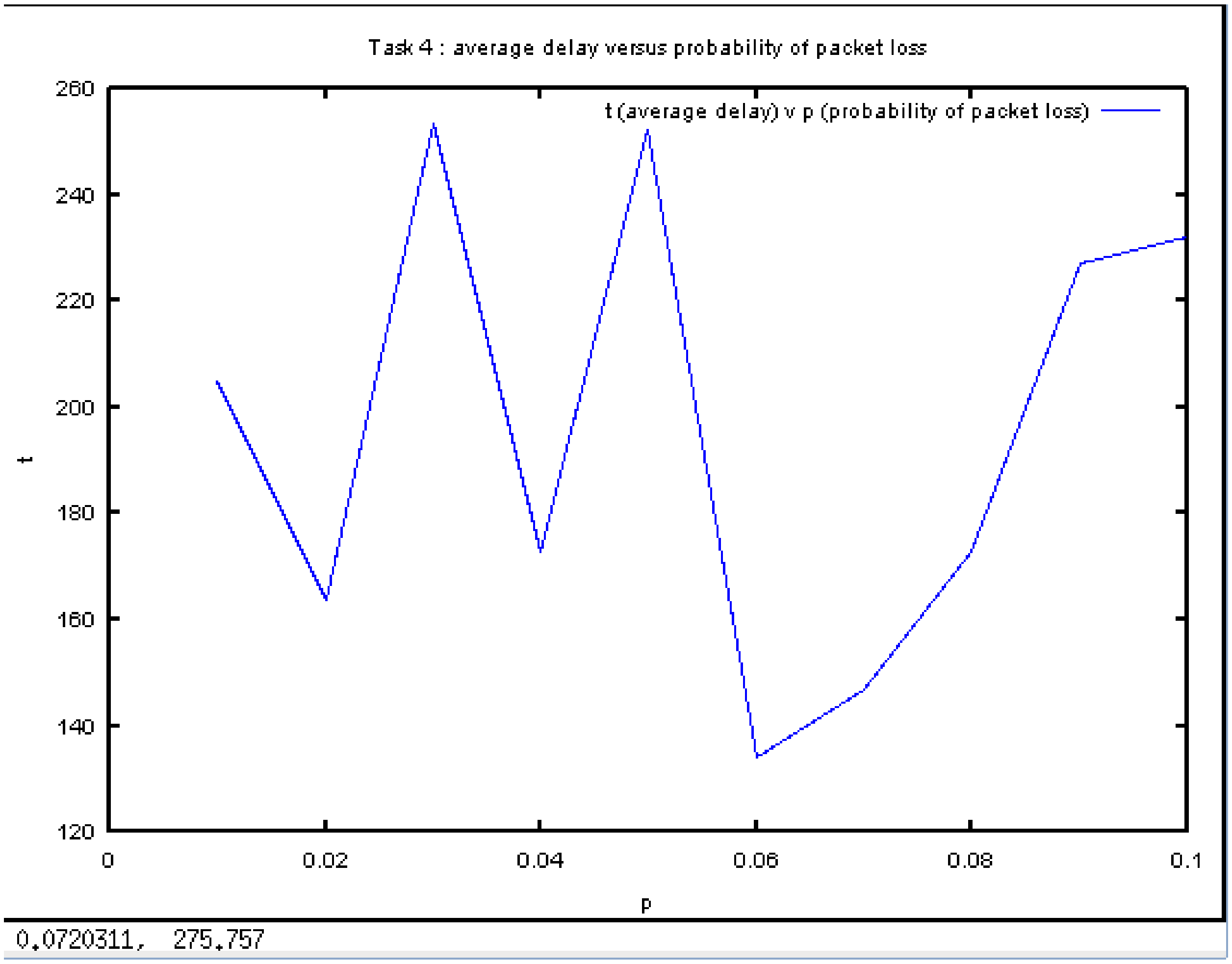
Trial	1	2	3	4	5	Average
Time	138	149.3	200.1	187.3	188.2	172.58

P =0.09

Trial	1	2	3	4	5	Average
Time	191	187	236.2	244.2	277.2	227.12

P =0.10

Trial	1	2	3	4	5	Average
Time	203	233.6	247	278	198.3	231.98



This graph is erratic. This stems from the probabilistic drop of packets. Ideally as the packet loss increases, the average delay should increase. Owing to the fact that we use a pseudo-random number generator and depend on a mere 5 measurements, the graph doesn't give a clear picture. As the measurements increase, we will get a more accurate curve.

5. Conclusions and Future work

In this report, we presented the n-dent protocol for reliable multicast transfer of files and we believe there is a lot of scope for improvement. We outline the future work we intend to pursue.

- Currently, sequence numbers start from 0. But, this proves a candidate for security breaches. So, sequence numbers can be made to start randomly.
- UDP's no restriction on sending packets increases the congestion in the network. Though n-dent provides a basic reliability service using ARQ schemes.
- There are some loose ends currently, for instance the last acknowledgement from the receiver can be lost, but the sender wouldn't know it and would timeout. This can be solved by having a proper handshaking and teardown of the connection with associated timers.

- Currently, the rtt is measured using a different program and is fed to determine the timeout value. However, the rtt can be measured as part of the program and also can be used to dynamically change the timeout value as in TCP.
- For the program to work between hosts behind NATs, we explored the possibility of using port forwarding as part of our implementation process. However, we propose to explore another method called UDP hole punching to establish a "connection" between the hosts, as explained here - http://www.brynosaurus.com/pub/net/p2pnat/

We thank Dr. Rouskas for giving us the opportunity to explore the wonderful world of protocol design and implementation. We believe this to be an excellent precursor for the literature report.