Venkata Surya Teja Penumatcha
Michael Patel
ECE 573 Project 2 Report

**Overview**

This project explores point-to-multipoint reliable data transfer. Protocols used in this assignment include Stop-and-Wait (SAW), Automatic Repeat Request (ARQ) and UDP. Data is encapsulated in segments with appropriate header fields and transported as if the data transfer were reliable using checksums and window sizing management.

The scheme of the project uses Python and several hosts that are configured as either clients or servers. There is only one single client which serves as a sender for all servers (aka receivers). Thus, a single data segment is broadcasted at a time to all receivers. The sender must wait to receive an acknowledgement message from each separate receiver before continuing with the next data segment. All data transfer occurs from the sender to the receivers. The only messages that travel from the receivers to the sender are acknowledgements.

The sender begins by reading in a data file specified by the command line. It then calls rdt_send() - a reliable data transfer send function to implement data transfer to the receivers. Using a MSS value also specified by the command line, a data segment is formed that includes an MSS worth of byte data and a header. The sender then broadcasts this segment to all receivers separately using UDP socket connections.

For each segment sent, a timeout counter is initiated. If the timeout counter expires before the sender receives all acknowledgements, retransmits the segment, but only to those receivers which did not receive the initial segment. Once all outstanding acknowledgements have been received at the sender, the sender will proceed to sending out the next data.

A SAW protocol is used with an initial sequence number set to zero. Thus, there is only one outstanding data segment at a time. The timer value is set at 100 ms. The header of the segment includes a 32-bit sequence number, 16-bit UDP checksum and 16-bit field with a unique value signifying that it is a ‘data’ packet.

All receivers listen on the same universally-known port numbers. Whenever, a receivers receives incoming segments, a UDP checksum is calculated and verified against the checksum calculated by the sender in the header. If the checksum is incorrect, then nothing else happens. If the checksum is correct and the incoming segment arrived in-order, then the receiver writes the extracted data to a text file as specified by the command line.

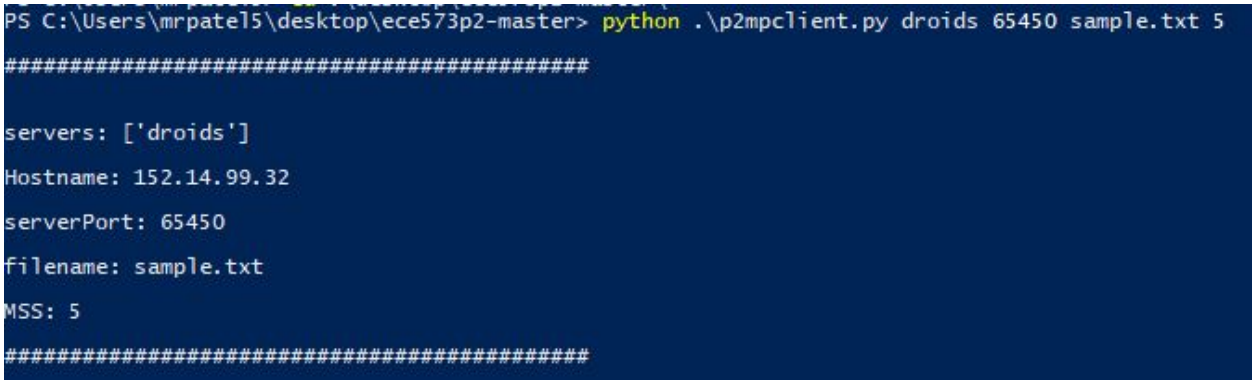
An acknowledgement segment is then sent in response to the sender. The header of the acknowledgement segment includes a 32-bit sequence number of that which is being acknowledged, a 16-bit field of all zeroes and 16-bit field with a unique value signifying that it is an ‘acknowledgement’ packet.

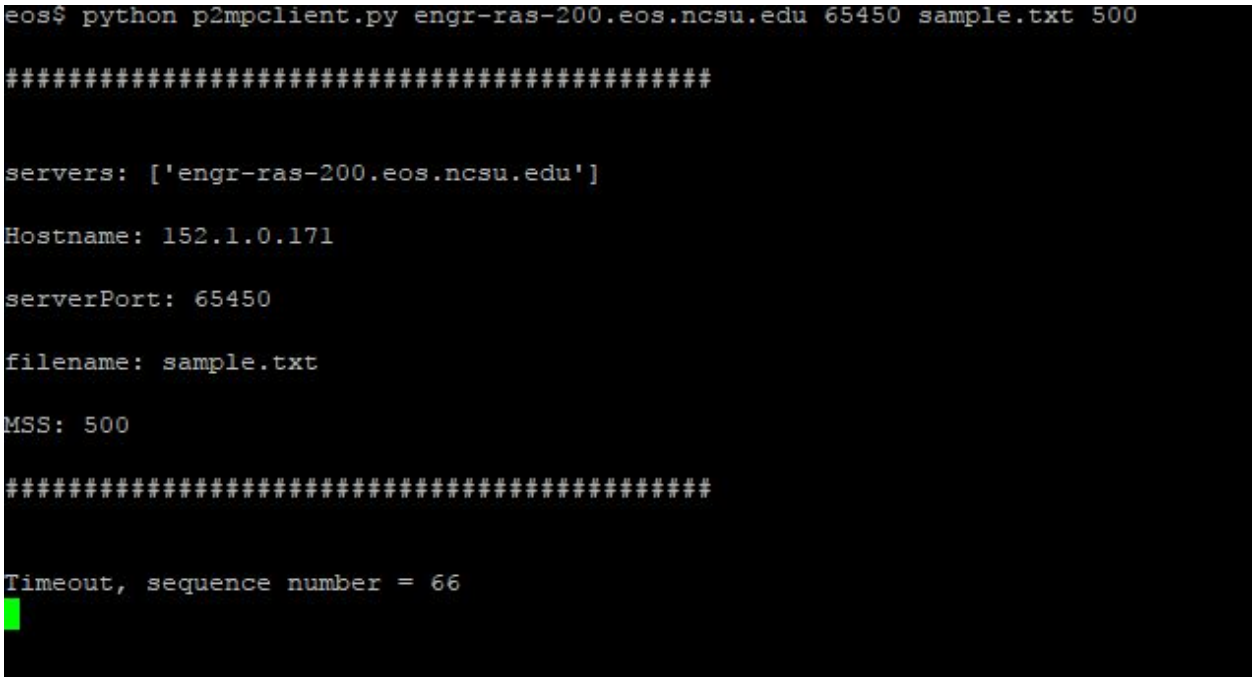
To simulate packet losses, a packet loss probability p is specified by the command line. When a receiver receives an incoming segment, a random value r is generated between 0 and 1. If r > p, then the segment is accepted. If r <= p, then the segment is ‘dropped’ by ignoring it.

For this project, 125 simulation runs were done. The purpose was to examine the varying effect of MSS, packet loss probability and number of receivers against the average delay to send a 1MB text file in seconds.

**Format of command line: Client (sender):**

<lf> <python> <p2mpclient.py> <hostname of server 1> <hostname of server 2> < hostname of server 3> … <port number> <text file to be read> <message segment size> <lf>



**Format of command line: Server (receiver):**

<lf> <python> <p2mpserver.py> <port number> <text file where data is written> <packet loss probability> <lf>

```
PS C:\Users\mrpatel5\desktop\ece573p2-master> python .\p2mpserver.py 65450 new.txt 0.1
##############################################

Hostname: DROIDS
IP address: 152.14.99.32
Port number: 65450
Filename: new.txt
Packet Loss Probability: 0.1
##############################################

UDP Server is listening...
```

```
eos$ python p2mpserver.py 65450 1.txt 0.1
##############################################

Hostname: engr-ras-200.eos.ncsu.edu
IP address: 152.1.0.171
Port number: 65450
Filename: 1.txt
Packet Loss Probability: 0.1
##############################################

UDP Server is listening...
```

```
eos$ python p2mpserver.py 65450 2.txt 0.03

################################################

Hostname: engr-ras-201.eos.ncsu.edu

IP address: 152.1.0.172

Port number: 65450

Filename: 2.txt

Packet Loss Probability: 0.03

################################################

UDP Server is listening...
```

**Format of message exchange: Client (sender):**

<lf> <”Timeout, sequence number = “> <sequence number when timeout occurs> <lf>

```
Timeout, sequence number = 8
Timeout, sequence number = 16
Timeout, sequence number = 27
Timeout, sequence number = 32
Timeout, sequence number = 40
Timeout, sequence number = 44
Timeout, sequence number = 63
Timeout, sequence number = 85
```

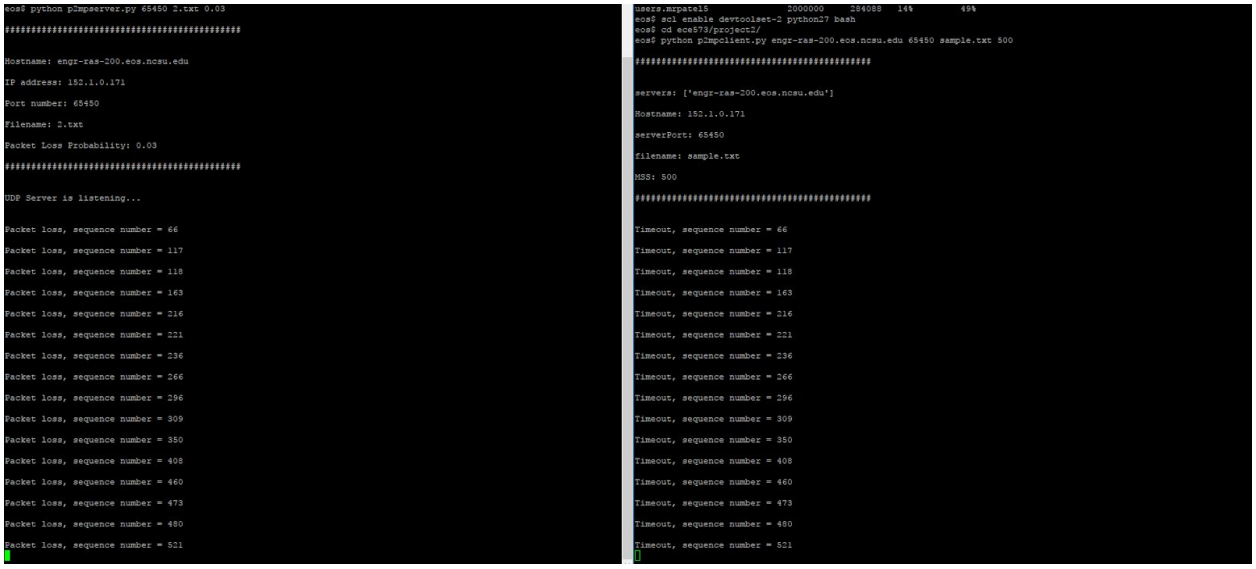
**Format of message exchange: Server (receiver):**

<lf> <"Packet loss, sequence number = "> <sequence number when 'loss' occurs> <lf>

```
UDP Server is listening...

Packet loss, sequence number = 8
Packet loss, sequence number = 16
Packet loss, sequence number = 27
Packet loss, sequence number = 32
Packet loss, sequence number = 40
Packet loss, sequence number = 44
Packet loss, sequence number = 63
Packet loss, sequence number = 85
```

**Format of message exchange:**




**Header: Client (sender):**

DATA_FIELD = '0101010101010101'
LAST_FIELD = '1100110011001100' (for the last data segment sent as may not be size equal to MSS bytes)

**Header: Server (receiver):**

ACK_FIELD = '1010101010101010'

**Ping Experiments** to determine an appropriate timeout value

```
eos$ ping -c 4 engr-ras-201.eos.ncsu.edu
PING engr-ras-201.eos.ncsu.edu (152.1.0.172) 56(84) bytes of data.
64 bytes from engr-ras-201.eos.ncsu.edu (152.1.0.172): icmp_seq=1 ttl=64 time=0.504 ms
64 bytes from engr-ras-201.eos.ncsu.edu (152.1.0.172): icmp_seq=2 ttl=64 time=0.368 ms
64 bytes from engr-ras-201.eos.ncsu.edu (152.1.0.172): icmp_seq=3 ttl=64 time=0.403 ms
64 bytes from engr-ras-201.eos.ncsu.edu (152.1.0.172): icmp_seq=4 ttl=64 time=0.393 ms

--- engr-ras-201.eos.ncsu.edu ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.368/0.417/0.504/0.051 ms
eos$
```

```
eos$ ping -c 4 engr-ras-200.eos.ncsu.edu
PING engr-ras-200.eos.ncsu.edu (152.1.0.171) 56(84) bytes of data.
64 bytes from engr-ras-200.eos.ncsu.edu (152.1.0.171): icmp_seq=1 ttl=64 time=0.261 ms
64 bytes from engr-ras-200.eos.ncsu.edu (152.1.0.171): icmp_seq=2 ttl=64 time=0.366 ms
64 bytes from engr-ras-200.eos.ncsu.edu (152.1.0.171): icmp_seq=3 ttl=64 time=0.281 ms
64 bytes from engr-ras-200.eos.ncsu.edu (152.1.0.171): icmp_seq=4 ttl=64 time=0.349 ms

--- engr-ras-200.eos.ncsu.edu ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.261/0.314/0.366/0.045 ms
eos$
```

```
PS C:\Users\mrpatel5> ping engr-ras-201.eos.ncsu.edu

Pinging engr-ras-201.eos.ncsu.edu [152.1.0.172] with 32 bytes of data:
Reply from 152.1.0.172: bytes=32 time=1ms TTL=56
Reply from 152.1.0.172: bytes=32 time=1ms TTL=56
Reply from 152.1.0.172: bytes=32 time=1ms TTL=56
Reply from 152.1.0.172: bytes=32 time=1ms TTL=56

Ping statistics for 152.1.0.172:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 1ms, Average = 1ms
PS C:\Users\mrpatel5>
```

**Offline Experiments:**

p = packet loss probability

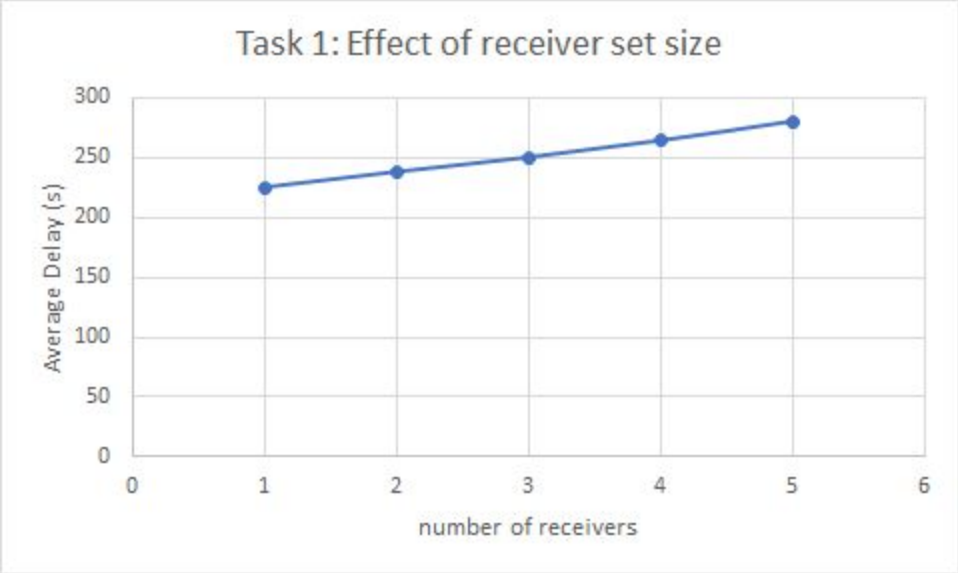
MSS = maximum segment size

n = number of receivers

Venkata Surya Teja Penumatcha
Michael Patel
ECE 573 Project 2 Report

**Task 1: Effect of the receiver set size n**

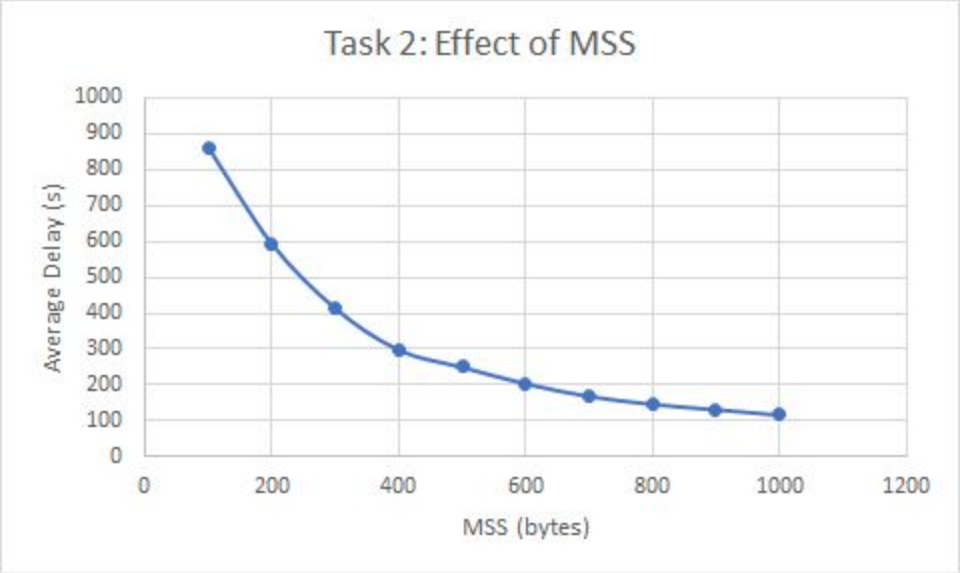
To simulate the effect of a varying receiver set size on the average delay, a grendel machine was used as the sender with 1...5 receivers set up on different remote-linux eos machines. A 1MB text file was broadcasted with MSS = 500 and p = 0.05 to the different sets of receivers. Each simulation was repeated four additional times, and an average time delay was computed. Thus, there were twenty-five Python simulation runs.




As the number of receivers increases, the average delay increases linearly as well. Although the best performance was achieved with only one receiver, the average delay was not severely negatively impacted by the increasing receiver set size. The MSS was set to 500 bytes, and the packet loss probability was 0.05.

**Task 2: Effect of MSS**

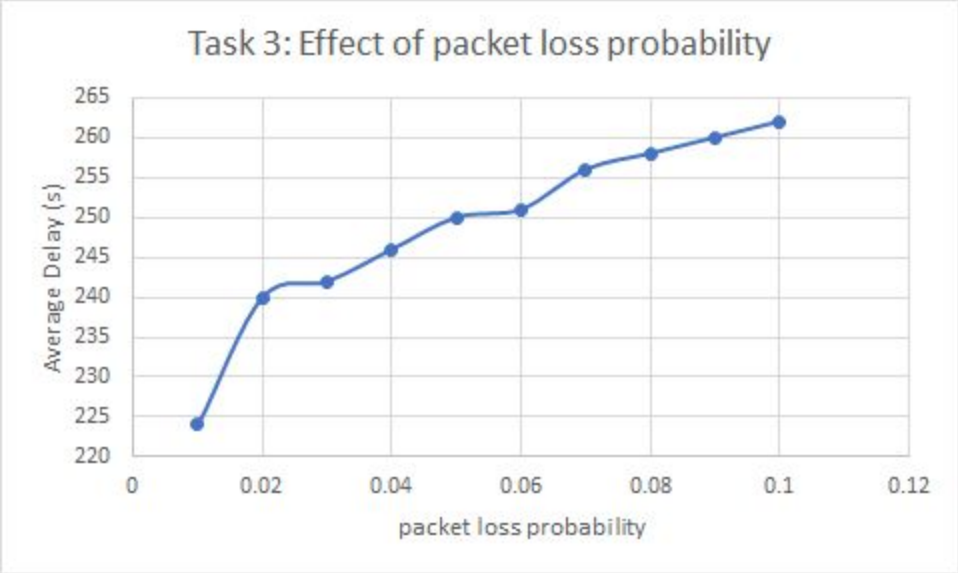
To simulate the effect of varying MSS specifications on the average delay, a grendel machine was used as the sender with 3 receivers set up on different remote-linux eos machines. A 1MB text file was broadcasted with p = 0.05 to the receiver set. The MSS was varied from 100 to 1000 in increments of 100. Each simulation was repeated four additional times, and an average time delay was computed. Thus, there were fifty Python simulation runs.




As the MSS increases, the average delay exponentially decreases approaching close to 100ms. This coincides with the specified timeout value at 100ms. Thus, a larger MSS leads to better average delay performance. The number of receivers was fixed to 3, and the packet loss probability was 0.05.

**Task 3: Effect of packet loss probability p**

To simulate the effect of varying packet loss probability on average delay, a grendel machine was used as the sender with 3 receivers set up on different remote-linux eos machines. A 1MB text file was broadcasted with MSS = 500 to the receiver set. The packet loss probability p was varied from 0.01 to 0.10 in increments of 0.01.. Each simulation was repeated four additional times, and an average time delay was computed. Thus, there were fifty Python simulation runs.




As the packet loss probability increases, so does the average delay. With a small packet loss probability (p = 0.01), the average delay is near 100ms. With a packet loss probability equal to 0.02 or greater, the average delay jumps up to approximately 240s, and then increases approximately linearly. This is the same as the specified timeout value. The number of receivers was fixed to 3, and the MSS was set to 500.

**Conclusion**

After the Python simulations were completed, the following conclusions can be determined: (1) Large receiver sets result in long average delays. The average delay increases linearly with the number of receivers. (2) Large MSS specifications result in short average delays. The average delay decreases exponentially with MSS. (3) Large packet loss probability result in long average delays. The average delay increases exponentially with large deviations in packet loss probability and linearly with short deviations.

A reliable data transfer model was constructed using UDP, SAW and ARQ protocols in the Python27 and Python36 languages. Data checksums and window management were practices used to ensure reliability.