1. Install Mininet on your computer/laptop/MacBook (host node): Follow the Prefer Option1 mode to setup
Mininet. Download Mininet VM and create new VM in VirtualBox say “Mininet-VM1” in VirtualBox. Add the NAT Network Interface. Boot upthe VM and check the ifconfig and connectivity to outside network either by trying to ping to your local host IP address or trying to ping some website.

a. Capture the output of ‘ifconfig’ command and try to identify the Interface address, IP Address, Subnet Address details. What specific observations can you make here?

b. Run Wireshark or TCPdump tool on your local host node (Windows/MAC/Linux desktop or laptop that you are using) and capture the traffic exchanged with the Ubuntu VM. What kind of packets do you see?

c. Install the Networking tools “iperf3” – very simple, lightweight, cross-platform network performance benchmark tool on your host node.

Solution:

a.

```
mininet@mininet-vm:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:74:9a:f9
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:369 errors:0 dropped:0 overruns:0 frame:0
          TX packets:327 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:34476 (34.4 KB)  TX bytes:38775 (38.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:1215 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1215 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:61560 (61.5 KB)  TX bytes:61560 (61.5 KB)

s1        Link encap:Ethernet  HWaddr 9e:7a:14:83:05:47
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

1. **Interface address**: 127.0.0.2 (connected to this NAT through local host and port forwarding)
2. **IP address:** 10.0.2.15
   If connected through bridge method: we will get the following results. Mininet would have the values same as the host network.
    1. IP - 192.168.1.7
    2. Subnet- 255.255.255.0
3. **Subnet Adress:** 10.0.2.X

It is connected to the host internet connection through a virtual wired network. In my case, the host is connected to the internet through wifi (in ifconfig, it will be shown as w0) while inside the VM, it is showing as eth0 or wired connection.

b. Wireshark is used for packet capture.
An exchange of TCP packets takes place between the host and Ubuntu VM.

```
29 14.565972516  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=885 Ack=1 Win=24566 Len=68 TSval=…
30 14.566111810  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=953 Ack=1 Win=24566 Len=68 TSval=…
31 14.566209658  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1021 Win=32742 Len=0 TSval=55531…
32 16.018098176  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1021 Ack=1 Win=24566 Len=68 TSval…
33 16.018124527  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1089 Win=32708 Len=0 TSval=55532…
34 16.019064676  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1089 Ack=1 Win=24566 Len=68 TSval…
35 16.019145788  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1157 Win=32742 Len=0 TSval=55532…
36 16.056055594  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1157 Ack=1 Win=24566 Len=68 TSval…
37 16.056195147  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1225 Win=32742 Len=0 TSval=55532…
38 16.056894289  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1225 Ack=1 Win=24566 Len=68 TSval…
39 16.056992215  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1293 Win=32742 Len=0 TSval=55532…
40 16.080259941  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1293 Ack=1 Win=24566 Len=68 TSval…
41 16.080384861  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1361 Win=32742 Len=0 TSval=55532…
42 16.080998501  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1361 Ack=1 Win=24566 Len=68 TSval…
43 16.081078892  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1429 Win=32742 Len=0 TSval=55532…
44 16.118169251  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1429 Ack=1 Win=24566 Len=68 TSval…
45 16.118485053  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1497 Win=32742 Len=0 TSval=55532…
46 16.119334880  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1497 Ack=1 Win=24566 Len=68 TSval…
47 16.120102292  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1565 Win=32742 Len=0 TSval=55532…
48 18.798707148  127.0.0.1   127.0.0.2   TCP   134 52942 → 2281 [PSH, ACK] Seq=1565 Ack=1 Win=24566 Len=68 TSval…
49 18.798717942  127.0.0.2   127.0.0.1   TCP    66 2281 → 52942 [ACK] Seq=1 Ack=1633 Win=32708 Len=0 TSval=55532…
```

c. installed using sudo apt-get install iperf3
It is a tool to measure throughput between two devices or nodes.

2. Text files from Project Gutenberg

S.NO	Name	Size of File (MB)
1	Bleak House	2.0
2	Don Quixote	2.4
3	Les Miserables	3.4

4	Middlemarch	1.9
5	War and Peace	3.4

3.

a. To run use: $python tcp_server.py

```
import socket
import os

#Variables
IP_ADDRESS = socket.gethostname()
PORT_NO = 12345
PROTOCOL = 'TCP'
BUFFER = 1024


server_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server_socket.bind((IP_ADDRESS,PORT_NO))
server_socket.listen(5)

# Creating a dictionary of names to get files from index numbers coming from
the client
file_name_list = os.listdir('./text_files')
index_to_names = {}
for key,name in enumerate(file_name_list):
    index_to_names[key]=name

while True:
    print("Im listenting")
    client_socket,address = server_socket.accept()
    print(f"Connection with {address} has been establish")
    index = (client_socket.recv(10)).decode('utf-16')
    print(f"Cleint Requested to sent {index_to_names[int(index)]}")
    book_name = index_to_names[int(index)]
```

```
    book_path = './text_files/'+book_name
    book_size = os.path.getsize(book_path)
    client_socket.send(bytes(book_name[:-4]+str(book_size),'utf-16'))
    with open(book_path) as file:
        message = str(1)
        print("Started to read the file")
        while (len(message)>0):
            message = file.read(BUFFER)
            client_socket.send(bytes(message,'utf-16'))

    print("Hey Client, I'm done with the sending. Enjoy your book!")
```

b. tcp_client.py

To run: $python tcp_client.py -i 0 -b 1024 -r './results/tcp_result.json'

Here -i → Index of the required file
-b → Buffer Size
-r → to save the results into a json file

```
import socket
import os
import argparse
import time
import json
import re

#Argument Parser Section
ap =argparse.ArgumentParser()
ap.add_argument("-i","--index",help ="index to the dictionary")
ap.add_argument("-b","--buffer",help="buffer capacity")
```

```
ap.add_argument("-r","--result",help="output result path")
args = vars(ap.parse_args())

#Variables
IP_ADDRESS = socket.gethostname()
PORT_NO = 12345
PROTOCOL = 'TCP'
INDEX = args['index']
BUFFER = int(args['buffer'])
RESULT_PATH = args['result']

start_time = time.time()
client_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client_socket.connect((IP_ADDRESS,PORT_NO))


# Sending the index of the required file to the server
client_socket.send(bytes(INDEX,'utf-16'))
print(f"Index {INDEX} searching in the server")
book_and_size = (client_socket.recv(100)).decode('utf-16')
book_and_size_list =re.split('(\d+)',book_and_size)
book = book_and_size_list[0]
book_size = book_and_size_list[1]
print(f"Found {book} with index {INDEX}")

# Opening the output file and creating the file name
pid = str(os.getpid())
output_path ='./received_files/'+book+'_'+PROTOCOL+'_'+pid+'.txt'
write_file = open(output_path,'w')
message = bytes('','utf-16')
print(f"Started receiving the file")



while True:
```

```
  try:
      temp_message = client_socket.recv(BUFFER)
      message += temp_message
      client_socket.settimeout(0.1)

      print(f"Time elapsed: {time.time() - start_time}s")
  except socket.timeout:



      write_file.write(message.decode('utf-16'))
      final_time = time.time() - start_time
      print(f"it took {final_time}s to receive the file")
      print(f" {book} received completely using TCP protocol with {BUFFER}
bytes buffer. Thanks server!")
      size_of_file = os.path.getsize(output_path)
      through_put = size_of_file/final_time
      dictionary =
{"Book_Name":book,"Protocol":PROTOCOL,"PID":pid,"Buffer_Size":BUFFER,"Total_
Time_Taken":final_time,"Original
Size":book_size,"Size_of_the_file_created":size_of_file,"Throughput":through
_put}

      with open(RESULT_PATH, 'r+') as f:
          if len(f.read()) == 0:
              f.write('['+json.dumps(dictionary))
          else:
              f.write(',\n' + json.dumps(dictionary))


      client_socket.close()
      break
```

c. udp_server.py

To run: $python udp_server.py

```
import socket
import os



#Variables
IP_ADDRESS = socket.gethostname()
PORT = 12345
BUFFER = 1024

server_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
server_socket.bind((IP_ADDRESS,PORT))

# Creating a dictionary of names to get files from index numbers coming from
the client
file_name_list = os.listdir('./text_files')
index_to_names = {}
for key,name in enumerate(file_name_list):
   index_to_names[key]=name

while True:
   print("I'm Listening...")

   data,address = server_socket.recvfrom(32)
   print(f"Connection with {address} has been establish")
   index = int(data.decode('utf-16'))
   book_name = index_to_names[index]
   book_path = './text_files/'+book_name
   book_size = os.path.getsize(book_path)

server_socket.sendto(bytes(book_name[:-4]+str(book_size),'utf-16'),address)
```

```
    print(f"Client Requested to sent {book_name[:-4]}")
    with open(book_path) as file:
        print("Started to read the file")
        message = str(1)

        while (len(message)>0):
            message = file.read(BUFFER)
            server_socket.sendto(bytes(message,'utf-16'),address)
    print("I'm done with the sending. Enjoy you book!")
```

d. Udp_client.py

To run: $ python udp_client.py -i 0 -b 1024 -r './results/udp_result.json'
Here -i → Index of the required file
    -b → Buffer Size
    -r → to save the results into a json file

```
import socket
import os
import argparse
import time
import json
import re

#argument parser to get input from command
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--index", help = "index to the dictionary")
ap.add_argument("-b", "--buffer", help = "index to the dictionary")
ap.add_argument("-r", "--result", help = "path to the output")
args = vars(ap.parse_args())


#Variables
```

```
IP_ADDRESS = socket.gethostname()
PORT = 12345
BUFFER = int(args['buffer'])
INDEX = args['index']
PID = str(os.getpid())
PROTOCOL = 'UDP'
RESULT_PATH =args['result']

start_time = time.time()
client_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
print(f"Requesting server to send the file with index {int(INDEX)}")
client_socket.sendto((bytes(INDEX,'utf-16')),(IP_ADDRESS,PORT))
data,address = client_socket.recvfrom(200)

book_and_size = data.decode('utf-16')
book_and_size_list =re.split('(\d+)',book_and_size)
book = book_and_size_list[0]
book_size = book_and_size_list[1]
print(f"Started receiving {book}")

full_message = bytes('','utf-16')

while True:

    try:
        data,address = client_socket.recvfrom(BUFFER)
        full_message += data
        print(f"Time Elapsed: {time.time() - start_time}")
        client_socket.settimeout(0.1)

    except  socket.timeout:
        print(f"Successfully Recieved {book} using UDP protocol with {BUFFER}
size")
        output_path = './received_files/'+book+'_'+PROTOCOL+'_'+PID+'.txt'
        write_file =open(output_path,"w")
```

```
        write_file.write(full_message.decode('utf-16'))
        final_time = time.time() - start_time

        size_of_file = os.path.getsize(output_path)
        through_put = size_of_file/final_time
        dictionary =
{"Book_Name":book,"Protocol":PROTOCOL,"PID":PID,"Buffer_Size":BUFFER,"Total_Ti
me_Taken":final_time,"Original
Size":book_size,"Size_of_the_file_created":size_of_file,"Throughput":through_p
ut}

        with open(RESULT_PATH, 'r+') as f:
            if len(f.read()) == 0:
                f.write('['+json.dumps(dictionary))
            else:
                f.write(',\n' + json.dumps(dictionary))

        client_socket.close()
        break
```

Part 2: Experiments

A,B and C:

All the experiments are done using the file named les_miserables which had a size of 3.4 MB.

	TCP Results	UDP Results
1 Bytes	I got a unicode Decode error. Since I was reading it as string and encoding it using 'utf-16' to send to the client. The error arises since my text file had characters which needs to be send as two bytes (latin or greek letters) and it created an error. However, the results can be predicted by observing the table. If the error hadn't happened: The time taken would be more for 1 byte. The throughput would be lower. And in case of UDP, the lossed words would be higher.	
32 Bytes	A. Time taken:329.961260 s	A. Time taken: 0.202123 s
	B. Throughput: 10.01777 kbps	B. Throughput: 217.6542 Kbps
	C. Word Count: 5,68,749	C. Word Count: 6,400
	D. Remarks: No packet loss	D. 5,24,734 words lossed
1 KB	A. Time taken:11.768550 s	A. Time taken: 0.271623 s
	B. Throughput: 280.8738 kbps	B. Throughput:910.7893 Kbps
	C. Word Count: 5,68,749	C. Word Count: 42,403
	D. Remarks: No packet loss	D. 3,21,336 words lossed
32 KB	A. Time taken: 0.551063 s	A. Time taken:0.271964 s
	B. Throughput: 5998.367 kbps	B. Throughput:1195.646 Kbps
	C. Word Count: 5,68,749	C. Word Count: 55,335
	D. No packet loss	D. 2,43,554 words were lossed.

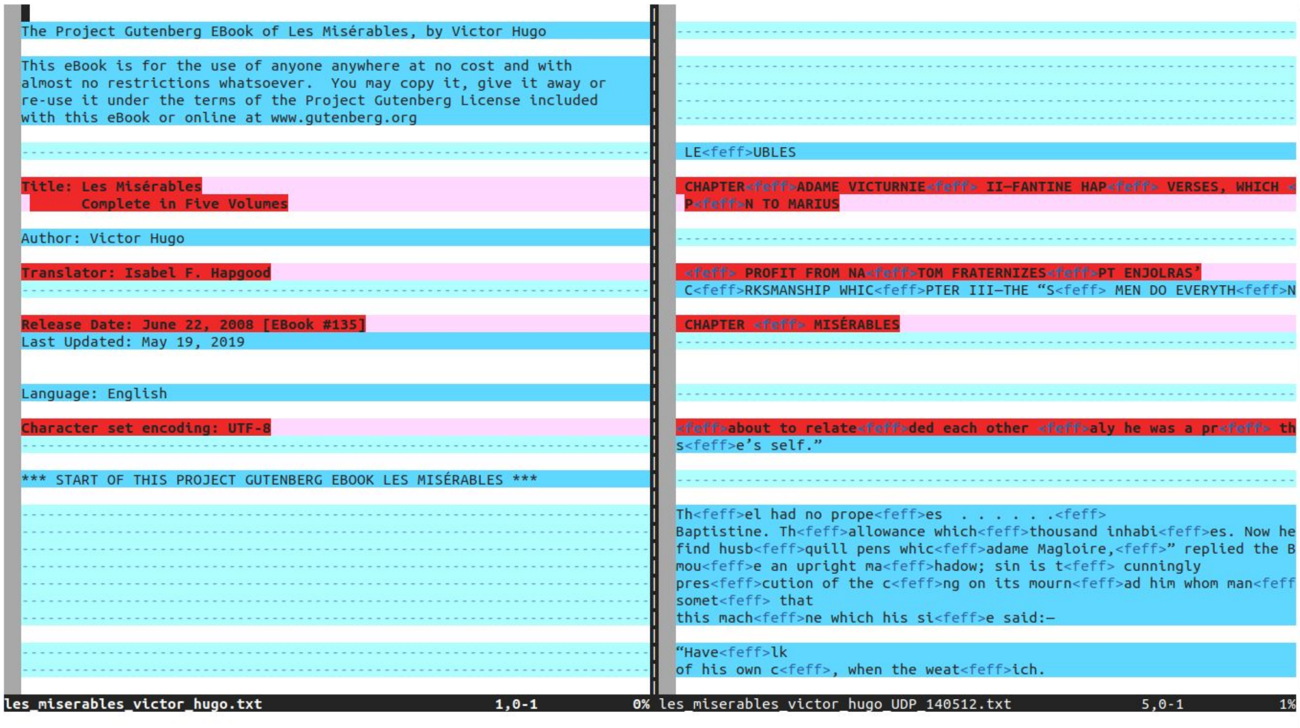
64 KB	A. Time taken:0.385094 s	A. Time taken: 0.165617 s
	B. Throughput: 8583.555 kbps	B. Throughput: 1992.097Kbps
	C. Word Count: 5,68,749	C. Word Count: 73,808
	D. Remarks: No packet loss	D. 1,37,759 words were lossed.

Commands used for finding the difference:

1. Word count:
   $ cat {output_path} | wc -w

2. To find exactly what words were missing[1].
   $ vim -d <original file> <received file>

[1] "Comparing two files in linux terminal - Stack Overflow." 25 Jan. 2013, https://stackoverflow.com/questions/14500787/comparing-two-files-in-linux-terminal. Accessed 28 Oct. 2020.



An example of vim-d

Remarks:

1. Throughput of TCP increases significantly in TCP with increase in BUFFER size when compared to that of UDP. Though, throughput UDP is also increased, but the increase is very less.
2. More words were lossed in UDP protocol and there were no packet loss in TCP as expected. Since TCP provides reliable data transfer.
3. With increase in BUFFER size, the number of words lossed decreases in case of UDP.

D, F and G:

The BUFFER size is fixed to 32 bytes. And the largest file is chosen. In our case, the largest file is Les Miserables by Victor Hugo.

D.

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Word_Count	Nagle
0	les_miserables_victor_hugo	TCP	12410	32	296.752178	12133.899151	569408	Disabled only on Client
1	les_miserables_victor_hugo	TCP	14949	32	295.924002	12167.857208	569408	Disabled only on Server
2	les_miserables_victor_hugo	TCP	12695	32	286.625585	12562.594526	569408	Enabled on Both
3	les_miserables_victor_hugo	TCP	14015	32	276.645680	13015.786093	569408	Disabled on Both

Analysis: The main impact is reflected in the throughput and the total time taken for the file transfer.

- More throughput is obtained when Nagle's Algorithm is disabled on both client and server.

E.

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Word_Count	Delayed Ack
0	les_miserables_victor_hugo	TCP	8843	32	297.441493	12105.779077	569408	Disabled only on Client
1	les_miserables_victor_hugo	TCP	10390	32	293.612989	12263.629807	569408	Disabled on Both
2	les_miserables_victor_hugo	TCP	8560	32	290.174750	12408.939792	569408	Enabled on Both
3	les_miserables_victor_hugo	TCP	9888	32	266.980675	13486.972444	569408	Disabled only on Server

- More throughput is obtained when Delayed Ack is Disabled only on Server

F.

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Word_Count	Nagle and Delayed Ack
0	les_miserables_victor_hugo	TCP	19356	32	292.496484	12310.442003	569408	Enabled on Both
1	les_miserables_victor_hugo	TCP	20457	32	291.396158	12356.926821	569408	Disabled on Server
2	les_miserables_victor_hugo	TCP	19663	32	290.988906	12374.220896	569408	Disabled on Client
3	les_miserables_victor_hugo	TCP	20906	32	289.390486	12442.568689	569408	Disabled on both

- More throughput is obtained when Naggle and Delayed Acks are disabled on both server and client side.

G.

The best throughput is obtained when 64KB buffer is used.

Experiment D

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Word_Count	Nagle
0	les_miserables_victor_hugo	TCP	22210	65536	0.730155	4.931501e+06	569408	Disabled on Both
1	les_miserables_victor_hugo	TCP	22179	65536	0.627564	5.737680e+06	569408	Disabled only on Server
2	les_miserables_victor_hugo	TCP	22030	65536	0.510500	7.053398e+06	569408	Enabled on Both
3	les_miserables_victor_hugo	TCP	22057	65536	0.448986	8.019766e+06	569408	Disabled only on Client

- Best throughput: Disabled Only on Client

Experiment E

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Word_Count	Delayed Ack
0	les_miserables_victor_hugo	TCP	23193	65536	0.602188	5.979464e+06	569408	Disabled on Both
1	les_miserables_victor_hugo	TCP	23106	65536	0.585263	6.152376e+06	569408	Enabled on Both
2	les_miserables_victor_hugo	TCP	23167	65536	0.514671	6.996244e+06	569408	Disabled only on Server
3	les_miserables_victor_hugo	TCP	23135	65536	0.468687	7.682651e+06	569408	Disabled only on Client

- Best throughput: Disabled only on client

Experiment G

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Word_Count	Nagle and Delayed Ack
0	les_miserables_victor_hugo	TCP	24530	65536	0.633880	5.680510e+06	569408	Disabled on Server
1	les_miserables_victor_hugo	TCP	24559	65536	0.524704	6.862459e+06	569408	Disabled on both
2	les_miserables_victor_hugo	TCP	24146	65536	0.515764	6.981419e+06	569408	Disabled on Client
3	les_miserables_victor_hugo	TCP	24119	65536	0.472421	7.621938e+06	569408	Enabled on Both

- Best Throughput: Enabled on Both

Summary Table:

Cases with best throughput:

	Nagle Algorithm	Delayed Ack	Nagle and Delayed Ack
32 Bytes	Disabled on both Client and Server	Disabled only on Server	Disabled on Both
64 KB	Disabled only on Client	Disabled only on Client	Enabled on Both

The experiments shows there is an impact of Nagle's algorithm and Delayed Ack. For smaller buffer sizes, disabling Nagle is seen to perform well and for the greater buffer sizes, it enabling Nagle's Algorithm is beneficial. It also agrees with the Nagle Algorithms's use case, in greater buffer size, since it can send big sized packages at a time, it is okay to wait for some time to get the packets filled. While in other case, the waiting either does not have an impact or can impact negatively.

Delayed Ack, the effect is quite controversial since for lower buffer sizes, delayed Ack disabled only on server said has greater throughput. While for higher buffer sizes, when it is disabled only on client has greater throughput.

The experiment tends to give different results when run at different times. The throughputs was seen varying. It can be due to the processor speed and load on the processor while the program was running. The relative order of the best throughput was expected. However, the relative order can also been changing when run at multiple times. It can conclude to two reasons, either the experiment is not a valid experiment or the minute difference in time made it insignificant to notice (Each program would be run after 3 to 4 minutes for lower buffer sizes. The processor speed would vary within these time limits.).

4. Modify the Server program such that every time it transmits data, it sleeps for about 100 micro seconds. (20 points)

a. TCP: Run two copies of TCP clients(that download two different files), but single TCP server. Observe and list the client process TCP ports, server process tcp ports.
b. Repeat the same with UDP clients.

Present your analysis on the the port allocation on the client/server side in each case. Overall impact on throughput and download completion time in each case. Describe if you find any interesting observations in this experiment.

c.Launch only the Client programs without starting the server programs. What do you observe? Any significant differences for TCP and UDP? And What happens when the Server programs are launched later (say after 30 seconds)

d. Further, check if you can deploy both the TCP and UDP servers and run the TCP and UDP clients at the same time. Possible? Not possible? IF yes why? and if not, why not?

Solution:

Allocating different ports for different clients at the server side will help in connecting multiple clients at the server time. Threading function can be used to run many process simultaneously

Analysis and Observation Table:

1. In the decreasing order of the throughputs:

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Run Type
0	war_and_peace	TCP	121324	1024	1.617032	2.077624e+06	one client connected to server
1	war_and_peace	TCP	121349	1024	1.620137	2.073642e+06	two client connected to single server
2	les_miserables_victor_hugo	TCP	121344	1024	1.635750	2.060079e+06	two client connected to single server
3	les_miserables_victor_hugo	TCP	121305	1024	1.649327	2.043122e+06	one client connected to server
4	les_miserables_victor_hugo	UDP	120475	1024	1.121421	2.289461e+05	two client connected to single server
5	les_miserables_victor_hugo	UDP	120444	1024	1.106875	2.226466e+05	one client connected to server
6	war_and_peace	UDP	120479	1024	1.176312	1.963645e+05	two client connected to single server
7	war_and_peace	UDP	120262	1024	1.173921	1.812958e+05	one client connected to server

2. In the decreasing order of time taken to download.

	Book_Name	Protocol	PID	Buffer_Size	Total_Time_Taken	Throughput	Run Type
0	les_miserables_victor_hugo	TCP	121305	1024	1.649327	2.043122e+06	one client connected to server
1	les_miserables_victor_hugo	TCP	121344	1024	1.635750	2.060079e+06	two client connected to single server
2	war_and_peace	TCP	121349	1024	1.620137	2.073642e+06	two client connected to single server
3	war_and_peace	TCP	121324	1024	1.617032	2.077624e+06	one client connected to server
4	war_and_peace	UDP	120479	1024	1.176312	1.963645e+05	two client connected to single server
5	war_and_peace	UDP	120262	1024	1.173921	1.812958e+05	one client connected to server
6	les_miserables_victor_hugo	UDP	120475	1024	1.121421	2.289461e+05	two client connected to single server
7	les_miserables_victor_hugo	UDP	120444	1024	1.106875	2.226466e+05	one client connected to server

3. The port allocation data at server and client side:

	Client IP_address	Client Port	Server IP	Server port	Protocol	Book_Name	Run Type
0	127.0.0.1	41294	127.0.1.1	12346	TCP	les_miserables_victor_hugo	one client connected to server
1	127.0.0.1	32976	127.0.1.1	12347	TCP	war_and_peace	one client connected to server
2	127.0.0.1	56432	127.0.1.1	12348	TCP	les_miserables_victor_hugo	two client connected to single server
3	127.0.0.1	60262	127.0.1.1	12349	TCP	war_and_peace	two client connected to single server
4	127.0.0.1	48212	127.0.1.1	12346	UDP	war_and_peace	one client connected to server
5	127.0.0.1	60176	127.0.1.1	12347	UDP	les_miserables_victor_hugo	one client connected to server
6	127.0.0.1	50843	127.0.1.1	12348	UDP	les_miserables_victor_hugo	two client connected to single server
7	127.0.0.1	44979	127.0.1.1	12349	UDP	war_and_peace	two client connected to single server

Remarks:

- Effect on throughput:
    - UDP servers: Adding Multiple Clients at the same time have increased throughputs as well as increase download time.
    - TCP servers: Throughput for multiple clients is lesser than when a single client is connected although the difference is very small.
- Effect On Download Time:
    - UDP Server: The time taken to download is higher when multiple clients are connected at the same time. The increase in download time may be due to less packet loss at that time of connection. Since, results of UDP experiments tend to vary greatly because of the packet loss.
    - TCP server: The time taken to download is lesser when a single client is connected to the server
- Port Allocation:
    - There wasn’t much of a difference in server side port allocation as it is programmed by us. However differences can be noticed in the client side port allocation.

- On the client side, the ports were randomly allocated but the randomness seems to exist in both the TCP and UDP servers. However, a significant difference can be noticed if the program is written in a different way. Here, inorder to allow multiple clients, whenever a new client comes, a new port is allocated at the server side. And that port is not reused unless, the server is restarted.

  But, if we program in such a way that, the server side port is made constant and clients are made to connect one after another, a good difference can be noticed. In TCP connection, clients gets nearby ports for example, if an allocated port is 55500 for one client, then the next client might get a port number within a range of 10 (55501 -55510). In case of UDP, it would be very random in that case also.

C. **Launching Only Client Programs without Server Programs:**

- TCP :

  We will get a connection refused error. Since we need a server to establish the connection in case of TCP.

```
(base) shamir:4CD$ python tcp_client1.py -i 5 -b 1024 -r ./results/tcp_warAndPeace_single.json
Traceback (most recent call last):
  File "tcp_client1.py", line 46, in <module>
    client_socket.connect((IP_ADDRESS,PORT_NO))
ConnectionRefusedError: [Errno 111] Connection refused
(base) shamir:4CD$
```

- UDP:

  The program runs. Since it does not have to make a handshake to send and receive data. It waits for the server to turn on.

```
(base) shamir:4CD$ python udp_client1.py -i 5 -b 1024 -r ./results/tcp_warAndPeace_single.json
Requesting server for a port
```

Launching Server after 30s:

- TCP : No effect, since the TCP program is already closed.
- UDP: It will start receiving the data.

D. Yes, it is possible to Deploy both TCP and UDP, if we are using different ports for each client server pair. Since each connection is not related to one another, there is no hindrance caused.

The two servers are called simultaneously. Both are waiting for clients.

The client and server programs are called. The unique client and server ports are allocated for each case. And it runs simultaneously.