

Part 1

Question 1

Core Idea

We first create a standard UDP socket at both the client and server end with a localhost port and IP address. Once this is done, we start transmitting the packets across the connection.

The server runs followed by the client. The client on execution asks for the necessary user input which is given interval, number of echo messages, and given packet size as command-line arguments.

As we need to keep the buffer size the same as the packet size, we first send a message from the client to the server to update the buffer size to the packet size. This input value is also transferred as a packet to the server which then updates the buffer size.

Once the buffer size is updated, we can start transmitting packets of fixed buffer size from the client to the server and vice-versa.

The client on sending packets marks it with a timestamp and the server receives the packet. It sends a reply against that packet like an acknowledgement that is captured by the client. The difference of timestamps of receipt and sending give the Round-Trip Time (RTT) of the packet, which is displayed.

For the client message to the server, we send a randomly generated string of random length. The server for the acknowledgement sends a randomly generated string of random length. This is very handy as it can help distinguish each packet from another packet as encoded string sent across the connection will be different. It also eliminates the possibility of the same packet being repeatedly transmitted.

We need to send every echo client message after a fixed interval of time. As the RTT of a packet is generally very small, the previous packet returns before the next packet being sent. Thus, to keep the interpacket interval consistent we add a delay between the next interval and the currently transmitted packet's RTT. This ensures that the packets are transmitted only after the interpacket interval has passed.

At the end of packet transmission, we can use this to compute the Average RTT.

Since we are transmitting packets over localhost, there are no packet drops caused by congestions in the network. To calculate loss, we introduce an artificial packet loss in the network.

To do this, we generate a random number from 1 to 100 and check if it is equal to a fixed constant. Since the number generation is random, it can be equal to any number from 1 to 100 with a fixed probability. So, the probability of that number is equal to 50 is low i.e., $1/100$ as all numbers are generated with equal probability. In such a scenario, a packet is dropped.

This is done in the following code segment:

```
# Generate a random number from one to 100
random_num = randint(1, 100)

# Check any particular random number is a fixed constant
if random_num == 50:

    # Next iteration of the loop
    # Skip sending the reply packet to the client from the server
    continue

# Since we are generating a random number from 1 to 100 and the probability of that number
== 50 is low
# We get the probability of a number in 100 numbers equal to 50 become 1/100
# On average 1 in 100 packets will be dropped, as that particular packet will not be sent from
server to client
```

The server computes a random number on receiving any client packet and drops sending the acknowledgement in the case when that generated number is equal to a fixed constant (50 in our case). On average 1 in 100 packets will be dropped, as that particular packet will not be sent from server to client

As the acknowledgement packet with the message is dropped, it will result in an acknowledgment timeout from the socket. This is the situation of a packet drop. Since the packet is dropped, its RTT is not estimated and it is added to the lost packet count.

After all, packets are transferred, we will get multiple packets drops which are then estimated to compute the loss percentage.

The RTT is only computed for packets that are not dropped and successfully transmitted packets are not counted as lost packets. As we have computed the RTT, Average RTT, and packet lost percentage, all computations between the UDP echo client-server are completed.

After this, the connection is terminated on both ends. As all the packets are transmitted between the echo client-server and all the computations are completed, we can close the sockets.

Codes

UDP_server.py

Import the necessary libraries

from random import choice
from random import randint

from string import ascii_lowercase
from string import digits

Import the necessary functions from the socket library
from socket import AF_INET
from socket import socket
from socket import SOCK_DGRAM

Set the server IP address and port tuple to localhost and arbitrary value respectively
Readjust this value if you want to communicate with another client on the same network
server_IP_address_port = ("127.0.0.1", 20001)

Set the value of the size of the buffer to the default value before it is changed by the user
buffer_size = 1024

Create a UDP datagram socket with any IP address
UDP_server_socket = socket(family = AF_INET, type = SOCK_DGRAM)

Bind to IP address and port
UDP_server_socket.bind(server_IP_address_port)

Appropriate message is displayed
print("UDP server has bound and is waiting for client connection")

Listen for incoming datagrams until the process is not exited
while True:

Split the incoming message from the client into the message and address
msg, addr = UDP_server_socket.recvfrom(buffer_size)

Decode the byte stream to obtain the client message
client_msg = msg.decode()

Appropriate message is displayed
print("Message from the Client is", client_msg)

If the client message is to exit, then we will stop processing datagrams and terminate the process

```
if client_msg == 'exit':
```

The appropriate message to end the connection, sent as a byte stream

```
UDP_server_socket.sendto("Terminated the connection successfully.".encode(), addr)
```

Appropriate message is displayed

```
print("Exiting...")
```

Exit the process of accepting datagrams

```
break
```

The case where the server needs to change the size of the buffer obtained from the client end

C indicates it is a client message for calibrating

```
if client_msg[0] == 'C':
```

Appropriate message is displayed

```
print("The IP Address and port of the Client is", addr)
```

Obtain the updated size of the buffer from the message

```
buffer_size = int(client_msg.split(':')[1])
```

Appropriate message is displayed

```
print("Updated buffer size to", buffer_size, "bytes")
```

Send the appropriate reply message as a byte stream from the server to the client

```
UDP_server_socket.sendto(("Buffer Size Calibrated at Server.").encode(), addr)
```

After calibration is performed at the server, move to the next datagram

```
continue
```

Generate a random number from one to 100

```
random_num = randint(1, 100)
```

Check any particular random number is a fixed constant

```
if random_num == 50:
```

Next iteration of the loop

Skip sending the reply packet to the client from the server

```
continue
```

Since we are generating a random number from 1 to 100 and the probability of that number == 50 is low

We get the probability of a number in 100 numbers equal to 50 become 1/100
On average 1 in 100 packets will be dropped, as that particular packet will not be sent from server to client

Sending a reply to client

The reply is a random string of a random length, num is also randomly generated

It is useful as each reply to the client is a distinct string

`reply = ''.join([choice(ascii_lowercase + digits) for _ in range(random_num)]).encode()`

Send the reply message as a byte stream from the server to the client

`UDP_server_socket.sendto(reply, addr)`

UDP_client.py

Import the necessary libraries

from datetime import datetime as dt

from decimal import Decimal

from random import choice

from random import randint

from string import ascii_lowercase

from string import digits

from time import sleep

Import the necessary functions from the socket library

from socket import AF_INET

from socket import socket

from socket import SOCK_DGRAM

from socket import timeout

Set the server IP address and port tuple to localhost and arbitrary value respectively

Readjust this value if you want to communicate with another client on the same network

server_IP_address_port = ("127.0.0.1", 20001)

Get the necessary inputs from the user

User-defined - Number of echo messages

msg_total = int(input("Enter the number of echo messages to be sent: "))

User-defined - Given interval

interval = Decimal(input("Enter the interval: "))

User-defined - Given Packet Size

As the size of the buffer is the same as the packet size

buffer_size = int(input("Enter the packet size in bytes: "))

Create a UDP socket at the client-side

UDP_client_socket = socket(family = AF_INET, type = SOCK_DGRAM)

Set a timeout after which we move to the next packet in the case of a packet drop

UDP_client_socket.settimeout(1)

```

# To store the Average Round Trip Time value
avg_round_trip_time = 0

# Count of packets that have completed a round trip and not been dropped
packets_success = 0

# Calibration of Buffer Size
calibration_msg = ("C:" + str(buffer_size)).encode()

# Print appropriate message to the user
print("Calibrating the size of the buffer...")

# Appropriate message is sent to the server
UDP_client_socket.sendto(calibration_msg, server_IP_address_port)

# Receive acknowledgement message from the server to verify that the calibration is complete
ack_msg = UDP_client_socket.recvfrom(buffer_size)

# Send echo message to the server using the created UDP socket
# Iterate through each message as they need to be sent one by one
# msg_count is the iterator that stores the number of echo messages sent
for msg_count in range(1, msg_total + 1):

    # Print appropriate message to the user
    print("Sending message for packet number", msg_count, "of size", buffer_size, "bytes")

    # Estimate the current timestamp to put a timestamp of sending on the packet
    # It marks the sending time of the echo message
    send_time_stamp = Decimal(dt.now().timestamp())

    # Send a random message of a random fixed length
    # It is useful as each message to the server is a distinct string
    client_msg = ".join([choice(ascii_lowercase + digits) for _ in range(randint(10,
20))]).encode()

    # Send the message as an appropriate byte stream to the server
    UDP_client_socket.sendto(client_msg, server_IP_address_port)

    # Receive the reply packet from the server
    try:

        # Try to receive message from the server
        server_msg = UDP_client_socket.recvfrom(buffer_size)

```

```

# Unable to receive the packet from the server
# Case of the packet being dropped leading to a socket timeout
except timeout:

    # Print appropriate message to the user
    print("Packet lost...\nSending next packet")

    # Move to the next datagram
    continue

# Estimate the current timestamp to put a timestamp of receiving the packet
# It marks the receiving time of the echo message
receive_time_stamp = Decimal(dt.now().timestamp())

# The difference of the receiving time stamp and sending time stamp gives Round Trip Time
round_trip_time = receive_time_stamp - send_time_stamp

# Print appropriate message to the user
print("Message from Server is", server_msg[0].decode("utf-8"))

# Print appropriate message to the user
print("Round Trip Time for packet", msg_count, "is", round_trip_time, "seconds")

# Every packet Round Trip Time contributes to the Average Round Trip Time
avg_round_trip_time += round_trip_time

# Since packet has a Round Trip Time and has not been dropped
# It has successfully returned
packets_success += 1

# As we need to send echo messages at a given interval within each other
# We hold the echo message till the interval is complete
# If Round Trip Time exceeds interval, send the packet immediately
sleep(float(max(interval - round_trip_time, 0)))

# Count the number of packets lost/dropped
# The difference of the total packets and successfully transferred packets
packets_lost = (msg_total - packets_success)

# Calculate the loss percentage for the dropped packets (in percentage)
packet_loss_percentage = Decimal(packets_lost / msg_total) * 100

# As we need to find the Average Round Trip Time,
# We divide by the total number of packets successfully transferred
avg_round_trip_time /= packets_success

```



```
# Print appropriate message to the user
print("Average Round Trip Time is", avg_round_trip_time, "seconds")

# Print appropriate message to the user
print("The number of packets dropped is", msg_total - packets_success)

# Print appropriate message to the user
print("The packet loss percentage is", packet_loss_percentage, "%")

# Print appropriate message to the user
print("Finished sending, Terminating the connection...")

# Send the message to exit to the server, to terminate the process
UDP_client_socket.sendto('exit'.encode(), server_IP_address_port)

# Receive the termination message from the server
# Indicates that the server has closed the connection
exit_msg = UDP_client_socket.recvfrom(buffer_size)

# Print appropriate message to the user
print(exit_msg[0].decode("utf-8"))
```

Results

Executing Server file

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1$

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 UDP_server.py
UDP server has binded and is waiting for client connection
```

Executing Client file

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 UDP_client.py
Enter the number of echo messages to be sent: 100000
Enter the interval: 0.001
Enter the packet size in bytes: 128

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 UDP_server.py
UDP server has binded and is waiting for client connection
```

We can see that the standard inputs have been asked to which number of packets are 100000, Interval is 0.001 seconds (1 millisecond) and Packet Size is 128 bytes

On seeing the outputs below:

This is an example of a situation where we have done the packet dropped and the next packet is sent. Since it is dropped there is no message from the server and no Round-Trip-Time is calculated

```
Message from server is rx1ws
Round Trip Time for packet 99983 is 0.0002129077911376953125 seconds
Sending message for packet number 99984 of size 128 bytes
Packet lost...
Sending next packet
Sending message for packet number 99985 of size 128 bytes
Message from Server is rx1ws
Round Trip Time for packet 99985 is 0.0005359649658203125000 seconds
```

We can see the exit condition of the packet

```
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1
Round Trip Time for packet 99985 is 0.0005359649456283125000 seconds
Sending message for packet number 99986 of size 128 bytes
Message from Server is 5km09g38gtvl1j
Round Trip Time for packet 99986 is 0.0003740787506103515625 seconds
Sending message for packet number 99987 of size 128 bytes
Message from Server is ud7cqr33lzwtrhaxeckduap78umnu7a7ko4o3w7nof5madxaugpa9cx10kn6e1k1dx090b2appu6z3sltpgmk
Round Trip Time for packet 99987 is 0.0003678798675537109375 seconds
Sending message for packet number 99988 of size 128 bytes
Message from Server is
Round Trip Time for packet 99988 is 0.0003678798675537109375 seconds
Sending message for packet 99989 is w5fynf194kqysr14
Message from the Client is pxog31eb43xp
Round Trip Time for packet 99989 is 0.0003678798675537109375 seconds
Sending message for packet 99990 is 91v9ou4q8d5jm
Message from the Client is flm4ixzy39u
Round Trip Time for packet 99990 is 0.0003678798675537109375 seconds
Sending message for packet 99991 is xdpkojhs2j7p
Message from the Client is qd106y95tdnmt4aqm73
Round Trip Time for packet 99991 is 0.0003678798675537109375 seconds
Sending message for packet 99992 is shdzubfjmje
Message from the Client is 5lbjslsaxr
Round Trip Time for packet 99992 is 0.0003678798675537109375 seconds
Sending message for packet 99993 is 49jn6o0m0lwnu83xd4q
Message from the Client is 9tz3skbbwni2x3
Round Trip Time for packet 99993 is 0.0003678798675537109375 seconds
Sending message for packet 99994 is 514z3nz2zzlwyoiaq3
Message from the Client is ybxq6789Fhyrlrphq
Round Trip Time for packet 99994 is 0.0003678798675537109375 seconds
Sending message for packet 99995 is q62lumzprg
Message from the Client is kkcijung3cn1b
Round Trip Time for packet 99995 is 0.0003678798675537109375 seconds
Sending message for packet 99996 is vs8oe3sof
Message from the Client is bgntjb5igxjy5
Round Trip Time for packet 99996 is 0.0003678798675537109375 seconds
Sending message for packet 99997 is 7vaweup0zqmqy
Message from the Client is 665q6ad1w8b81wb
Round Trip Time for packet 99997 is 0.0003678798675537109375 seconds
Sending message for packet 99998 is utmgjlm1w0b3j
Message from the Client is 6xw6b25x1w67xtm
Round Trip Time for packet 99998 is 0.0003678798675537109375 seconds
Sending message for packet 99999 is baew2ukhb582cncvout
Message from the Client is ptfayjunh12cu0zheh
Round Trip Time for packet 99999 is 0.0003678798675537109375 seconds
Sending message for packet 100000 is 1zt0i8okbz
Message from the Client is 49hfpw16r2cpct7L5
Round Trip Time for packet 100000 is 0.0003678798675537109375 seconds
Sending message for packet 100001 is 8qzkljiorqrbz2f2ikst
Message from the Client is kim7pmridc40qlti9zt
Round Trip Time for packet 100001 is 0.0003678798675537109375 seconds
Sending message for packet 100002 is jqu3jphyxsc8npz3lyb
Message from the Client is exit
Round Trip Time for packet 100002 is 0.0003678798675537109375 seconds
Sending message for packet 100003 is Exiting...
Message from the Client is
Round Trip Time for packet 100003 is 0.0003678798675537109375 seconds
Sending message for packet 100004 is mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$
Message from the Client is packet number 99999 of size 128 bytes
Round Trip Time for packet 100004 is 0.0003678798675537109375 seconds
Sending message for packet 100005 is 7ccskt7l1mx10k5Shdpejgy1u96vanz0w1rua12rvs
Message from the Client is 0.000205039978802724375 seconds
Round Trip Time for packet 100005 is 0.000205039978802724375 seconds
Sending message for packet 100006 is 11k0jdtd0p9jddkimbeqkfmdoza01oafdd6dq1vpek3o5145t8du4n6spglx13edjuj0bvr0ws
Message from the Client is 0.000257968902587890625 seconds
Round Trip Time for packet 100006 is 0.000257968902587890625 seconds
Average Round Trip Time is 0.0002782072482137914179344697696 seconds
The number of packets dropped is 977
The packet loss percentage is 0.9769999999999999171218512117 %
Finished sending, Terminating the connection...
Terminated the connection successfully.
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$
```

If we look at the larger window which is the client execution, we can see the message from the server is a random string of random length. This distinctly identifies each packet transmitted.

The packet number of each transferred packet is displayed along with its Round-Trip-Time below. The size entered by the user as an input is also displayed.

In the end, we can see that the number of packets dropped is shown with the loss percentage of packets.

The algorithm randomly drops a packet with a probability of 1/100. We select a random number between 1 and 100 and check if it is equal to a fixed constant.

Since the packet drop is set to almost 1 drop in 100 drops, we can see that in about 100000 packets we get about 977 drops giving 0.977 %, very close to 1 % in accordance with the behaviour of the described algorithm.

After this, the connection is terminated on both ends. This happens when the transmission of all the packets is exhausted. An “exit” message is transferred (can be seen, the last message to be sent from the client to server is “exit”), which terminates the open sockets on both the client and server-side.

```

r is Message from the Client is kim7pmridc40qlti9zt
r pa Message from the Client is jq3jphyxsck8npz3lyb
r pa Message from the Client is exit
r is Exiting...
r pa mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$
r packet number 99999 of size 128 bytes

```

For a smaller range of packets:

```

mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 UDP_client.py
Enter the number of echo messages to be sent: 10
Enter the interval: 0.001
Enter the packet size in bytes: 128
Calibrating the size of the buffer...
Sending message for packet number 1 of size 128 bytes
Message from Server is pj6kid3c0softqp14
Round Trip Time for packet 1 is 0.000317096710205078125 seconds
Sending message for packet number 2 of size 128 bytes
Message from Server is uisjypeow9aba3euf6lcmoimfr7ovren1
Round Trip Time for packet 2 is 0.000317096710205078125 seconds
Sending message for packet number 3 of size 128 bytes
Message from Server is 0hecqh06m
Round Trip Time for packet 3 is 0.000317096710205078125 seconds
Sending message for packet number 4 of size 128 bytes
Message from Server is i778xl8sz
Round Trip Time for packet 4 is 0.000317096710205078125 seconds
Sending message for packet number 5 of size 128 bytes
Message from Server is au5strc1l
Round Trip Time for packet 5 is 0.000317096710205078125 seconds

```

This image is to see that what happens on the immediate execution of the client file (about 1 second). The user gives a fixed packet size, so this value is transmitted to the server as a message prefixed with a character “C”. This informs the server to update the size of the buffer to the input packet size.

Once this is updated, we can see that all packets are transmitted with fixed buffer size.

Question 2

Core Idea

The functionality to be implemented in this question is an extension of the functionality of Question 1. All the programming paradigms applied in Question 1 are also used here.

The code for server (IPERF_server.py is the same as server.py). This part has been explained in Question 1 with a full description. (Please refer to the above explanation).

We first create a standard UDP socket at both the client and server end with a localhost port and IP address. Once this is done, we start transmitting the packets across the connection.

The server runs followed by the client. The client on execution asks for the necessary user input which is given interval, number of echo messages, and given packet size as command-line arguments.

As we need to keep the buffer size the same as the packet size, we first send a message from the client to the server to update the buffer size to the packet size. This input value is also transferred as a packet to the server which then updates the buffer size.

Once the buffer size is updated, we can start transmitting packets of fixed buffer size from the client to the server and vice-versa.

The client on sending packets marks it with a timestamp and the server receives the packet. It sends a reply against that packet like an acknowledgement that is captured by the client. The difference of timestamps of receipt and sending give the Round-Trip Time (RTT) of the packet, which is displayed.

The server has the functionality to drop packets (acknowledgements of the client packets are not sent), updating the buffer size on receipt of the user input followed by an exit mechanism to close the connection when required.

This is done in the following code segment for packet drop in the IPERF_server.py

```
# Generate a random number from one to 100
random_num = randint(1, 100)

# Check any particular random number is a fixed constant
if random_num == 50:

    # Next iteration of the loop
```

```
# Skip sending the reply packet to the client from the server  
continue
```

```
# Since we are generating a random number from 1 to 100 and the probability of that number  
== 50 is low  
# We get the probability of a number in 100 numbers equal to 50 become 1/100  
# On average 1 in 100 packets will be dropped, as that particular packet will not be sent from  
server to client
```

It is the same as the code segment of Question (Please refer to the details there)

Moving onto the main extension which is done to add the IPERF functionality. We are calculating the effective throughput and average delay per second and plotting it. We also introduce a mechanism to reduce the inter-packet interval to increase the rate of sending packets.

To reduce the interpacket interval, we have defined a function that takes the current interval value and returns 90% of its value. This is done by computing 10% of its value (Dividing by 10) and subtracting this value from it. This new value is then returned from the function.

Code segment for the user-defined function:

```
# A function that reduces the interval to 90 % of its initial value  
def new_interval(interval_value):  
  
# Calculate 10 % of the interval value  
to_be_lost = Decimal(interval_value / 10)  
  
# Remove it from the current value  
new_interval_value = Decimal(interval_value - to_be_lost)  
  
# As 10 % is removed from the initial interval, 90 % is left  
return new_interval_value
```

After the transmission of a packet, we reduce the interpacket interval by directly calling the function. The code segment is:

```
# As packet's Round-Trip Time is estimated, we need to change the interpacket interval  
# Setting the current interval to the new interpacket interval, for the next packet  
# Above user-defined function specifies the change in this interval size  
interval = new_interval(interval)
```

To calculate the effective throughput and average delay per second, we use the approach of total successful packets transmitted in one second. We send the messages iteratively till all the messages are exhausted. Inside this iteration, we try to send the maximum possible messages in one second.

We run a loop that terminates the moment one second is over i.e., the difference of the end and starts timestamps are exactly one second. Before sending the current packet, first, it is checked that we don't have to wait for the previous interpacket interval to end. The interpacket interval may be more than one second (depending on the size of the user input), the delay will carry over to the next second.

For this, we wait for the delay to complete in the current second after which the packet is transmitted. Sometimes if the interval is very high (say 5 seconds), we need to wait for the whole second for the delay and in fact, multiple seconds before sending the next packet.

In such a situation, the client waits for the necessary time to pass (be it 5, 50, or 500 seconds) and then transfers the next packet. The client might have multiple seconds without any packet transfer, so an appropriate message is displayed to the user.

Once all delays are over, we transmit the packet with a random string (same as question 1) and wait for an acknowledgement. We have also included the situation of packet drop at the server side as briefed above. Such a situation will lead to a timeout forcing the client to move to the transmission of the next packet based on the interval value.

If the packet acknowledgement is received successfully then we calculate the Round-Trip Time similar to Question 1. To calculate the effective throughput and average delay only the packets that have an RTT are considered. Packets whose acknowledgements are dropped will not be considered as throughput and average delay cannot be estimated for such packets.

So, the successful packets are counted into the throughput and delay values. Since we are having a direct client-server connection there are no intermediate delays that can be added to the delay value being computed after the RTT.

We cannot have a queuing delay, propagation delay as we don't have a router and the packet reaches after multiple hops or a physical cable to transmit the packets. It is happening on the same system. Hence, for the delay, we are only including the successful packet's RTT.

Once all these computations are performed, we move to the transmission of the next packet. Since we need to reduce the interpacket interval, we use the above user-defined function and reduce the interval.

After this, we need to wait for the interpacket interval to complete before transmitting the next packet. If the wait is completed within the same second, we transmit the next packet within the same second and add it to the throughput and delay calculations.

If the delay exceeds the time left in the current second, it is carried over to the next second and the necessary steps are taken to wait for the delay to complete before sending the next packet. The client will wait for the delay to get over, no matter how much carry-over is present.

At the end of the second, the effective throughput and average delay are calculated based on the values obtained in the particular time scale loop iteration. The throughput depends on the number of success packets, as the information of packet size and completed time is already present. For a successful packet, the throughput is multiplied by TWO as the packet goes and comes back for the RTT to be calculated so the bandwidth is used twice leading to twice the data transfer.

For the average delay, the total delay is calculated divided by the successful packets (it is an average value) within the same second. However, this can lead to a `ZeroDivisionError` where there is no successful packet transferred. This could be possible if we had to wait for the whole second for the delay to complete.

In such a scenario, the average delay as no delay can be estimated as we do not have any packet to perform the estimation and computation on. Once these values are computed, they are stored as we need to plot graphs with the specified time scale.

After the transmission of all the packets, the values of the effective throughput and average delays are plotted using the requisite libraries and the results are stored. With this, we are done with the extension of the UDP echo client to build an IPERF architecture.

After this, the connection is terminated on both ends. As all the packets are transmitted between the IPERF echo client-server and all the computations are completed, we can close the sockets.

Codes

IPERF_server.py

Import the necessary libraries

from random import choice
from random import randint

from string import ascii_lowercase
from string import digits

Import the necessary functions from the socket library
from socket import AF_INET
from socket import socket
from socket import SOCK_DGRAM

Set the server IP address and port tuple to localhost and arbitrary value respectively
Readjust this value if you want to communicate with another client on the same network
server_IP_address_port = ("127.0.0.1", 20001)

Set the value of the size of the buffer to the default value before it is changed by the user
buffer_size = 1024

Create a UDP datagram socket with any IP address
UDP_server_socket = socket(family = AF_INET, type = SOCK_DGRAM)

Bind to IP address and port
UDP_server_socket.bind(server_IP_address_port)

Appropriate message is displayed
print("UDP server has binded and is waiting for client connection")

Listen for incoming datagrams until the process is not exited
while True:

Split the incoming message from the client into the message and address
msg, addr = UDP_server_socket.recvfrom(buffer_size)

Decode the byte stream to obtain the client message
client_msg = msg.decode()

Appropriate message is displayed
print("Message from the Client is", client_msg)

If the client message is to exit, then we will stop processing datagrams and terminate the process

```
if client_msg == 'exit':
```

The appropriate message to end the connection, sent as a byte stream

```
UDP_server_socket.sendto("Terminated the connection successfully.".encode(), addr)
```

Appropriate message is displayed

```
print("Exiting...")
```

Exit the process of accepting datagrams

```
break
```

The case where the server needs to change the size of the buffer obtained from the client end

C indicates it is a client message for calibrating

```
if client_msg[0] == 'C':
```

Appropriate message is displayed

```
print("The IP Address and port of the Client is", addr)
```

Obtain the updated size of the buffer from the message

```
buffer_size = int(client_msg.split(':')[1])
```

Appropriate message is displayed

```
print("Updated buffer size to", buffer_size, "bytes")
```

Send the appropriate reply message as a byte stream from the server to the client

```
UDP_server_socket.sendto(("Buffer Size Calibrated at Server.").encode(), addr)
```

After calibration is performed at the server, move to the next datagram

```
continue
```

Generate a random number from one to 100

```
random_num = randint(1, 100)
```

Check any particular random number is a fixed constant

```
if random_num == 50:
```

Next iteration of the loop

Skip sending the reply packet to the client from the server

```
continue
```

Since we are generating a random number from 1 to 100 and the probability of that number == 50 is low

We get the probability of a number in 100 numbers equal to 50 become 1/100
On average 1 in 100 packets will be dropped, as that particular packet will not be sent from server to client

Sending a reply to client

The reply is a random string of a random length, num is also randomly generated

It is useful as each reply to the client is a distinct string

`reply = ''.join([choice(ascii_lowercase + digits) for _ in range(random_num)]).encode()`

Send the reply message as a byte stream from the server to the client

`UDP_server_socket.sendto(reply, addr)`

IPERF_client.py

Import the necessary libraries

from datetime import datetime as dt

from decimal import Decimal

from matplotlib.pyplot import grid

from matplotlib.pyplot import plot

from matplotlib.pyplot import show

from matplotlib.pyplot import subplot

from matplotlib.pyplot import title

from matplotlib.pyplot import xlabel

from matplotlib.pyplot import ylabel

from random import choice

from random import randint

from string import ascii_lowercase

from string import digits

from time import sleep

Import the necessary functions from the socket library

from socket import AF_INET

from socket import socket

from socket import SOCK_DGRAM

from socket import timeout

Set the server IP address and port tuple to localhost and arbitrary value respectively

Readjust this value if you want to communicate with another client on the same network

server_IP_address_port = ("127.0.0.1", 20001)

Get the necessary inputs from the user

User-defined - Number of echo messages

msg_total = int(input("Enter the number of echo messages to be sent: "))

User-defined - Given interval

interval = Decimal(input("Enter the interval: "))

User-defined - Given Packet Size

As the size of the buffer is the same as the packet size

buffer_size = int(input("Enter the packet size in bytes: "))

```

# Create a UDP socket at the client-side
UDP_client_socket = socket(family = AF_INET, type = SOCK_DGRAM)

# Set a standard timeout after which we move to the next packet in the case of a packet drop
UDP_client_socket.settimeout(1)

# To store the Average Round Trip Time value
avg_round_trip_time = 0

# Count of packets that have completed a round trip and not been dropped
packets_success = 0

# Calibration of Buffer Size
calibration_msg = ("C:" + str(buffer_size)).encode()

# Print appropriate message to the user
print("Calibrating the size of the buffer...")

# Appropriate message is sent to the server
UDP_client_socket.sendto(calibration_msg, server_IP_address_port)

# Receive acknowledgement message from the server to verify that the calibration is complete
ack_msg = UDP_client_socket.recvfrom(buffer_size)

# A function that reduces the interval to 90 % of its initial value
def new_interval(interval_value):

    # Calculate 10 % of the interval value
    to_be_lost = Decimal(interval_value / 10)

    # Remove it from the current value
    new_interval_value = Decimal(interval_value - to_be_lost)

    # As 10 % is removed from the initial interval, 90 % is left
    return new_interval_value

# To store the Average Throughput values after each second
AVG_THROUGHPUT = []

# To store the Average Delay values after each second
AVG_DELAY = []

# To store the Time Scale each second
TIME = []

```

To store the number of seconds passed

`count_seconds = 0`

It will essentially store the amount of delay time before transmission of a packet in the next second

Since there is no delay initially, set to ZERO

`delta = 0`

Send the message to the server using created UDP socket iteratively

This loop will iterate over time in seconds and will increment time till all packets are not transferred

`while msg_total > 0:`

Since we have no more packets to be transferred, we can stop the packet transfer

`if msg_total == 0:`

Stop the packet transfer

`break`

Increment time as we have moved to the next second to transmit the packets

`count_seconds += 1`

Print appropriate message to the user

`print("Second", count_seconds, ":")`

To store the Average Delay Time value

`avg_delay = 0`

Count of packets that have completed a round trip and not been dropped

`packets_success = 0`

This stores the time from where one second starts to send the loop

`second_start = Decimal(dt.now().timestamp())`

Send each packet of that particular second one after the other iteratively

Transmit packets as long as the second is not over

Difference of current timestamp and starting timestamp must not exceed ONE

`while Decimal(dt.now().timestamp()) - second_start <= 1:`

The condition where the delay to send the current packet exceeds ONE

`if delta > 1:`

As the delay was more than one second, we need to sleep for the whole second

`tosleep = 1`

```

    # Reduce the delay by one second
    delta -= 1

    # As we wait for the whole second on account of the delay, no packet is sent in this
second
    # Print appropriate message to the user
    print("No packet will be sent in this second")

else:

    # Set the sleep value to the delay
    tosleep = Decimal(delta)

    # Wait/Sleep for the delay to complete
    sleep(float(tosleep))

    # Since we have no more packets to be transferred, we can stop the packet transfer
    if msg_total == 0:

        # Stop the packet transfer
        break

    # Since we have skipped the whole second, we cannot transfer any packets
    if tosleep == 1:

        # Stop the packet transfer
        break

    # Send message to server
    # Send a random message of a random fixed length
    # It is useful as each message to the server is a distinct string
    client_msg = ".join([choice(ascii_lowercase + digits) for _ in range(randint(10,
20))]).encode()

    # Send the message as an appropriate byte stream to the server
    UDP_client_socket.sendto(client_msg, server_IP_address_port)

    # Estimate the current timestamp to put a timestamp of sending on the packet
    # It marks the sending time of the echo message
    send_time_stamp = Decimal(dt.now().timestamp())

    # As we have sent a packet, we decrement the packets left to be sent
    msg_total -= 1

```

```

# Receive the reply packet from the server
try:

    # Try to receive message from the server
    server_msg = UDP_client_socket.recvfrom(buffer_size)

    # Print appropriate message to the user
    print("Received message:", server_msg[0].decode())

# Unable to receive the packet from the server
# Case of the packet being dropped leading to a socket timeout
except timeout:

    # Print appropriate message to the user
    print("Packet lost...\nSending next packet")

    # Move to the next datagram
    continue

# Estimate the current timestamp to put a timestamp of receiving the packet
# It marks the receiving time of the echo message
receive_time_stamp = Decimal(dt.now().timestamp())

# The difference of the receiving time stamp and sending time stamp gives Round Trip Time
round_trip_time = Decimal(receive_time_stamp - send_time_stamp)

# Since packet has a Round Trip Time and has not been dropped
# It has successfully returned
packets_success += 1

# Every packet Round Trip Time contributes to the Average Delay in the network
avg_delay += Decimal(round_trip_time)

# As we need to send echo messages at a given interval within each other
# We hold the echo message till the interval is complete
# If Round Trip Time exceeds interval, send the packet immediately
# This will essentially store the delay of the next packet from the previous
sleep_time = Decimal(max(interval - round_trip_time, 0))

# As packet's Round-Trip Time is estimated, we need to change the interpacket interval
# Setting the current interval to the new interpacket interval, for the next packet
# Above user-defined function specifies the change in this interval size
interval = new_interval(interval)

```



```

    # The time which is over in this particular second
    passtime = Decimal(Decimal(dt.now().timestamp()) - second_start)

    # Wait for the inter-packet interval to complete before sending the next packet
    # Sleep for the time which is left to send the next packet

    # The time for this second has already exceeded the deadline
    if passtime > 1:

        # Store the remaining delay to be carried over to the next second
        delta = Decimal(tosleep)

        # Move to the next second immediately
        break

    # The case if the delay is so high that it is moving over to the next second and we have time
    leftover in this second
    elif passtime + sleeptime > 1 and passtime <= 1:

        # Set the sleep value to the rest of the second
        tosleep = Decimal(1 - passtime)

        # Store the remaining delay to be carried over to the next second
        delta = Decimal(sleeptime - (1 - passtime))

    # We have time in the current second to complete the delay
    else:

        # Set the sleep value to the delay
        tosleep = Decimal(sleeptime)

        # Wait/Sleep for the delay to complete
        sleep(float(tosleep))

    # The situation where the socket slept for the whole second and no packet could be transferred
    # Since there is no successful packet whose RTT is calculated successfully, there is no delay
    if packets_success == 0:

        # As there is no RTT of any packet, no delay is estimated due to which average delay stays
        ZERO

        # It is not estimated, so we can safely return ZERO to the user
        avg_delay = 0

```

else:

*# As we need to find the Average Delay,
We divide by the total number of packets successfully transferred*
avg_delay /= packets_success

The Average Throughput is given by the total data transferred by the total time passed (in bytes per second)

The total data transferred is the successful packets transferred multiplied by the data carried by each packet

Since we need to calculate Average Throughput every one second, we don't need to explicitly divide by 1

As the size of the buffer is taken in bytes, no change in the unit is required

Since the same packet is transferred from the client and vice-versa, the data transferred is twice (hence the factor of TWO)

avg_throughput = packets_success * buffer_size * 2

Print appropriate message to the user

print("The Average Throughput is", avg_throughput, "bytes/seconds")

Print appropriate message to the user

print("The Average Delay is", avg_delay, "seconds")

Append for plotting average throughput

AVG_THROUGHPUT.append(avg_throughput)

Append for plotting average delay

AVG_DELAY.append(avg_delay)

Append for plotting time scale

TIME.append(count_seconds)

Plotting the graph

Create TWO Horizontal subgraphs to show both the plots

Create a Subplot for Throughput

subplot(1, 2, 1)

Plot the Average Throughput in bytes per second on the Y-axis, Time scale in seconds on the X-axis

Colour of the plot is Red

plot(TIME, AVG_THROUGHPUT, 'red')

Label the X-axis in the subplot with the Time scale in seconds

xlabel("Time (in seconds)")

```

# Label the Y-axis in the subplot with the Average Throughput in bytes per second
ylabel('Average Throughput (bytes/seconds)')

# Label the graph as Y-axis vs X-axis
title('Average Throughput (bytes/seconds) vs Time (in seconds)')

# Make a grid out of the graph to give a better approximation
grid(True)

# Create a Subplot for Delay
subplot(1, 2, 2)

# Plot the Average Delay values in seconds on the Y-axis, Time scale in seconds on the X-axis
# Colour of the plot is Blue
plot(TIME, AVG_DELAY, 'blue')

# Label the X-axis in the subplot with the Time scale in seconds
xlabel('Time (in seconds)')

# Label the Y-axis in the subplot with the Average Delay in seconds
ylabel('Average Delay (in seconds)')

# Label the graph as Y-axis vs X-axis
title('Average Delay (in seconds) vs Time (in seconds)')

# Make a grid out of the graph to give a better approximation
grid(True)

# Display the graph to the user
show()

# Print appropriate message to the user
print("Graph Plotting is Complete")

# Print appropriate message to the user
print("Finished sending, Terminating the connection...")

# Send the message to exit to the server, to terminate the process
UDP_client_socket.sendto('exit'.encode(), server_IP_address_port)

# Receive the termination message from the server
# Indicates that the server has closed the connection
exit_msg = UDP_client_socket.recvfrom(buffer_size)

```

```
# Print appropriate message to the user  
print(exit_msg[0].decode("utf-8"))
```

Results

We consider 3 situations:

- Without Packet Drop
- With packet drop
- Large delay values, and understand the behaviour of our application.

The situation where packet drop is considered.

The inputs have been entered where the message count is 500, the interpacket interval is 0.5, and the Packet size is 128 bytes.

We can see the message in each second with the effective throughput and average delay values being printed. The exchange of messages is continued between the server and client. The buffer size is also calibrated at the server end in a similar mechanism as Question 1.

```
mastershubham@LAPTOP-8Q155HE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 IPERF_client.py
Enter the number of echo messages to be sent: 500
Enter the interval: 0.5
Enter the packet size in bytes: 128
Calibrating the size of the buffer...
Second 1 :
Received message: xsur76tvxbhjmi
Received message: aya537g9ew08Fjha3h5qpf9ayr
Received message: at
The Average Throughput is 768 bytes/seconds
The Average Delay is 0.0002393722534179687500 seconds
Second 2 :
Received message: 3m5zye8m
Received message: n3ghdwj070uwjulsui9xe1jibqgufowujzfsah5v1pnnuee56qguwLou3433h56f4yyvj57ywdita
The Average Throughput is 512 bytes/seconds
The Average Delay is 0.00023400783538818359375 seconds
Second 3 :
Received message: qvefsy9hrc7kevdykxxy35oudar18ga6cs22cclaysdpttilloes8xkdnuglobh95kklhgn
Received message: xahriu5e0tqnam2oaqvr90hvec6eip1ltig6hh7y9a24jwbjo84noenrucc27y8ap
The Average Throughput is 512 bytes/seconds
The Average Delay is 0.0002830028533935546875 seconds
Second 4 :
Received message: jypajyl1otmq2hpq653c0d05qp5wqpfu6d19bwi138j27reiy67Bhq2kw0
Received message: ubcu0dzu8j14fosp2qhb7z7usujlxt4nqjpb0faxhcocnmh
The Average Throughput is 512 bytes/seconds
The Average Delay is 0.00033748149871826171875 seconds
Second 5 :
Received message: l0uac8oj1lw183gna1vpp2j49tg2ddmvezry5c4cbgao15oclp1xqzajcagzr13
Received message: t12em1f3oz84Bekhh7xzc7c24kqvz2v9vw21j7x9isy2btjc95qh3xx3wvo
Received message: mr4ntd6wdudhjfghtozq2s1qjtcydl19hnh3g6198rzkwnoq79v9y0565mbe3nxuqe
The Average Throughput is 768 bytes/seconds
The Average Delay is 0.000269730885823567708333333333 seconds
Second 6 :
Received message: 9wqo2043vg4uo4dhn0nb3upzmbj28bi6ikqhk
Received message: f15u6u150p32d2aauacvghnuxr29nmpqhpxcao77euz21vknkp13q11zmarw5dsebeacij80hwt73u9acsvd
Received message: u0hr1svhhcv2
Received message: rwt028z0n77u599572K5frfdq1d4mwzbj6272dxa0n2vzklysw255h32n1zotvchn0wbqx464ttpu0r
Received message: qz08z280h0ay00ufad0hqsrema0w
Received message: sefivc9scu1ab1rt4x5omzz13bch3349kwhypfag1o569hc4ih0wov15sz0tqre4deg6gtmc0y2911qoutvey2mw47mc
Received message: ap7
The Average Throughput is 1792 bytes/seconds
The Average Delay is 0.0002678462437220982142857142857 seconds
Second 7 :
Received message: xuddaayz68f79uxkggnrkjtaa
Received message: h1law0g1g1606te3kan4owffhw
Received message: ggs6dl18wgx73uotm18ro3l4q9hfyggq75lu0v59q5h713h1exu659jve1uvgydhnegqtsuqelybh2p
Received message: xtpmxu27jvz9wt8yp3kop2bsvk45fkqfonz1dclhtycbct4vffn6cke1w1lpki3pvvz4xjljqae4n3q1d33og0fquew
Received message: nfwzgbjorgeem37fhw1uh1e1bf6wungbhtlujz14hr43yrt5u5zvw30s5f
Received message: pksfhib4yuoxyso5but2lrtf90um3q71j7m
Received message: 1lgrbh1yckvj6r01y3c8e8exb1dyb5823h76m0ymyck9kg4e16drj1xf7v1m6mm0hft3t

mastershubham@LAPTOP-8Q155HE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 iperf_server.py
UDP server has binded and is waiting for client connection
Message from the Client is C :128
The IP Address and port of the Client is ('127.0.0.1', 57521)
Updated buffer size to 128 bytes
Message from the Client is bny0u54kudowbn
Message from the Client is pui5and4tliujlu6
Message from the Client is d99outhzrxjk5pam0
Message from the Client is 16eneaiug40
Message from the Client is 497mcl6ex
Message from the Client is 47hnc19vmda9
Message from the Client is tfkd5diih0eb3x9d70
Message from the Client is 7xidbv8voe6ja12zy
Message from the Client is 26saecv92xg
Message from the Client is xwp0z6140or2e6v
Message from the Client is cto8hw4qso
Message from the Client is u7lwa5185r
Message from the Client is 5zuJ0g9zb1xppq
Message from the Client is j2yfv3cursct5eft
Message from the Client is p0299d755p1m
Message from the Client is 282paga7oiva515t8cel
Message from the Client is c2pm8a2fpg00r8k2z6t
Message from the Client is 4m6ef6mlwa8rtt9f1mm
Message from the Client is 8rk1cp7dq5
Message from the Client is u0thag8za19twap
Message from the Client is guh080bkcs0u43185v
Message from the Client is c0131xdt5
Message from the Client is 8gpgq421zp98x
Message from the Client is 0fppevankqtx340y
Message from the Client is fb55biez8dxpudroj
```

On zooming in a particular part of the output, we can see the situation of a packet drop. In the case of a packet drop, we see that the socket timeout will cause a large delay to transmit the next packet. A large delay will take the packet to the next second, so we move to the next second to transmit the next packet.

```
Received message: mqkkvq5526he50h8wjmd4vtv8i3ruxca8c0kv141ftsbkoifgfsiwmsswnm87ch2ascafmlqs78g18bzkqyl1z4iqf0ypzxnqqqw
Packet lost...
Sending next packet
The Average Throughput is 16896 bytes/seconds
The Average Delay is 0.0004282322796908291903409090909 seconds
Second 9 :
```

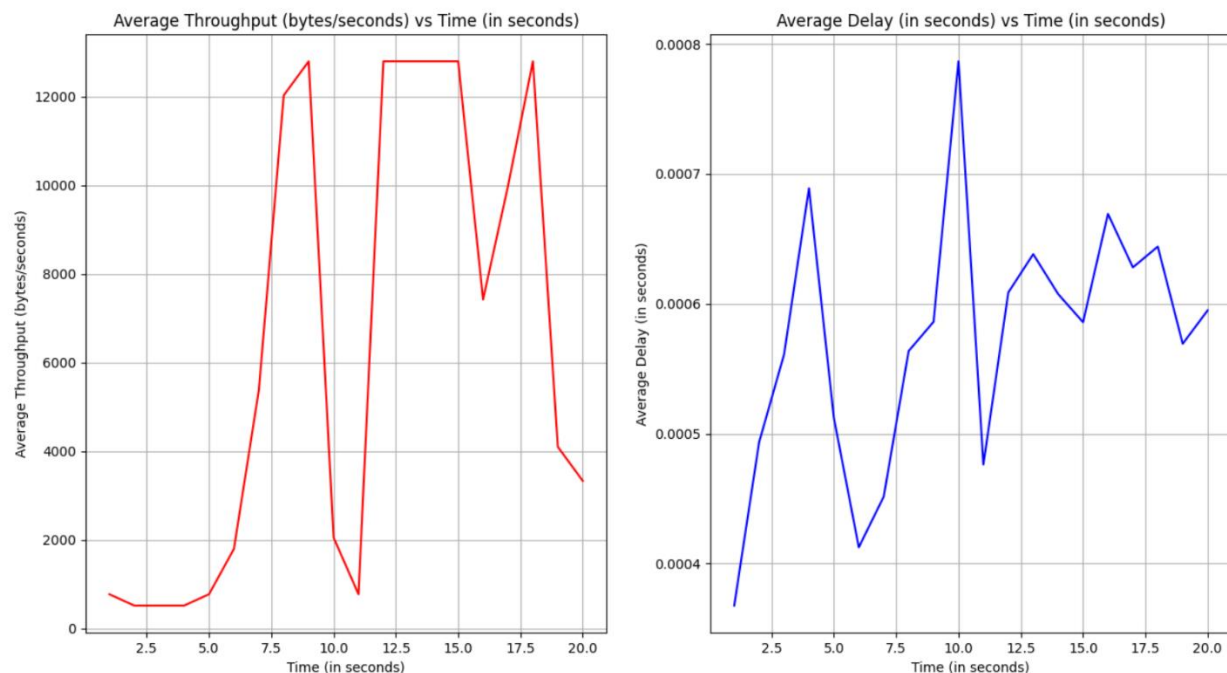
We now plot the graph. If you closely observe the graph, the effective throughput continues to increase after every second due to the reduction in the interpacket interval.

Once it goes up, it never comes down and the threshold is maintained. We can see sharp drops in between due to the occurrence of a packet drop. Since we have a scenario of packet drop, the timeout does not send more packets in the same second reducing effective throughput by a large margin.

In the graph, you can see sharp drops which come back to the same value again, this happens because in one second there was a packet drop whereas there is no packet drop in the next. In the end, the throughput reduces because we are nearing the completion of the packet transfer.

As very few packets remain to be transferred, less data is transferred reducing the effective throughput. We don't use the whole second to transfer packets as the data transfer is complete, so a reduced effective throughput.

If we focus on the average delay the scale of the Y-axis is very small with very low fluctuation. Since, we have to find the average value of the delay if there are low packets transferred it won't impact (Average does not depend on size), there is a very small channel across which the results vary. We can see that in the graph below as well.



In the end, we can see that message transfer is complete and a message that graph plotting is complete is displayed to the user indicating that the computation is over and we exit the application.

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1

```
Received message: jkca0cxwpc1lg5dp94p5e8v94k9y2u5nurmw4hwwcxmjgaq0t8ihsfkkln5tbp8ntqz
Received message: sw14c0g9m
Received message: 30c1u4k4dzk2l4q8ei7mfqzhbwe1m9lhhck86uqdr5j5o
Received message: d8aqo10jw540u86
Received message: 1amdhnpg3ti6xulwjg93lrtcmmn0bi74aic320bn3saf8l1p8vawor4a91vnt8f4b6j457s
Received message: 4txn4pvs7k29airopzlanf
Received message: spd1
Received message: c929ay2s0uxrjeu9w1m83ihb3cc32udozrysw7o19zr5rs1p8vqmrgsq2jfvk4d1acuuw211i9o
Received message: kdtxdro
Received message: rmq2fptv8khqsbwya2lwlqvxe26cnzqvt5
Received message: 1oxppscvugg6f0g72ljlork7xvkdej94r4ss3tvce00o7y6jyyw807a1c2emr13n3v0mvjtpul
Received message: 4f981acmuo2ak8msfg80mm7sprh0d1luxe14g7fto8i52vsthsqk
Received message: p2rpgzj12oj4e0r9t6mwhfmgnlw5scrkte
Received message: 5kxclfb7lppb39p62vso8ju
Received message: sj8wzfr0q65udv24mcqytesge8y6usxg8tta0ri2ma4grq006mf2skynumw4p0hsm2
Received message: wljiw3lgrxw5xljvewmgbpzl
Received message: wrs0
Received message: ry4o8srulnlbbgw73saa3k3qf8wrrou3wrfgnnesy9wntfjs1xu8b5hyppz8dz0538ou9u8mn1s
Received message: omceavchb5dm10tgp60ln4k82be2mf2oo8yf85apsdw830enoxo81j8z
Received message: e47robdpun2b352tkol26s7r1d5oqhne73cgcjlv5jj2v9ztfxs8fvhpxrsg1kjk1q4ky1q7ik
Received message: er65tlbvnqdr1ad2z8vp4zzyhi9v8isyxrfssas8im1an593nw6wqdhxb
Received message: egee2ux368xr70egrpaxg62ij6dlnsaos1ymwuiaf78bovsas
Received message: 46jrlj8xng3oovvye1jvsqedohaek5lrrqrw1fp4bqtpidd5rmttt81hl
Received message: ilsr0lp3hamlxfhnbvtrvkqdfqmdm1smn509izrxdxrmtnkzng
Received message: hahl11492ewve33l4o7kcgz7ausxqlsbf4gk2bxcid4lvksu4yathtuqz6tevooldvrrn944vr1n
Received message: 6b21v6xvqxwf70ppx6a84seij766rxrjic6em6hja3rlxin987nd18paydin8ozjrse5duxjmhuj
Received message: tk119pygavnoh7xdh45apoixsgupmzmunikkcjdrdxftw
Received message: uu3z6oqe0pp17zh5
Received message: eugrxtplwlpocm4wt16ad
The Average Throughput is 28416 bytes/seconds
The Average Delay is 0.0004068022375708227759009009 seconds
Graph Plotting is Complete
Finished sending, Terminating the connection...
Terminated the connection successfully.
```

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1\$

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1

```
Message from the Client is 5csind1jnczf3lne
Message from the Client is 3cmu1i40ay020
Message from the Client is r6saqmyemy7wdafabi
Message from the Client is kna0um3r0xe2vy9zck2
Message from the Client is a3rbbwvhtbtr7n
Message from the Client is yduxysmxcvao
Message from the Client is vz60flwjr97zz
Message from the Client is 09tqhaj78yqc6ajel
Message from the Client is zj4h6frvmgv
Message from the Client is exit
Exiting...
```

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1\$

The situation where there is no packet drop

This is the code section that drops the acknowledgement from the server end, so commenting on it will remove the situation of packet drop.

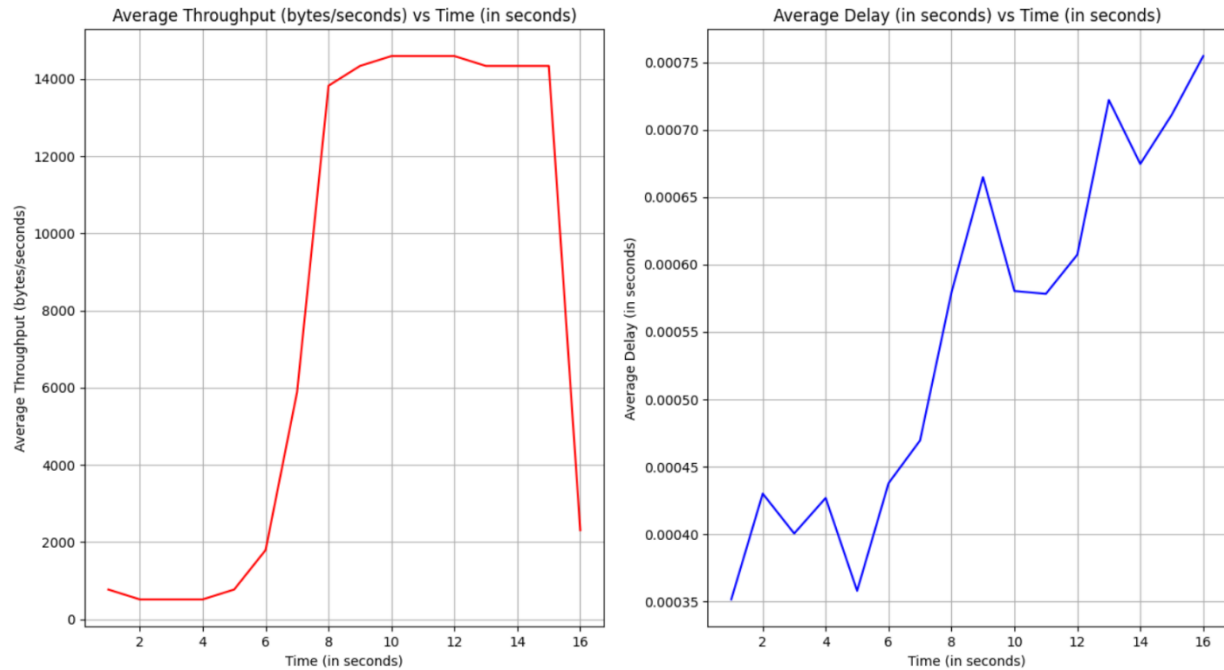
I have commented the section:

```
'''
# Check any particular random number is a fixed constant
if random_num == 50:

    # Next iteration of the loop
    # Skip sending the reply packet to the client from the server
    continue
'''
```

Coming to the results, we can see that there are no sharp drops in between in the case of the effective throughput as there are no packets dropped. The value continues to increase as previously explained and falls in the end as we are nearing completion of the packet transfer.

No abrupt drops are observed due to the elimination of the packet drop scenario. In the case of the average delay, we see that the value goes up by small margins as we have more packets being transferred due to a reduced interval size. However, the fluctuation in the value is very low as the scale of the graph is extremely small.



In the end, we can see that message transfer is complete and a message that graph plotting is complete is displayed to the user indicating that the computation is over and we exit the application.

```

Received message: s1ynxs5my0hxsagcggv14s2rqwhuyt5g219qyiticsp50988y1xbalsrc6mkh40ycvf046alfthv
Received message: 9emzf4145jwvru75hq62b9dlqo4141s7and16fvba8qlwj2feuyaic2vc6w41219qe62xs9ms1
Received message: nu6bvyyhh88j
Received message: he8wo724mfw3093oq7178vtpsm8j1z1zhkek
Received message: n4iwsntx6az00rai2whuy39ouvdsw5dxcgvhcnrtabf3wqh6fghqcvwr85d13cxufe7tss8iup
Received message: q8s0seidmy8cy7ryenip9cjpqlrgcw1vkl526u7xcdh2ugxrfch1fgax3nhnl6e1rdds3x2cycsty
Received message: my86j6f310x8px7x22i3wn6tkfrn1dg35x2cycsty
Received message: pgcuf1kynw8
Received message: f9oz11ggvwrh2g42111t4dkmpr91ef10u7zkyggs26zyfsczvg0umymom8lmbxp1tewenp62p9p
Received message: a4w43t10eeucy2b225wp8lwevy1esxuvjzcfj5262wqhwzwt0379qpvdz17671920171exjkm
Received message: i2nytt80xcz168j8d4jht2322akw00855jtg8y83mbdculw5p0bst9u08czboodhb9h1kaip8
Received message: e399j2gh44wpxakq90s2q9d6mdgi26qnsq4exrh2agua9jegbr76lrgd64it467301jvvpjrp
Received message: ddy561s43ceyx8xdjkw96jzqfkrke6ykv8174gxnt14xdojqc9zciq7w0glw2yfp48i
Received message: m2xxgv8d6x4f2d0ogejd077qlp8b9qd
Received message: 4nrh09usmkd74u
Received message: xr520zy7c0hrr3eywyjdjq1piyuupvv7knyntui
Received message: r6seo8iodf034ss7n7113e0fkixux31p37u
Received message: 3ygv1zctwpwjmd4yfdhgz7h9wi1oe3fr8jq46vg11a2vm2yboa1p0eyabhubbkewy5k
Received message: ehx46vsud3ngcqkgr7rxmw5e1vl1qnfrws90vj38j1omza2dyntwap35z9yuv
Received message: z6vv3pa4xw1el1jvwvph093eayjm
Received message: x603xv7cha66s752wxt5g3jwq6qjq2o0zoi3sj3cxel1g1jix
Received message: 70vuigknh65jthrr5ery3gytzk6nb5t3wq7ifj7hiuxqyygn8dg71205k1dyf1chy2o05kdkbr173ka0wb6mvt6lpekm
Received message: skgn2h61dhst9g24
Received message: mb0e1ux3pxh1h
Received message: mvxmi517ocus10o0t33j61gax0ucp1r9yyel108bjm72thp01sya4nlmmwiapkrms4za1j766swxnmf4cdcy5vnhrtk9g
The Average Throughput is 10752 bytes/seconds
The Average Delay is 0.0004915339606148856026785714286 seconds
Graph Plotting is Complete
Finished sending, Terminating the connection...
Terminated the connection successfully.
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$

```

Example of a situation where the initial delay is very high

It is a situation where the packet delay exceeds a second leading to multiple seconds with very low throughput. As the delay runs into multiple seconds, there are multiple instances where no packet will be transferred. We have set the interval to 3 seconds and are transferring 10 packets of size 128 bytes.

```
mastershubham@LAPTOP-8Q15SH66: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1
mastershubham@LAPTOP-8Q15SH66:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 iperf_client.py
Enter the number of echo messages to be sent: 10
Enter the interval: 3
Enter the packet size in bytes: 128
Calibrating the size of the buffer...
Second 1 :
Received message: rvk0frs3dxwzlp1liim5jt937Wknnxbntkhzeq0urt6q5ctjn
The Average Throughput is 256 bytes/seconds
The Average Delay is 0.000234127044677734375 seconds
Second 2 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 3 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 4 :
Received message: 45jp5t0a9q7brftswaxup8vn87y65uhx1k
The Average Throughput is 256 bytes/seconds
The Average Delay is 0.000255107879638671875 seconds
Second 5 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 6 :
Received message: 0ddw73mp0hcjvhl49j8h
The Average Throughput is 256 bytes/seconds
The Average Delay is 0.000247955322656250000 seconds
Second 7 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 8 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 9 :
Received message: pnrwzgm2tu3ur12zetdk2u0taw7
The Average Throughput is 256 bytes/seconds
The Average Delay is 0.0002760887145996093750 seconds
Second 10 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 11 :
Received message: 1xgi9ur233i
The Average Throughput is 256 bytes/seconds
The Average Delay is 0.000299930572599765625 seconds
Second 12 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
```

```
mastershubham@LAPTOP-8Q15SH66: /mnt/d/CS301-Computer Networks/Assignment 3/Part 1
mastershubham@LAPTOP-8Q15SH66:/mnt/d/CS301-Computer Networks/Assignment 3/Part 1$ python3 iperf_server.py
UDP server has binded and is waiting for client connection
Message from the Client is c :128
The IP Address and port of the Client is ('127.0.0.1', 42609)
Updated buffer size to 128 bytes
Message from the Client is 1h7lqkpgb7ay6hr9rxx4
Message from the Client is hvaazxhp5gvvtne8anu
Message from the Client is 8du885ehz887oefxk
Message from the Client is 1s3jjerb17cm
Message from the Client is 91m073talkm5p0q9
Message from the Client is ca5vqkew7trw
Message from the Client is 1g0c7fmk117i0lex5osi
Message from the Client is 5jpxhd09wm6
Message from the Client is so2cirayc4zg7m6
Message from the Client is 8mui4pb87w4qlw5e5z9
```

If you look closely:

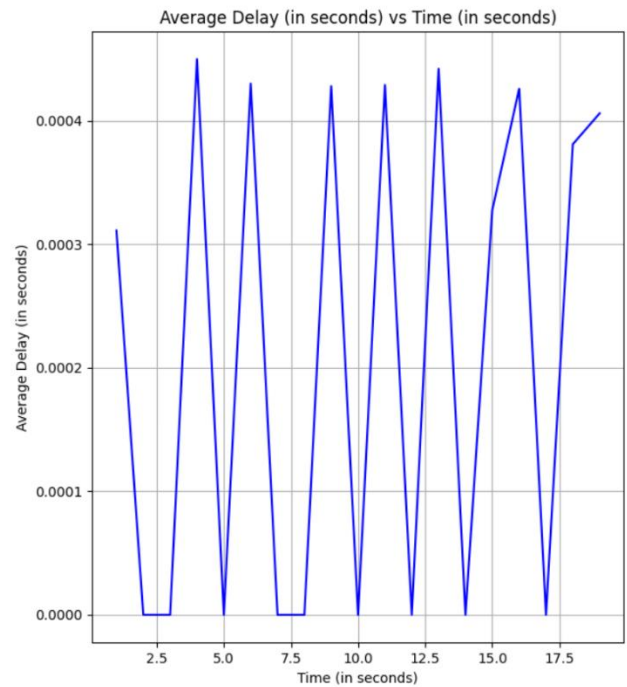
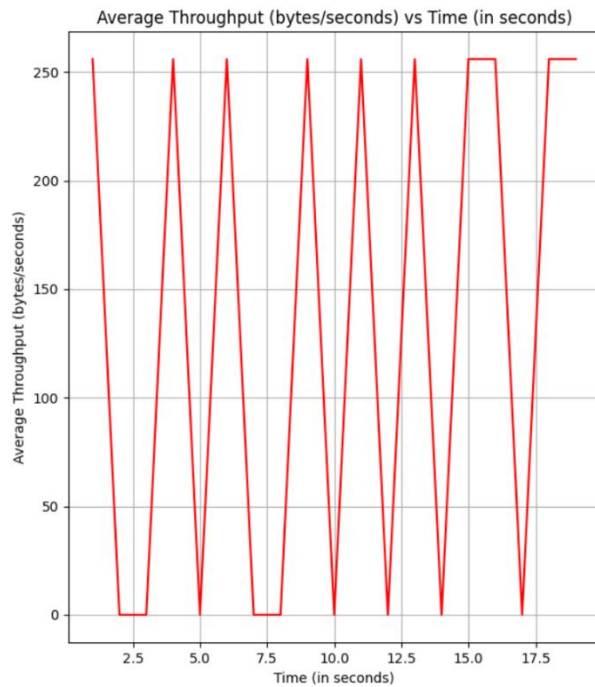
We can see instances of effective throughput and average delay to be ZERO as no packet is being transferred.

```
The Average Throughput is 256 bytes/seconds
The Average Delay is 0.000234127044677734375 seconds
Second 2 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 3 :
No packet will be sent in this second
The Average Throughput is 0 bytes/seconds
The Average Delay is 0 seconds
Second 4 :
```

In the graph also we can see that throughput continuously fluctuates up and down. The large interval cases where the throughput is ZERO and situations where one packet is transferred lead to a high throughput value.

The same principle applies to the average delay as well. The value increases when packets are transferred or stays at ZERO if no packet is transferred on the second time scale.

If there are a large number of packets, then the interval size will become very small while transmitting the latter half of the packet leading to a situation similar to the 1st drop where we transfer a large number of packets with negligible interpacket interval, leading to a polynomial increase of the effective throughput and sharp drops due to the artificial packet loss scenario as previously explained.



Part 2

Feature 1

File Transfer

For the File Transfer, I have implemented sharing of a file present at the server to the client. Since the file transfer cannot afford packet loss, I am using TCP Stream.

The idea of implementation is pretty straightforward. We first create sockets on both sides, client, and server. The server binds and listens for a client socket. It then accepts the connection from the client-side.

After the connection is established, the file is transferred. The server reads the file line by line and transmits the data in bytes to the client. The client creates a new file i.e., the file to copy the data coming from the server.

It then writes the data from the receiving buffer line by line onto the newly created file till all the data from the buffer has been exhausted. Once all the data has been copied from the server to the client file, the connection is terminated and sockets are closed at both ends.

Simple Python file I/O is being used to read and write the data coming from the stream. It is like copying a file on two different local hosts and the bandwidth is transmitting the byte stream.

Use cases of File Transfer:

- It is very important for downloading (server to client) and uploading (client) files. It is very useful for getting and putting data on the internet. Books, Music, Papers, Webpages, Images, etc are transferred using File Transfer.
- This is for a client-server architecture, but it essentially is a peer-to-peer from one IP to another IP in my local machine. The entire Peer-to-Peer architecture uses File Transfer to send data across machines. BitTorrent is a very well-known example of the same.
- It can also be used for large and continued large data transfer from an organization/entity to another organization/entity.
- This is an example of local file transfer which can be extended to remote transfer as well

Codes

```
file_transfer_server.py

# Import the necessary libraries

from sys import exit

from time import sleep

# Import the necessary functions from the socket library
from socket import AF_INET
from socket import error
from socket import socket
from socket import SOCK_STREAM
from socket import SOL_SOCKET
from socket import SO_REUSEADDR

# Recursive function to try to bind to the socket
def bind_socket():

    # As we need to bind the uninitialized socket, calling it
    global s

    # Try to complete the binding of the socket
    try:

        # Appropriate message is displayed
        print("Binding the Port:", port)

        # Bind to the specified host and port
        s.bind((host, port))

        # Since it is the server socket, it will listen
        s.listen(5)

    # Socket is unable to bind to the specific socket address
    except error as msg:

        # We seek to try binding again
        # Appropriate message is displayed
        print("Socket Binding error", msg)
```

```

    # Seek to retry
    # Appropriate message is displayed
    print("Retrying...")

    # Wait for the function to end
    sleep(5)

    # Retry binding by calling the same function recursively
    bind_socket()

# Host is initialized
host = "

# Standard port is initialized
port = 8888

# Uninitialized socket at server
s = None

# Create the server socket
try:

    # Try to create the server socket with any IP address and TCP Stream
    s = socket(AF_INET, SOCK_STREAM)

# Unable to create server socket
except error as msg:

    # Appropriate message is displayed
    print("Socket creation error:", msg)

    # Since we are not able to normally bind to the socket
    # We will forcibly bind to the socket

    # SOL_SOCKET manipulates the current socket in use
    # SO_REUSEADDR forcefully attaches to the address of the same socket
    # 1 is a BOOL value for True which sets the forceful attachment to True
    s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# Bind the socket to the specified host and port using the above user-defined function
bind_socket()

# Binding is complete which indicates the server is listening
# Appropriate message is displayed
print("Server Listening...")

```

```
# Accept the socket connection from the client connection
conn, address = s.accept()

# Appropriate message is displayed
print("Connection has been established! |", "IP", address[0], "| Port", address[1])

# Connection has been established which allows to proceed with the file transfer to the client

# Open the file which needs to be sent in read mode
f = open('test_file.txt','rb')

# Read the file line by line
l = f.read(1024)

# Read as long as there is data left in the file
while (l):

    # Send the file data line by line to the client
    conn.send(l)

    # Read the next line in the file
    l = f.read(1024)

# As we have fully read the file, we can close it
f.close()

# Wait for an interval of 5 seconds for the file to fully close
sleep(5)

# As all the file contents are sent, the closing message is sent
# This is important as it will tell the client to close the connection
conn.send(str.encode("Thank you for connecting"))

# Appropriate message is displayed
print('Done sending')

# Close the connection
conn.close()

# Close the socket at server
s.close()

# Exit the server program
exit()
```

file_transfer_client.py

Import the necessary libraries

Import the necessary functions from the socket library

from socket import AF_INET

from socket import error

from socket import socket

from socket import SOCK_STREAM

Uninitialized socket at client

s = None

Standard host is initialized

host = 'localhost'

Standard port is initialized

port = 8888

Create the client socket

try:

Try to create the client socket with any IP address and TCP Stream

s = socket(AF_INET, SOCK_STREAM)

Unable to create client socket

except error as msg:

Print appropriate message to the user

print("Socket creation error:", msg)

Connect the client socket to the server host and port

try:

Try to connect the socket

s.connect((host, port))

Unable to connect client socket

except error as msg:

Print appropriate message to the user

print("Socket connection error:", msg)


```
# Client socket has been created and established, so the transferred file is written  
with open('received_file.txt', 'w') as f:  
  
    # Open a new file  
    print('file opened')  
    while True:  
  
        # Collecting data from the server  
        print('receiving data...')  
  
        # Receiving server data  
        # Decoding the data sent as bytes as character strings  
        # This makes it very easy to compare and write on the file  
        data = s.recv(1024).decode("utf-8")  
  
        # If there is a message to complete the connection i.e., data receival is complete  
        if data == 'Thank you for connecting':  
  
            # Print appropriate message to the user  
            print('Successfully get the file')  
            print('Thank you for connecting')  
  
            # As no more data will come, exit the loops  
            break  
  
        # Write the incoming data to the established file  
        f.write(data)  
  
    # Close the file as the writing is complete  
    f.close()  
  
    # Close the socket as data transfer is complete  
    s.close()
```

Results

We have created a file named “test_file.txt” which will be transferred from server to client. The contents are below.

Note:

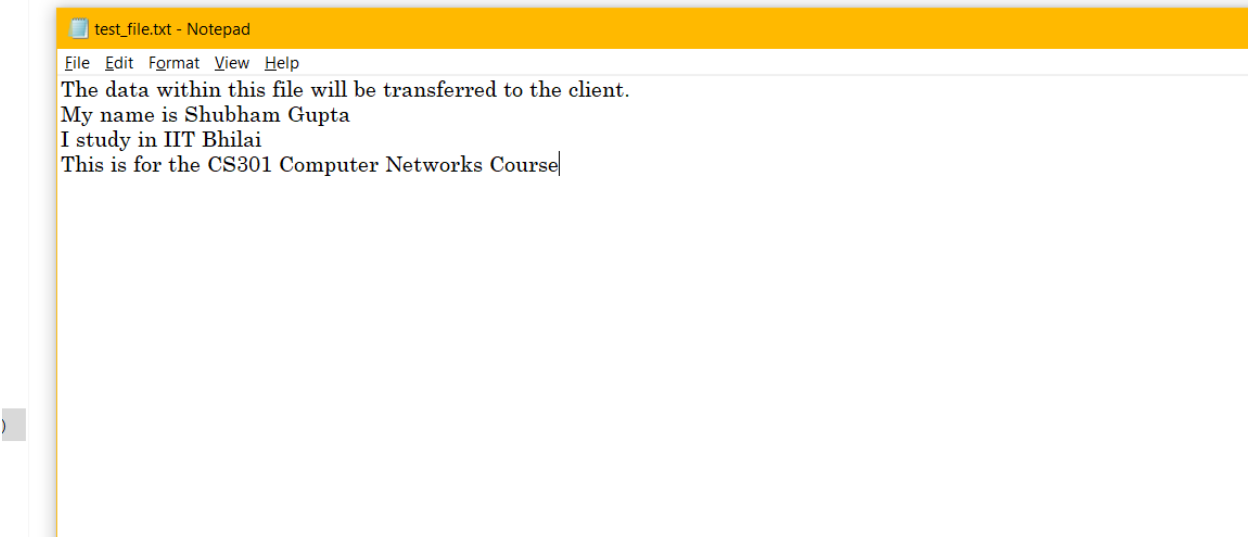
- It is important to initialize the “test_file.txt” because the server will transfer only the file with that name. The code segment specifies this.

```
# Connection has been established which allows to proceed with the file transfer to client  
  
# Open the file which needs to be sent in read mode  
f = open('test_file.txt','rb')
```

- It will run into a file opening error if the file is not present

This PC > New Volume (D:) > CS301-Computer Networks > Assignment 3 > Part 2

Name	Date modified	Type	Size
file_transfer_client.py	12-11-2021 17:49	PY File	2 KB
file_transfer_server.py	12-11-2021 17:50	PY File	4 KB
test_file.txt	12-11-2021 17:42	Text Document	1 KB



Executing Server file

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 file_transfer_server.py
Binding the Port: 8888
Server Listening...
```

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

Executing client file

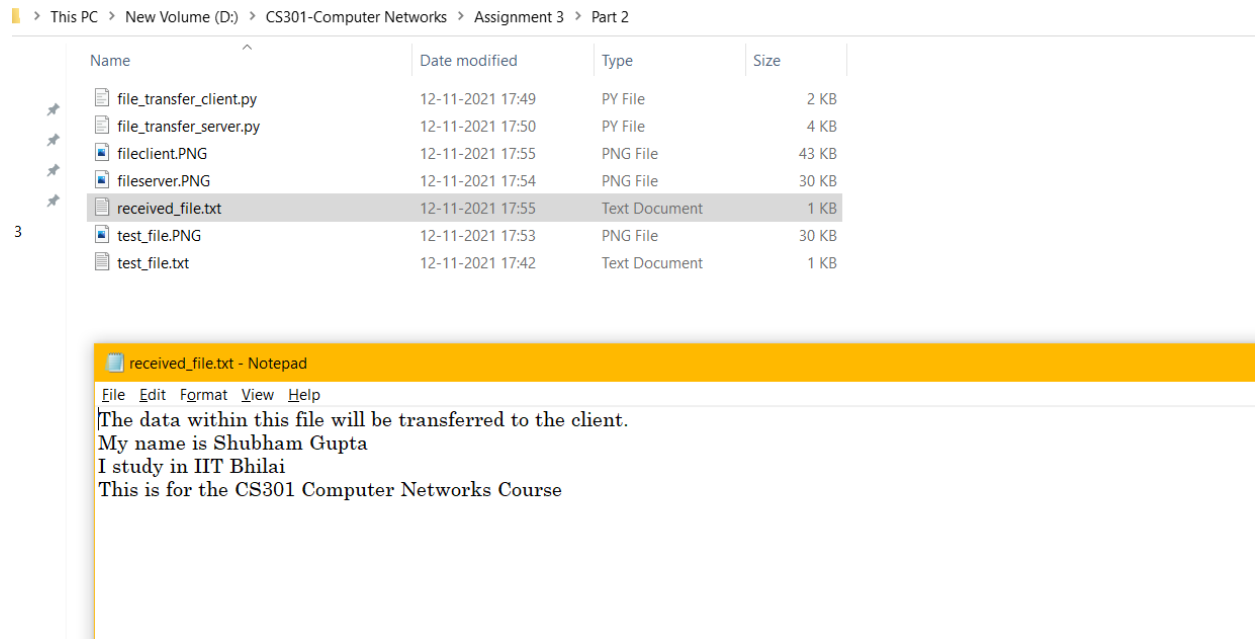
```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 file_transfer_server.py
Binding the Port: 8888
Server Listening...
Connection has been established! | IP 127.0.0.1 | Port 49906
Done sending
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 file_transfer_client.py
file opened
receiving data...
receiving data...
Successfully get the file
Thank you for connecting
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

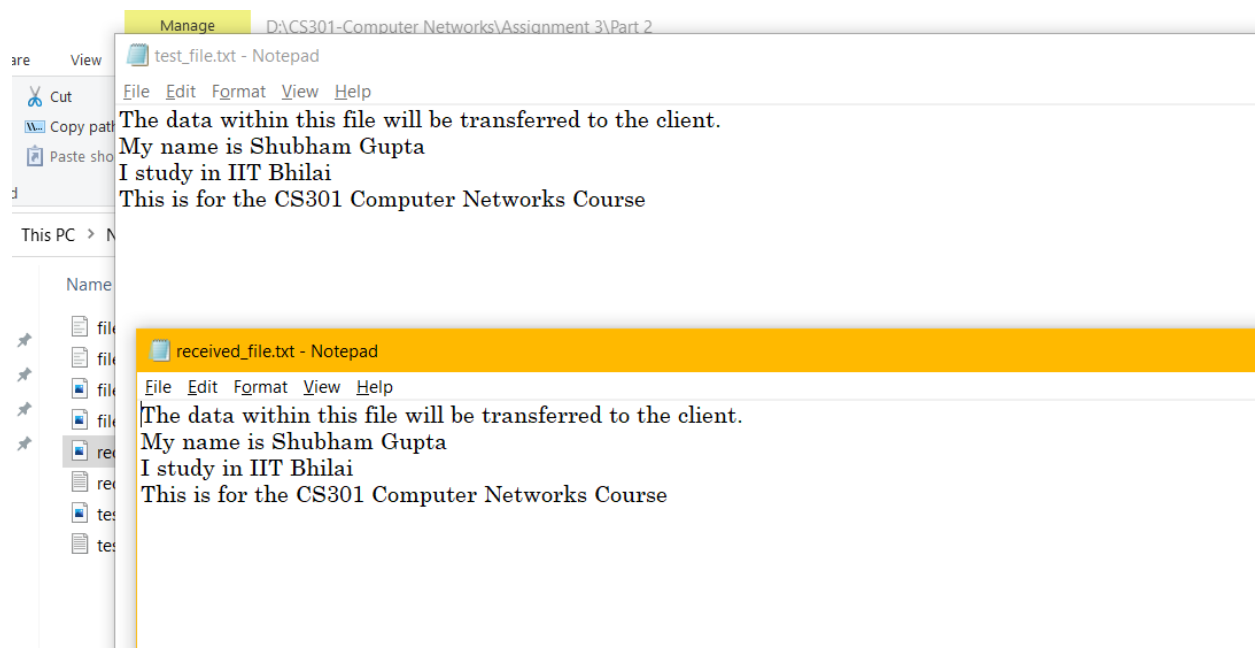
Post execution of the client file.

The file transfer is complete i.e., it is received at the client-side from the server-side and stored as “received_file.txt” with the contents same as the server file. It is essentially just creating a copy of the file at the server.

Note: “test_file.txt” file is previously present and not changed



As you can see that the contents of both the files are the same



Feature 2

Reverse Shell

For the reverse shell, I have implemented a system to connect to another remote computer's shell. The server opens a connection and waits for the client to respond. The moment the client connects to the server, the prompt of the client is opened at the server's end. Once this bash shell is opened at the server end, the server can run any commands on the client end remotely.

The idea of implementation is pretty straightforward. We first create sockets on both sides, client, and server. The server binds and listens for a client socket. It then accepts the connection from the client-side.

After the connection is established, the reverse shell starts operating at the server end which is ready to accept commands. The server takes commands as strings and encodes them as bytes and sends them to the client. As it is a reverse shell of the client the commands need to be run on the client-side.

Once the commands are sent to the client-side, it is run on the client shell as a child subprocess to the currently running parent shell process. This happens in the background with a pipe to standard input, output, and error stream as we need to access the standard I/O to run the shell commands.

In the case of command "cd", we need to change the directory at both the client and server end, so the directory is changed on both the client and server-side in the shell process itself. Since we are directly running the command on the shell, we don't need to create a child subprocess as described above to run the "cd" command.

As it is running in the background, nothing will display on the client-side but data will be sent to the server-side.

Once the process execution is complete on the client-side, the standard output and standard error are encoded and sent at a byte stream to the server end. The server decodes this byte stream and displays it to the user.

In this manner, the execution of the client is done via the server. The client executes the command taken as input on the server and sends the necessary details back to the server. The server can run necessary commands on the client using this shell.

On the command "exit", the instance of the reverse shell is terminated and the connections are closed on the server and client end.

Use cases of Reverse Shell:

- Consider the case of remote communication. If you and your friend are living in two different cities and are working on an Open-Source project that demands a lot of testing and his terminal has crashed which he is unable to fix. To fix his system, the server file can run on localhost and client file on his system. Once the connection is performed, you can access his shell and make the necessary changes to fix the system. Accessing the shell can fix intricate and deep-seated system problems easily without much ado.
- This has come in very handy in the current scenario where everyone is working remotely. It can be used for frequent communication and git projects with multiple students.
- Management of a dedicated server remotely can be done with a reverse shell. We can monitor logs, files, and user requests in a different place and a different system and even modify the requisite files easily. It can make management easy and time-saving.
- The idea of acquiring a shell remotely can backfire. We can take control of another system and disrupt it. This code does not deal with permissions and cannot access files that need elevated access. If we extend this to a shell that has elevated access like root then we can destroy the system with a few commands.
- It is important to use this kind of remote shell carefully

Codes

reverse_shell_server.py

Import the necessary libraries

from sys import exit

from time import sleep

Import the necessary functions from the socket library

from socket import AF_INET

from socket import error

from socket import socket

from socket import SOCK_STREAM

from socket import SOL_SOCKET

from socket import SO_REUSEADDR

Recursive function to try to bind to the socket

def bind_socket():

*# As we need to bind the uninitialized socket, calling it
global s*

*# Try to complete the binding of the socket
try:*

*# Appropriate message is displayed
print("Binding the Port:", port)*

*# Bind to the specified host and port
s.bind((host, port))*

*# Since it is the server socket, it will listen
s.listen(5)*

*# Socket is unable to bind to the specific socket address
except error as msg:*

*# We seek to try binding again
Appropriate message is displayed
print("Socket Binding error", msg)*

```

    # Seek to retry
    # Appropriate message is displayed
    print("Retrying...")

    # Wait for an interval of 5 seconds for the function to end
    sleep(5)

    # Retry binding by calling the same function recursively
    bind_socket()

# Host is initialised
host = "

# Standard port is initialised
port = 8888

# Uninitialized socket at server
s = None

# Create the server socket
try:

    # Try to create the server socket with any IP address and TCP Stream
    s = socket(AF_INET, SOCK_STREAM)

# Unable to create server socket
except error as msg:

    # Appropriate message is displayed
    print("Socket creation error:", msg)

    # Since we are not able to normally bind to the socket
    # We will forcibly bind to the socket

    # SOL_SOCKET manipulates the current socket in use
    # SO_REUSEADDR forcefully attaches to the address of the same socket
    # 1 is a BOOL value for True which sets the forceful attachment to True
    s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# Bind the socket to the specified host and port using the above user-defined function
bind_socket()

# Binding is complete which indicates the server is listening
# Appropriate message is displayed
print("Server Listening...")

```



```

# Accept the socket connection from the client connection
conn, address = s.accept()

# Appropriate message is displayed
print("Connection has been established! |", "IP", address[0], "| Port", address[1])

# The client sends the working directory to print before entering the command
# It needs to be displayed to the user for him to learn the current working directory

# The received bytes have to be decoded to string
print(conn.recv(2048).decode("utf-8"), end = "")

# We will continue to take shell commands till we don't exit
while True:

    # Take the shell command from the user as a string
    cmd = str(input())

    # If the command to exit the shell is entered, exit the shell
    if cmd == 'exit':

        # Tell the client to close the connection
        # Send a closing message to the client
        # We encode to send the string as bytes across the connection
        conn.send(cmd.encode())

        # Close the connection
        conn.close()

        # Close the socket
        s.close()

        # Exit the program
        exit()

    # If a valid command is entered in the shell other than exit
    # We need to parse it and send it to the server to execute on the reverse shell

    if len(cmd.encode()):

        # We encode to send the commanded string as bytes across the connection
        conn.send(cmd.encode())

```

```
# Display the contents of the output of the executed shell command on the screen  
# As the reverse shell runs on the client it is sent to the server which displays it  
  
# Decode the received bytes to a string to display string output  
print(conn.recv(2048).decode("utf-8"), end = "")
```

reverse_shell_client.py

Import the necessary libraries

from os import chdir
from os import getcwd

from subprocess import PIPE
from subprocess import Popen

Import the necessary functions from the socket library
from socket import AF_INET
from socket import error
from socket import socket
from socket import SOCK_STREAM

Uninitialized socket at client
s = None

Standard host is initialized
host = 'localhost'

Standard port is initialized
port = 8888

Create the client socket
try:

Try to create the client socket with any IP address and TCP Stream
s = socket(AF_INET, SOCK_STREAM)

Unable to create client socket
except error as msg:

Print appropriate message to the user
print("Socket creation error:", msg)

Connect the client socket to the server host and port
try:

Try to connect the socket
s.connect((host, port))

Unable to connect client socket
except error as msg:

```

# Print appropriate message to the user
print("Socket connection error:", msg)

# The server needs to print the current working directory in the reverse shell
currentWD = getcwd() + "> "

# The calculated current working directory is encoded and sent as a byte stream to the server
s.send(currentWD.encode())

# We will continue to receive shell commands from the server till the user does not want to exit
while True:

    # Receive the data from the server and process it as a shell command
    # Decoding the byte stream as a string to process as a valid command
    data = s.recv(2048).decode("utf-8")

    # If the command to exit the shell is entered, exit the shell
    # Exiting the loop will prevent it from taking more commands
    if data == 'exit':

        # Terminate the loop
        break

    # Since this is a command to change the directory to a new directory
    # The path of the shell needs to be changed, it is handled separately
    elif data[:2] == 'cd':

        # To store the necessary message for the user
        output_str = ""

        # Try to change the directory path to the specified directory
        try:

            # Specified directory path is the string after "cd"
            # We leave the 1st 2 characters and consider the string after the 3rd character
            chdir(data[3:])

        # Unable to change the directory path as it does not exist
        except:

            # The necessary error message for the user
            output_str = "No such directory available\n"

```

```

# As the working directory has been changed
# We need to update the current working directory before sending it to the user
currentWD = getcwd() + "> "

# Send the necessary details to the user
# Error messages and changed paths must be visible to the user in the reverse shell
# String is encoded as a byte stream and sent
s.send(str.encode(output_str + currentWD))

# It is a valid shell command that does not interfere with the directory path
# It must be processed as a proper shell process
elif len(data):

    # Create a child process as a new process to the currently running parent process
    # As we will be using stdin, stdout, stderr, and a shell, we need to create a pipe to open a
    stream for the standard processes
    # All the arguments use PIPE as the standard streams need to be opened

    # As the shell command needs to run as a full process, we create a child process and run it
    in that process
    cmd = Popen(data[:], shell = True, stdout = PIPE, stdin = PIPE, stderr = PIPE)

    # The output byte stream must consist of the standard output and standard error of the
    created child process
    output_byte = cmd.stdout.read() + cmd.stderr.read()

    # As the user will be seeing a string, the byte stream is decoding into a string
    output_str = output_byte.decode("utf-8")

    # We need to update the current working directory before sending it to the user
    currentWD = getcwd() + "> "

    # Send the necessary details to the user
    # Output and Error messages and current path must be visible to the user in the reverse
    shell
    # String is encoded as a byte stream and sent
    s.send(str.encode(output_str + currentWD))

# As the reverse shell has been exited, close the socket connection
s.close()

```

Results

Executing server file

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server Listening...

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

Executing client file

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server Listening...
Connection has been established! | IP 127.0.0.1 | Port 49918
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2>

mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_client.py
```

As we can see that the reverse shell has begun. The current working directory has been printed and is open to commands.

With this, we can enter a few commands on the reverse shell and show the working.

\$ ls -l

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server listening...
Connection has been established! | IP 127.0.0.1 | Port 49918
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ ls -l
total 288
-rwxrwxrwx 1 mastershubham mastershubham 44169 Nov 12 18:22 copy.PNG
-rwxrwxrwx 1 mastershubham mastershubham 1995 Nov 12 19:17 file_transfer_client.py
-rwxrwxrwx 1 mastershubham mastershubham 3448 Nov 12 19:11 file_transfer_server.py
-rwxrwxrwx 1 mastershubham mastershubham 43965 Nov 12 17:55 fileclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 8169 Nov 12 18:25 fileneeded.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30513 Nov 12 17:54 fileserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 40243 Nov 12 18:03 receive.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:55 received_file.txt
-rwxrwxrwx 1 mastershubham mastershubham 4367 Nov 12 22:56 reverse_shell_client.py
-rwxrwxrwx 1 mastershubham mastershubham 4005 Nov 12 22:56 reverse_shell_server.py
-rwxrwxrwx 1 mastershubham mastershubham 35140 Nov 12 23:04 reverseclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 29264 Nov 12 23:03 reverseserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30362 Nov 12 17:53 test_file.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:42 test_file.txt
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_client.py
```

As we can see that the files and directories present in the current working directory have been printed on the execution of the commands. It means the shell is working and can run all the necessary shell commands

\$ netstat -a

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server listening...
Connection has been established! | IP 127.0.0.1 | Port 49918
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ ls -l
total 288
-rwxrwxrwx 1 mastershubham mastershubham 44169 Nov 12 18:22 copy.PNG
-rwxrwxrwx 1 mastershubham mastershubham 1995 Nov 12 19:17 file_transfer_client.py
-rwxrwxrwx 1 mastershubham mastershubham 3448 Nov 12 19:11 file_transfer_server.py
-rwxrwxrwx 1 mastershubham mastershubham 43965 Nov 12 17:55 fileclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 8169 Nov 12 18:25 fileneeded.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30513 Nov 12 17:54 fileserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 40243 Nov 12 18:03 receive.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:55 received_file.txt
-rwxrwxrwx 1 mastershubham mastershubham 4367 Nov 12 22:56 reverse_shell_client.py
-rwxrwxrwx 1 mastershubham mastershubham 4005 Nov 12 22:56 reverse_shell_server.py
-rwxrwxrwx 1 mastershubham mastershubham 35140 Nov 12 23:04 reverseclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 29264 Nov 12 23:03 reverseserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30362 Nov 12 17:53 test_file.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:42 test_file.txt
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 0.0.0.0:8888 0.0.0.0:* LISTEN
tcp 0 0 0.0.0.0:49918 0.0.0.0:* ESTABLISHED
tcp 0 0 0.0.0.0:8888 0.0.0.0:* ESTABLISHED
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags Type State I-Node Path
unix 2 [ ACC ] SEQPACKET LISTENING 21511 /run/NSL/9_interop
unix 2 [ ACC ] SEQPACKET LISTENING 1101 /run/NSL/12_interop
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_client.py
```

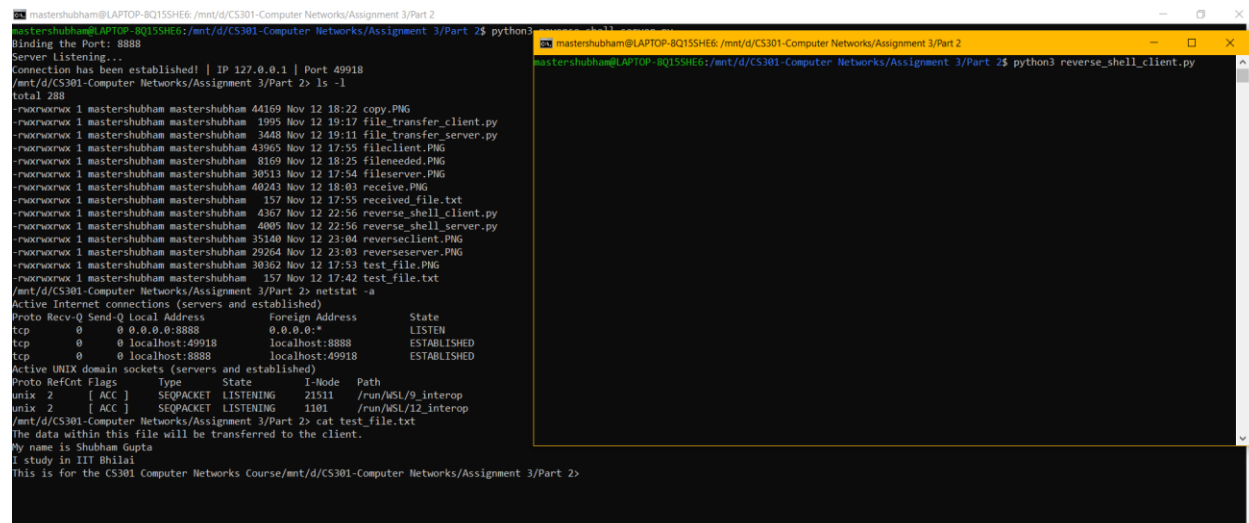
If you look at the image carefully, all the connections to the server have been listed. We can see that the server is linked to the client from port 8888 to 49718 | IP 127.0.0.1 on stating this command.

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server listening...
Connection has been established! | IP 127.0.0.1 | Port 49918
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ ls -l
```

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_client.py
```

It gives information that the client-server connection is live and we can continue to enter commands to get the necessary data as needed. UNIX domain sockets are also listed in the same command giving us a full report of the running sockets in the system.

\$ cat

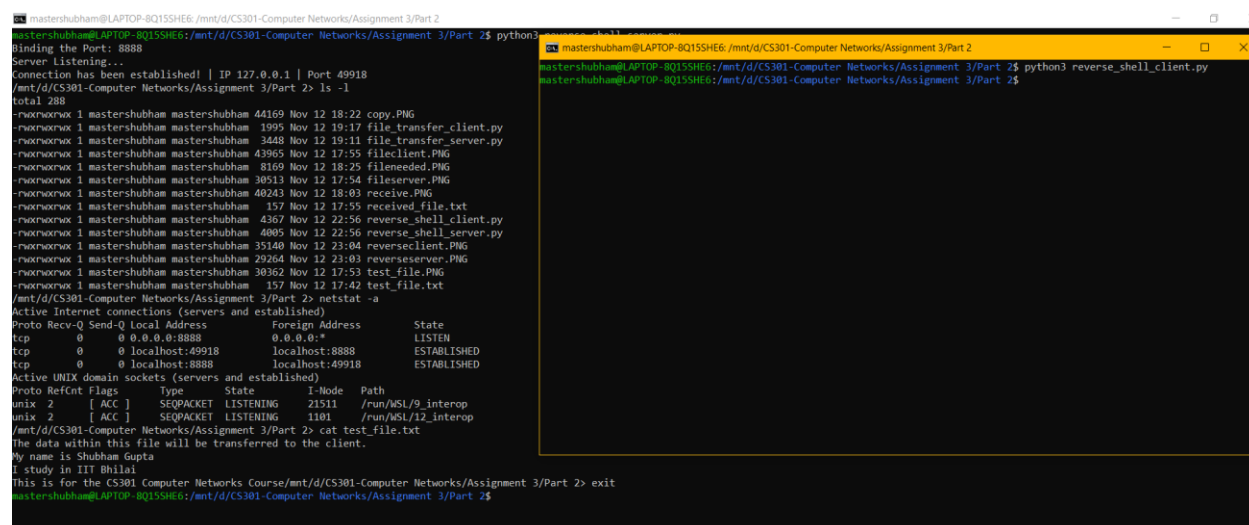


```
mastershubham@LAPTOP-BQ155HE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server Listening...
Connection has been established! | IP 127.0.0.1 | Port 49918
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> ls -l
total 288
-rwxrwxrwx 1 mastershubham mastershubham 44169 Nov 12 18:22 copy.PNG
-rwxrwxrwx 1 mastershubham mastershubham 1995 Nov 12 19:17 file_transfer_client.py
-rwxrwxrwx 1 mastershubham mastershubham 3448 Nov 12 19:11 file_transfer_server.py
-rwxrwxrwx 1 mastershubham mastershubham 43965 Nov 12 17:55 fileclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 8169 Nov 12 18:25 fileneeded.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30513 Nov 12 17:54 fileserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 40243 Nov 12 18:03 receive.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:55 received_file.txt
-rwxrwxrwx 1 mastershubham mastershubham 4367 Nov 12 22:56 reverse_shell_client.py
-rwxrwxrwx 1 mastershubham mastershubham 4005 Nov 12 22:56 reverse_shell_server.py
-rwxrwxrwx 1 mastershubham mastershubham 35140 Nov 12 23:04 reverseclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 29264 Nov 12 23:03 reverseserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30362 Nov 12 17:53 test_file.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:42 test_file.txt
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:8888             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:49918            localhost:8888          ESTABLISHED
tcp        0      0 0.0.0.0:8888             localhost:49918        ESTABLISHED
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   Path
unix 2      [ ACC ] SEQPACKET LISTENING    21511    /run/MSL/9_interop
unix 2      [ ACC ] SEQPACKET LISTENING    1101     /run/MSL/12_interop
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> cat test_file.txt
The data within this file will be transferred to the client.
My name is Shubham Gupta
I study in IIT Bhilai
This is for the CS301 Computer Networks Course/mnt/d/CS301-Computer Networks/Assignment 3/Part 2>
```

As we can see that the contents of the file have been displayed on the terminal

We have shown enough examples of the working of the reverse shell. There are a lot of Linux commands out there that can be tried and experimented with. We will exit and show the result

\$ exit



```
mastershubham@LAPTOP-BQ155HE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$ python3 reverse_shell_server.py
Binding the Port: 8888
Server Listening...
Connection has been established! | IP 127.0.0.1 | Port 49918
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> ls -l
total 288
-rwxrwxrwx 1 mastershubham mastershubham 44169 Nov 12 18:22 copy.PNG
-rwxrwxrwx 1 mastershubham mastershubham 1995 Nov 12 19:17 file_transfer_client.py
-rwxrwxrwx 1 mastershubham mastershubham 3448 Nov 12 19:11 file_transfer_server.py
-rwxrwxrwx 1 mastershubham mastershubham 43965 Nov 12 17:55 fileclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 8169 Nov 12 18:25 fileneeded.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30513 Nov 12 17:54 fileserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 40243 Nov 12 18:03 receive.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:55 received_file.txt
-rwxrwxrwx 1 mastershubham mastershubham 4367 Nov 12 22:56 reverse_shell_client.py
-rwxrwxrwx 1 mastershubham mastershubham 4005 Nov 12 22:56 reverse_shell_server.py
-rwxrwxrwx 1 mastershubham mastershubham 35140 Nov 12 23:04 reverseclient.PNG
-rwxrwxrwx 1 mastershubham mastershubham 29264 Nov 12 23:03 reverseserver.PNG
-rwxrwxrwx 1 mastershubham mastershubham 30362 Nov 12 17:53 test_file.PNG
-rwxrwxrwx 1 mastershubham mastershubham 157 Nov 12 17:42 test_file.txt
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:8888             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:49918            localhost:8888          ESTABLISHED
tcp        0      0 0.0.0.0:8888             localhost:49918        ESTABLISHED
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   Path
unix 2      [ ACC ] SEQPACKET LISTENING    21511    /run/MSL/9_interop
unix 2      [ ACC ] SEQPACKET LISTENING    1101     /run/MSL/12_interop
/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> cat test_file.txt
The data within this file will be transferred to the client.
My name is Shubham Gupta
I study in IIT Bhilai
This is for the CS301 Computer Networks Course/mnt/d/CS301-Computer Networks/Assignment 3/Part 2> exit
mastershubham@LAPTOP-BQ155HE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 2$
```

We can see that the reverse shell has been terminated and the sockets and connections are closed on both, the client and server end. Thus, we can see that both, the instances of the server and client are terminated.

Part 3

Core Idea

Since we need to make the echo client-server protocol independent, we will essentially use the functionality of AF_UNSPEC. This prevents the protocol to distinguish between IPv4 and IPv6 addressing as it can accept both IPv4 and IPv6 addresses; when the getaddrinfo socket function is called to establish a socket.

Another shortcoming is the length of the IPv6 address. Instead of classifying it as an IPv4 address length, we store it in a variable that can accommodate an arbitrary length bigger than IPv6. So, storing the variable with an IPv6 address will not be an issue.

The code segment from the code is as follows:

```
# AF_UNSPEC indicates that the caller will accept any protocol family and will not distinguish between IPv4 and IPv6
# SOCK_STREAM means that it is a TCP socket
for addrFamily, socketKind, protocol, cn, socketAddress in getaddrinfo(host, port, AF_UNSPEC, SOCK_STREAM):
```

The code has been documented and will provide the other implementation details

getaddrinfo translates the given hostname to an address. If the input is ip6-localhost or localhost, the AF_UNSPEC will allow translation of both addresses.

As we can translate both types of addresses, we can establish the sockets and create the necessary socket address to bind to.

Once the sockets are established, we just need to transfer the data. With this, we can establish connections on both sides using both forms of addressing

Code

server.py

Import the necessary libraries

from os import _exit

Import the necessary functions from the socket library

from socket import AF_UNSPEC

from socket import getaddrinfo

from socket import SOCK_STREAM

from socket import socket

Collect the user input and inform the default inputs

host = input('Enter host (Default host = ip6-localhost) (Enter localhost for IPv4 hostname): ') # localhost or ip6-localhost

port = input('Enter port (Default port = 8000): ') # 8000

If host input is not entered, set to default value

if host == ":

host = 'ip6-localhost'

If port input is not entered, set to default value

if port == ":

port = 8000

Uninitialized socket at server

ServerSocket = None

Iterate through the list of tuples containing information about socket(s) that can be created with the service.

This is performed by getaddrinfo socket function <https://en.wikipedia.org/wiki/Getaddrinfo>

AF_UNSPEC indicates that the caller will accept any protocol family and will not distinguish between IPv4 and IPv6

SOCK_STREAM means that it is a TCP socket

for addrFamily, socketKind, protocol, cn, socketAddress in getaddrinfo(host, port, AF_UNSPEC, SOCK_STREAM):

We will try to establish a socket at the server

try:

```

# Socket is created at the server
ServerSocket = socket(addrFamily, socketKind, protocol)

try:

    # Try to bind the created socket at the specific socket address
    ServerSocket.bind(socketAddress)

    # As it is a socket at the server, it is open to listening from the client-side
    ServerSocket.listen(1)

    # As socket is created and listening, it is in use
    print(f'Current socket in use: {addrFamily, socketKind, protocol, cn, socketAddress}')

    # The socket has been established and is in use
    break

except:

    # Socket is unable to bind to the specific socket address, hence close the connection
    ServerSocket.close()

    # As the socket is closed, there is no initialised socket
    ServerSocket = None

# As we are unable to establish the socket at the server, the socket remains uninitialized
except:

    # Socket remains uninitialized
    ServerSocket = None

# The case where we are unable to create a socket at the server, we need to exit
if ServerSocket == None:
    print('Cannot create a connection with provided socket.')
    print('Exiting...')
    _exit(1)

# As socket is created at the server, accept the connection with the client
Client, address = ServerSocket.accept()

# Client connection is accepted and connection is established
print(f'\n[*] Connected to: {address[0]}:{address[1]}')

# Keep connection open till data is being received
while True:

```

```
# Receive buffer of 1024 bytes from the Client-side
data = Client.recv(1024)

# Close the connection when data receival has stopped
if not data:

    # Connection is terminated at server end
    print(f'\n[*] Disconnected from: {address[0]}:{address[1]}')

    # Close the server socket
    Client.close()

    # Once the connection is closed from both sides
    # There is no data flow, so exit the loop
    break

# Since there is data flow from the client
# Send the same data back from the server to the client
Client.send(data)
```

client.py

Import the necessary libraries

from os import _exit

Import the necessary functions from the socket library

from socket import AF_UNSPEC

from socket import getaddrinfo

from socket import SOCK_STREAM

from socket import socket

Collect the user input and inform the default inputs

*host = input('Enter host (Default host = ip6-localhost) (Enter localhost for IPv4 hostname): ') #
localhost or ip6-localhost*

port = input('Enter port (Default port = 8000): ') # 8000

data = input('Enter the data to be sent (No default) (Please enter a value): ')

As we need an input for the data, we cannot proceed till an input is not entered

while data == ":

data = input('Please enter the data to be sent to server: ')

Proceed after the input has been received

If host input is not entered, set to default value

if host == ":

host = 'ip6-localhost'

If port input is not entered, set to default value

if port == ":

port = 8000

Uninitialized socket at client

ServerSocket = None

*# Iterate through the list of tuples containing information about socket(s) that can be created
with the service.*

This is performed by getaddrinfo socket function <https://en.wikipedia.org/wiki/Getaddrinfo>

*# AF_UNSPEC indicates that the caller will accept any protocol family and will not distinguish
between IPv4 and IPv6*

SOCK_STREAM means that it is a TCP socket

*for addrFamily, socketKind, protocol, cn, socketAddress in getaddrinfo(host, port,
AF_UNSPEC, SOCK_STREAM):*

```

# We will try to connect the client socket to the socket at the server
try:

    # Socket is created at the client
    ServerSocket = socket(addrFamily, socketKind, protocol)

    try:

        # Try to Connect the socket to the socket address at the server
        ServerSocket.connect(socketAddress)

        # Able to connect to the server socket
        print(f'This client socket: {addrFamily, socketKind, protocol, cn, socketAddress}')

        # The socket has been established and is in use
        break

    # Client socket is unable to connect to the server socket
    except:

        # Unable to connect, close the connection
        ServerSocket.close()

        # As the socket is closed, there is no initialised socket
        ServerSocket = None

# As we are unable to establish the socket at the client, the socket remains uninitialized
except:

    # Socket remains uninitialized
    ServerSocket = None

# The case where we are unable to create a socket at the client, we need to exit
if ServerSocket == None:
    print('Cannot create a connection with provided socket.')
    print('Exiting...')
    _exit(1)

# As the socket has been initialised and we have connected it to the server
print(f'Sending to Server: {data}')

# We can start sending data from the client to the server
ServerSocket.send(data.encode())

```

As the server is sending messages back to the client, it is received
`data = ServerSocket.recv(1024)`

Reply from the server side is complete
`print(f'Reply from Server: {data.decode()}')`

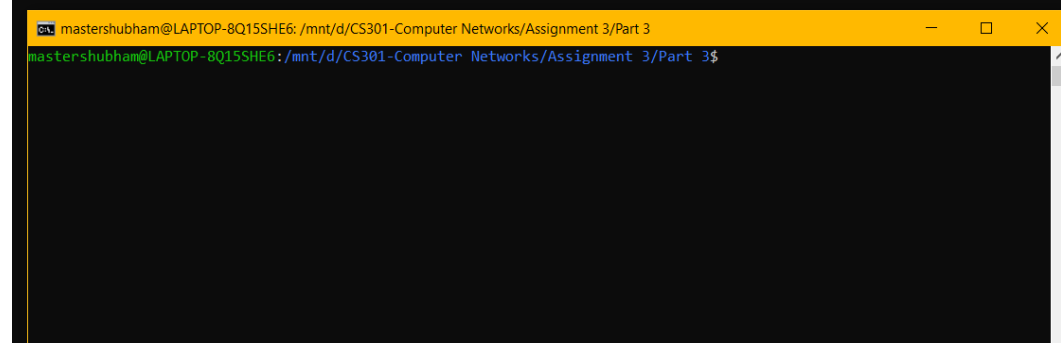
Close the socket connection at the client end
`ServerSocket.close()`

Results

IPv6 hostname: ip6-localhost

Executing server.py

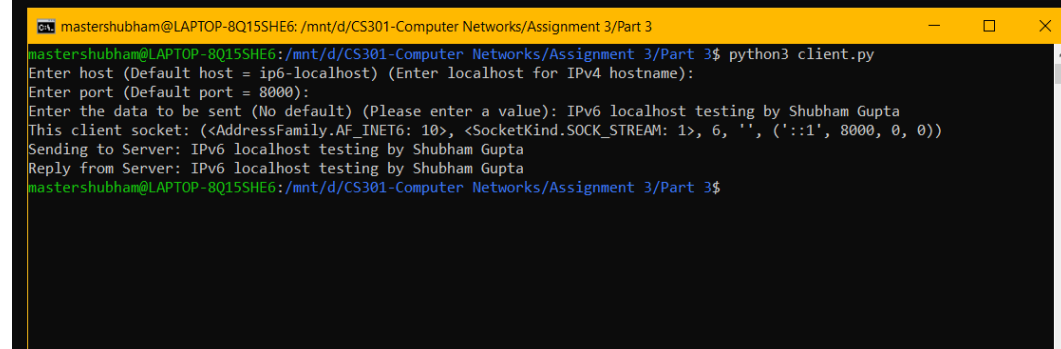
```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 3$ python3 server.py
Enter host (Default host = ip6-localhost) (Enter localhost for IPv4 hostname):
Enter port (Default port = 8000):
Current socket in use: (<AddressFamily.AF_INET6: 10>, <SocketKind.SOCK_STREAM: 1>, 6, '', ('::1', 8000, 0, 0))
```



Executing client.py

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 3$ python3 server.py
Enter host (Default host = ip6-localhost) (Enter localhost for IPv4 hostname):
Enter port (Default port = 8000):
Current socket in use: (<AddressFamily.AF_INET6: 10>, <SocketKind.SOCK_STREAM: 1>, 6, '', ('::1', 8000, 0, 0))

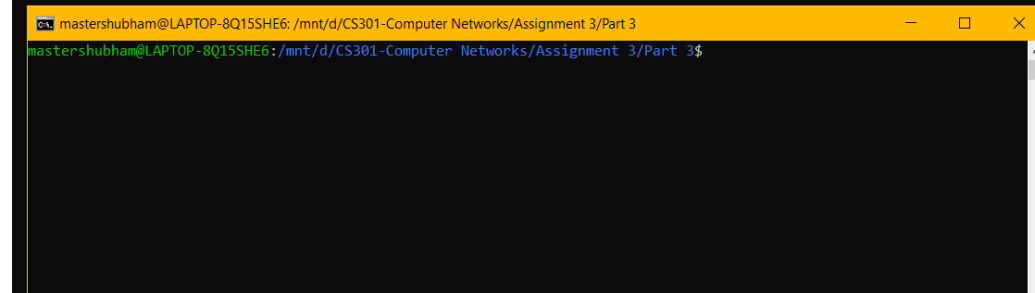
[*] Connected to: ::1:34170
[*] Disconnected from: ::1:34170
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/CS301-Computer Networks/Assignment 3/Part 3$
```



IPv4 Hostname: localhost

Executing server.py

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3$ python3 server.py
Enter host (Default host = ip6-localhost) (Enter localhost for IPv4 hostname): localhost
Enter port (Default port = 8000):
Current socket in use: (<AddressFamily.AF_INET: 2>, <SocketKind.SOCK_STREAM: 1>, 6, '', ('127.0.0.1', 8000))
```

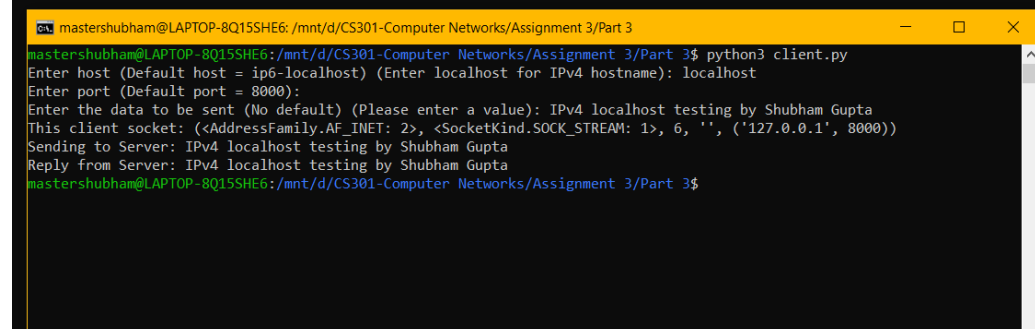


Executing client.py

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3$ python3 server.py
Enter host (Default host = ip6-localhost) (Enter localhost for IPv4 hostname): localhost
Enter port (Default port = 8000):
Current socket in use: (<AddressFamily.AF_INET: 2>, <SocketKind.SOCK_STREAM: 1>, 6, '', ('127.0.0.1', 8000))

[*] Connected to: 127.0.0.1:45388

[*] Disconnected from: 127.0.0.1:45388
mastershubham@LAPTOP-8Q15SHE6: /mnt/d/CS301-Computer Networks/Assignment 3/Part 3$
```



We can see that specifying any of the local hosts is establishing a connection on both sides and we can transfer data from the client to the server and vice-versa.

References

1. *Wikipedia*
2. *GitHub*
3. *Microsoft Docs*
4. *IBM Docs*
5. *Python Docs*
6. *Linux Man pages*
7. *Geeks for Geeks*
8. *Program Creek*
9. *Stack Overflow*

Link for the Demo Video

<https://drive.google.com/drive/folders/1tWFUrK0W3pap67jk54AK6DabT3dhr55R?usp=sharing>