Assignment 3

Name: Anmol Singhal
Roll Number: 11940140

Part-1

1.

Video Link **:** LINK

CORE IDEA :

We will have 2 scripts, one for server and one for client. Let the fixed interval, at which client will send the packets be ***i.***

In the start, the client will send a message**,** to calibrate the buffer size at server to fix the packet size. This packet size is taken as input.

After this, the client will send UDP packets to the server at the fixed interval ***i.*** If the client does not receive an acknowledgement before the next packet is to be sent, then the client considers this packet to be lost. Packet Loss as well as an Artificial Delay is implemented on the server side. On receiving the Acknowledgement, the client calculates the ***RTT*** time of this packet and displays it.

At the server, to implement the Packet loss and Artificial Delay, I have used the ***random*** module.

Upon receiving a message, I calculated a random number between 1 and 10 at the server side. If the number is 1, then I do not send the reply for this packet. This is considered as packet loss at the client side. Hence, ***Packet Loss*** occurs with a ***probability*** of ***1/10***.

Using the Random module, a random delay is calculated between **0.0001** second and **0.5 seconds.** Server sleeps for this much amount of time before sending the acknowledgement to cause the artificial delay.

Calculation of ***RTT*** is done at the client side with the help of the ***datetime*** module in ***Python***.

An initial timestamp is calculated before sending the packet and a second timestamp is calculated after receiving the acknowledgement. Difference between these timestamps is the ***RTT.***

I keep a count of packets which are lost and packets which were not lost.

Average RTT in the end is calculated by dividing the sum of all ***RTT's*** by the number of successful packets.

Loss Percentage is calculated by dividing the number of lost packets by total packets and then multiplying it by 100.

In the end, we sent an exit message to the server to tell the server to stop listening to requests and terminate both client and server scripts.

SERVER SIDE SCRIPT : *server.py*

```
import socket
import random
import time
import argparse

# IP at which the server will be listening for calls
localIP     = "0.0.0.0"
publicIP    = socket.gethostbyname(socket.gethostname())

# Take port from command line arguments
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--port", help="Port to listen on", type=int,
default=8080)
args = parser.parse_args()

# Port at which the server will be listening for calls
localPort   = int(args.port)

bufferSize  = 1024
```

```
# Acknowledgement Message
msgFromServer       = "Packet Acknowledged"

# Encoding the message
bytesToSend         = str.encode(msgFromServer)

# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))

# Printing the server details
print(f"UDP server up and listening on address {publicIP} / {localPort}")

# Server Hostname
hostname = socket.gethostname()
print(f"Hostname: {hostname}\n")

# Listen for incoming datagrams
while(True):

    # Receive message from client
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)

    # Destructuring the packet
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]

    # Printing the message and address of the client
    print("Message from Client:{}".format(message))
    print("Client Address:{}".format(address))
    print()

    # Decoding the message
    decodedMessage = message.decode()

    # Checking if the message is for the Calibration of Buffer Size
```

```
    # This packet cannot be dropped
    if(decodedMessage[0] == 'C'):
        bufferSize = int(decodedMessage.split()[1])
        UDPServerSocket.sendto(str.encode("Buffer Size Calibrated at Server."),
address)
        # Skip to next iteration
        continue

    # Checking if the message is for stopping the server
    # This packet cannot be dropped
    if (message == b'exit'):
        UDPServerSocket.sendto(str.encode("Stopping the Server."), address)
        print("Exiting")
        break

    # Artificial delay
    delay = random.randint(1,5000)
    time.sleep(delay/10000)

    # random packet drop with a probability of ~0.1
    num = random.randint(1,10)
    if(num <= 1):
        # Skip to next iteration
        continue

    # Sending the acknowledgement back to the client
    UDPServerSocket.sendto(bytesToSend, address)

# Close the server socket
UDPServerSocket.close()
```

CLIENT SIDE SCRIPT : *client.py*

```
import socket
import time
import datetime
import argparse
```

```
# Message to be sent to the server
msgFromClient       = "Hello Server"
bytesToSend         = str.encode(msgFromClient)

# Command line arguments using argparse
parser = argparse.ArgumentParser()
parser.add_argument('-ip' , dest="serverIP", help="IP address of the server",
type=str)
parser.add_argument('-port', dest="serverPort", help="Port number of the server",
type=int)
parser.add_argument('-n', dest="numberOfMessages", help="Number of messages to be
sent", type=int)
parser.add_argument('-i', dest="interval", help="Interval between messages",
type=float)
parser.add_argument('-s', dest="packetSize" ,help="Size of the packet", type=int)

args = parser.parse_args()

serverIP            = args.serverIP
serverPort          = int(args.serverPort)
numberOfMessages    = int(args.numberOfMessages)
interval            = float(args.interval)
packetSize          = int(args.packetSize)

# Server Address
serverAddressPort   = (serverIP, serverPort)

# Setting the buffer size
bufferSize          = packetSize

# Create a UDP socket at client side
UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

averageRTT = 0

# Setting Timeout
UDPClientSocket.settimeout(interval)

tmsgs = numberOfMessages
```

```
successPackets = 0

# Calibration of Buffer Size
calibrationMessage = str.encode(f"C {packetSize}")
print("\nCalibrating Buffer Size...")
UDPClientSocket.sendto(calibrationMessage, serverAddressPort) # Sending
calibration message to the server
Acknowledgement = UDPClientSocket.recvfrom(bufferSize) # Receive acknowledgement
message from the server
print(Acknowledgement[0].decode())
print()

# Send to server using created UDP socket
while numberOfMessages > 0:

    numberOfMessages -= 1

    print(f"Sending message of {bufferSize} bytes: " + str(tmsgs -
numberOfMessages))

    timestamp1 = datetime.datetime.now().timestamp()

    UDPClientSocket.sendto(bytesToSend, serverAddressPort) # Sending message to
the server

    try:
        msgFromServer = UDPClientSocket.recvfrom(bufferSize) # Receive message
from server
    except socket.timeout: # If the server does not respond within the timeout
period
        print("PACKET LOSS OCCURRED SENDING NEXT PACKET\n")
        continue

    timestamp2 = datetime.datetime.now().timestamp()

    # Printing the acknowledgement message and the RTT
    print("Message from Server {}".format(msgFromServer[0]))
    print(f"RTT for packet {tmsgs - numberOfMessages}:
{timestamp2-timestamp1}\n")
```

```
    averageRTT += timestamp2 - timestamp1

    successPackets += 1
    rtt = timestamp2-timestamp1

    # Sleep for the remaining interval
    time.sleep(max(interval-rtt, 0))


# Telling the server to stop
print("Finished sending.")
UDPClientSocket.sendto(str.encode('exit'), serverAddressPort)
msgFromServer = UDPClientSocket.recvfrom(bufferSize)
print("Final Message from the Server {}".format(msgFromServer[0]))

# Calculating the average RTT and the loss percentage
lossPackets = (tmsgs - successPackets)
lossPercentage = (lossPackets / tmsgs) * 100
averageRTT /= successPackets
print("Average RTT: {}\n".format(averageRTT))
print("Loss Percentage: {}%\n".format(lossPercentage))

# Close the client socket
UDPClientSocket.close()
```

USAGE :

-> Go to the Directory P1_Q1/

-> Please make sure you have all the required modules installed

-> Run the server.py via command:

python server.py -p <port>

-> **-p :** This flag is used to specify the port number. If you do not provide this flag, then port number is defaulted to **8080.**

-> Run the client.py via command:

python client.py -ip <serverIP> -p <serverPort> -i <interval> -n <number of messages to be sent> -s <packet size>

-> All flags are mandatory

-> Server Address can be found in the terminal in which server is running

2.

Video Link **:** **LINK**

Core Idea :

Clients will send UDP Packets to the server at decreasing intervals. The initial interval is given as an input from the user. After that, the intervals decrease at the rate of 5% per packet. For every packet, if the reply/acknowledgement does not come before it is time to send the next packet, then we simply consider it to be a packet loss.

For all the Packets sent, I store the message number, sending time, and the time at which acknowledgement was received. I use this data to calculate the average throughput and delay for each second that packets were being sent.

Average Throughput for each second will be equal to the number of successful messages sent in that second multiplied by packet size multiplied by 2. I multiply by 2 since the packet was sent as well as received. Average Delay is the average RTT of all successful packets sent in that second.

With the help of this data, I then plot the graph of average throughput and delay using the matplotlib library in Python.

SERVER SIDE SCRIPT :

The same server script as above is used for this.

CLIENT SIDE SCRIPT :

```
import datetime
import math
import matplotlib.pyplot as plt
import numpy as np
import socket
import time
import argparse
```

```
# Command line arguments using argparse
parser = argparse.ArgumentParser()
parser.add_argument('-ip' , dest="serverIP", help="IP address of the server",
type=str)
parser.add_argument('-port', dest="serverPort", help="Port number of the server",
type=int)
parser.add_argument('-n', dest="numberOfMessages", help="Number of messages to be
sent", type=int)
parser.add_argument('-i', dest="interval", help="Interval between messages",
type=float)
parser.add_argument('-s', dest="packetSize" ,help="Size of the packet", type=int)

args = parser.parse_args()

serverIP            = args.serverIP
serverPort          = int(args.serverPort)
numberOfMessages    = int(args.numberOfMessages)
interval            = float(args.interval)
packetSize          = int(args.packetSize)

# Function to plot the graph
def plot(X, Y_throughput, Y_delay):

    # Create a Subplot for Throughput and Interval
    plt.subplot(1, 2, 1)
    plt.scatter(X, Y_throughput, c='#cf0412', marker='*', s=70, zorder=10)
    plt.plot(X, Y_throughput, c='#f72533', zorder=5)
    plt.xlabel('Time (in sec)')
    plt.ylabel('Average Throughput (bits/sec)')
    plt.title('Average Throughput (bits/sec)')
    plt.grid(True, zorder=0)
    plt.xticks(np.arange(min(X), max(X)+1, 1.0))

    # Create a Subplot for Delay and Interval
    plt.subplot(1, 2, 2)
    plt.scatter(X, Y_delay, c='#0505a6', marker='*', s=70, zorder=10)
    plt.plot(X, Y_delay, c='#2525f7', zorder=5)
    plt.xlabel('Time (in sec)')
    plt.ylabel('Average Delay (in sec)')
```

```
    plt.title('Average Delay (in sec)')
    plt.grid(True, zorder=0)
    plt.xticks(range(min(X), max(X)+1, 1))

    # Show the graph
    plt.show()

msgFromClient       = "Hello Server."
bytesToSend         = str.encode(msgFromClient)
serverAddressPort   = (serverIP, serverPort)

# Adjusting the buffer size
bufferSize          = packetSize

# Create a UDP socket at client side
UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Initial Timeout
UDPClientSocket.settimeout(interval)

tmsgs = numberOfMessages

# Calibration of Buffer Size
calibrationMessage = str.encode(f"C {packetSize}")
print("\nCalibrating Buffer Size...")
UDPClientSocket.sendto(calibrationMessage, serverAddressPort) # Sending message
to the server
Acknowledgement = UDPClientSocket.recvfrom(bufferSize) # Receive message from
server
print(Acknowledgement[0].decode())
print()


tmsgs = numberOfMessages
messages = {}

# I will use this to get the time relative to the start of the sending process
originalStartTime = datetime.datetime.now().timestamp()
```

```
while numberOfMessages > 0:

    # set timeout for the next packet
    UDPClientSocket.settimeout(interval)

    # send message to server
    numberOfMessages -= 1

    # Timestamp1 is the time when i send the packet and timestamp2 is the time
when i will receive the acknowledgement
    messages[tmsgs - numberOfMessages] = {
        "interval" : interval,
        "timestamp1" : -1,
        "timestamp2" : -1
    }

    print(f"Sending message {tmsgs - numberOfMessages} of size {packetSize}: ",
msgFromClient)

    # Sending time
    timestamp1 = datetime.datetime.now()
    # Send the message to the server
    UDPClientSocket.sendto(bytesToSend, serverAddressPort)
    messages[tmsgs - numberOfMessages]["timestamp1"] = timestamp1.timestamp() -
originalStartTime

    # receive message from server
    try:
        Acknowledgement = UDPClientSocket.recvfrom(bufferSize)
        timestamp2 = datetime.datetime.now()
        messages[tmsgs - numberOfMessages]["timestamp2"] = timestamp2.timestamp()
- originalStartTime
        print("Acknowledgement: ", Acknowledgement[0].decode())
        print()

        # This is also the RTT for this packet
        delay = timestamp2.timestamp() - timestamp1.timestamp()

        print(f"Interval: {interval}")
```

```
        print(f"RTT: {delay}\n")

        # Sleeping for the remaining interval
        time.sleep(max(0, interval - delay))

        # Update the interval
        interval = 0.95*interval

    except socket.timeout:
        # In case of a timeout, timestamp2 will be -1
        print("Timeout, Packet Loss.\n")
        # Update the interval
        interval = 0.95*interval
        # Skip to the next iteration
        continue

interval = 1
UDPClientSocket.settimeout(interval)
# Telling the server that we are done sending and it should stop listening
print("Finished sending, Terminating the connection..")
UDPClientSocket.sendto(str.encode('exit'), serverAddressPort)
msgFromServer = UDPClientSocket.recvfrom(bufferSize)
print("Message from Server {}".format(msgFromServer[0]))

AverageThroughputs = []
AverageDelays = []

# Print all messages
for key in messages.keys():

    print(f"Message {key}")
    print(f"Interval: {messages[key]['interval']}")
    print(f"Timestamp1: {messages[key]['timestamp1']}")
    print(f"Timestamp2: {messages[key]['timestamp2']}\n")

    # Checking, which 'second' was this message was sent in.
    # Since RTT's are very small, we will consider that the acknowledgement was
received at the same second as the message was sent.
    timeSecond = 0
```

```
    if messages[key]['timestamp1'] != int(messages[key]['timestamp1']):
        timeSecond = math.ceil(messages[key]['timestamp1'])
    else:
        timeSecond = int(messages[key]['timestamp1'] + 1)

    # Second, in which no message was sent, will have average throughput of 0 and
average delay of 0
    while(len(AverageThroughputs) < timeSecond):
        AverageThroughputs.append(0)
        # Delays will be stored in the form [total delay, number of packets]
        AverageDelays.append([0, 0])

    # If the packet was not lost and acknowledgement was received, then we will
calculate the throughput and delay for this message.
    if(messages[key]['timestamp2'] != -1):
        # Multiply by 2 since packet was sent from client to server and then from
server to client
        AverageThroughputs[timeSecond-1] += packetSize * 8 * 2
        AverageDelays[timeSecond-1][0] += messages[key]['timestamp2'] -
messages[key]['timestamp1']
        AverageDelays[timeSecond-1][1] += 1

# These will be used to plot the graph
X = []
Y_throughput = []
Y_delay = []

# Print average throughput and delay
for second in range(len(AverageThroughputs)):
    print(f"Average Throughput for second {second + 1}:
{AverageThroughputs[second]}")

    X.append(second+1)
    Y_throughput.append(AverageThroughputs[second])

    if(AverageDelays[second][1] != 0):
        print(f"Average Delay for second {second + 1}: {AverageDelays[second][0]
/ AverageDelays[second][1]}\n")
        Y_delay.append(AverageDelays[second][0] / AverageDelays[second][1])
```

```
    else :
        print(f"Average Delay for second {second + 1}:
{AverageDelays[second][0]}\n")
        Y_delay.append(AverageDelays[second][0])

# Plot the graph
plot(X, Y_throughput, Y_delay)
```

USAGE :

-> Go to the Directory P1_Q2/

-> Please make sure you have all the required modules installed

-> Run the server.py via command:

python server.py -p <port>

-> **-p :** This flag is used to specify the port number. If you do not provide this flag, then port number is defaulted to **8080**

-> Run the client.py via command:

python client.py -ip <serverIP> -p <serverPort> -i <initial Interval> -n <number of messages to be sent> -s <packet size>

-> All flags are mandatory

-> Server Address can be found in the terminal in which server is running

Part-2

Feature 1: File Transfer

Video Link **:** LINK

I have implemented File Transfer from server to client. Server copies the data from the file it intends to transfer in Binary Format and transfers the data over to Client in small chunks. The below scripts show the server transferring a text file. Since, the file format is **.txt** the client script is also written to accept data in .txt format. We can extend the functionality to other file formats by simply changing the file extensions in the below scripts.

Significance : File transfer plays a very important role in everyone's life. For instance, If me and my friend are connected to the IIT Bhilai Wifi Network, and my friend is sitting in

the Academic Block whereas I am sitting in Tech Cafe, I can easily request a file from my friend using these scripts.

Server Side Script : server.py

```
import socket
import sys

localIP = '0.0.0.0'
publicIP = socket.gethostbyname(socket.gethostname())
port = int(sys.argv[1])
s = socket.socket()
s.bind((localIP, port))
filename = input('Enter the file path: ')
# opening the file
try:
    f = open(filename,'rb')
except:
    print('Please make sure you have entered the correct path')
    exit()

# Listening for the incoming connections
s.listen(1)
print(f'Server running at {publicIP}/{port}')
conn, addr = s.accept()
print('Got a new connection from', addr)

# receive authentication key
data = conn.recv(1024)
print('Received Data : ', repr(data))

if data != b'SECRET_KEY':
   conn.send(b'AUTH_ERROR')
   print('closing server')
   conn.close()
   exit()
else:
   conn.send(b'AUTH_SUCCESS')
```

```
print('Sending file...')
l = f.read(1024) # reading file in chunks
while (l):
   conn.send(l)
   print(f'Sent {1024} units of data')
   l = f.read(1024)

# Closing the file
f.close()
print('File sent successfully')
conn.close()
print('Connection closed')
```

CLIENT SIDE SCRIPT : client.py

```
import socket
import sys

s = socket.socket()
serverIP = sys.argv[1]
serverPort = int(sys.argv[2])
s.connect((serverIP, serverPort))

# Authenticating with server
s.send("SECRET_KEY".encode())
# Receive Authentication Result
data = s.recv(1024)

# If Authentication is not successful
if data == b'AUTH_ERROR':
    print("Authentication Error")
    print("Exiting...")
    s.close()
    exit()

# Authentication Successful
print('receiving data...')
```

```
# Creating a new file named RECIEVED_FILE.txt
with open('RECIEVED_FILE.txt', 'wb') as f:

    print('Opened a new file and copying data from server')

    while True:

        # Receiving data from server
        data = s.recv(1024)
        print('Received Data : ', repr(data))

        # File Transfer is complete
        if not data:
            break

        # Writing data to file
        f.write(data)

print('Successfully got the file')
print('Exiting..')
s.close()
print('connection closed')
```

USAGE :

-> Go to the Directory P2/FileTransfer/

-> Please make sure you have all the required modules installed

-> Run the server.py via command:

python server.py <port>

-> **<port> :** This is used to specify the port number on which server will be running.

-> Server's address will be printed in the terminal

-> Run the client.py via command:

python client.py <serverIP> <serverPort>

-> **<serverIP> :** Public IP of the server

-> **<serverPort> :** Port at which server process is running

-> Server Address can be found in the terminal in which server is running

Feature 2: Inter Process Communication

Video Link **:** Link

I have implemented a client server app which allows interaction between multiple processes (clients) running on different addresses. Clients need to know each other's IP and Port number to communicate between each other. A central server directly communicates with all the processes to make this happen.

Significance : Using this Feature, processes can work in tandem with each other to solve one big task. Or this can allow for simple chat between 2 clients who just want to talk to each other over the network. There can be multiple clients on different systems who all run a process in their system and all those processes can work on one big task to reduce the load on every system.

SERVER SIDE SCRIPT :

```
import socket
import sys

# IP at which the server will be listening for calls
localIP     = "0.0.0.0"
publicIP    = socket.gethostbyname(socket.gethostname())

# Port at which the server will be listening for calls
localPort   = int(sys.argv[1])
bufferSize  = 1024

# Acknowledgement Message
SuccessMessage  = "200, UDP Packet Sent Successfully"
ErrorMssg       = "401, Invalid Format"

# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))
```

```
# Printing the server details
print(f"UDP server up and listening on address {publicIP} / {localPort}")

# Server Hostname
hostname = socket.gethostname()
print(f"Hostname: {hostname}\n")

# Listen for incoming datagrams
while(True):

    # Receive the client packet along with the address it is coming from
    message, address = UDPServerSocket.recvfrom(bufferSize)
    print(f"Message: {message.decode()}")
    print(f"Address: {address}")
    message = message.decode()

    # Expected Format-> to : <destination IP> / <destination Port> ; content :
<message>

    # Check if the message is in the expected format

    if message.lower() == 'hii':
        UDPServerSocket.sendto(str(address).encode(), address)
        continue

    if message == 'exit server':
        break

    if(message.count(';') == 0):
        # Send Error Message
        UDPServerSocket.sendto("400".encode(), address)
        continue

    colonIndex = message.index(';')
    header = message[:colonIndex]
    data = message[colonIndex+1:]

    if(header.count(':') != 1 or header.count('/') != 1 or data.count(':') != 1):
```

```
        # Send Error Message
        UDPServerSocket.sendto("401".encode(), address)
        continue

    # Destructuring the header
    header = header.split(':')
    To = header[0]
    targetIP, targetPort = header[1].split('/')

    if(To.strip().lower() != 'to'):
        # Send Error Message
        UDPServerSocket.sendto("402".encode(), address)
        continue

    # strip targetIP and targetPort
    targetIP = targetIP.strip()
    try:
        targetPort = int(targetPort.strip())
    except:
        # Send Error Message
        UDPServerSocket.sendto("403".encode(), address)
        continue

    # Header is valid

    if data.split(':')[0].strip().lower() != 'content':
        # Send Error Message
        UDPServerSocket.sendto("404".encode(), address)
        continue

    message = data.split(':')[1].strip()

    # Data is valid

    # Send message to (target IP, target Port)
    try:
        # Put message in format : <source IP> / <source port> ; content :
<message>
```

```
        UDPServerSocket.sendto(f"{address[0]}/{address[1]} ; content :
{message}".encode(), (targetIP, targetPort))
    except Exception as e:
        # Send Error Message
        UDPServerSocket.sendto("405, Could not send the message to the requested
address".encode(), address)
        continue

    # Send Success Message
    UDPServerSocket.sendto(SuccessMessage.encode(), address)

# Close the server socket
UDPServerSocket.close()
```

Client Side Script :

```
# Implementation of the client side of my email program
# Importing the socket library
import socket
import sys
# Server IP address and port number
ip = sys.argv[1]
port = int(sys.argv[2])
myPort = -1
myIP = -1
initialized = False
# Creating a UDP socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

def init():
    global myIP
    global myPort
    # send a hii message to server
    s.sendto("hii".encode(), (ip, port))
    # Receiving the message from the server
    data, address = s.recvfrom(1024)
    # decoding data
    print(data.decode())
```

```
    data = data.decode()
    data = data[1:-1]
    # extracting IP and Port
    myIP = data.split(",")[0].strip()[1:-1]
    myPort = int(data.split(",")[1].strip())
    # printing the IP and Port
    print(f"My IP is {myIP} and my port is {myPort}")

def send_message():
    if not initialized:
        print("Initialize first.")
        return

    # recipient address
    recipientIP = input("Enter the recipient's IP address: ")
    recipientPort = int(input("Enter the recipient's port number: "))
    # Sending the message to the server
    message = input("Enter your message: ")
    # Message Format-> to : <recipientIP> / <recipientPort> ; content : <message>
    # Formatting the message
    mymessage = "to:" + recipientIP + "/" + str(recipientPort) + ";content:" +
message
    # Sending the message to the server
    s.sendto(mymessage.encode(), (ip, port))
    # Receiving the message from the server
    data, address = s.recvfrom(1024)
    if data.decode().startswith('200'):
        print("Message Sent")
    else:
        print("Message not sent")
        print("Error :", data.decode())

def receive_message():
    # Receiving the message from the server
    if not initialized:
        print("Initialize first.")
        return
    print(f"Waiting for the message at {myIP}/{myPort}...")
    print()
```

```
    data, address = s.recvfrom(1024)
    print(f"Message received from {address[0]}/{address[1]}")
    # Printing the message
    print("Message: ", data.decode())

while True:
    print("1. Initialize")
    print("2. Send Message")
    print("3. Receive Message")
    print("4. Exit")
    print("5. Close Server")
    choice = int(input("Enter your choice (1/2/3/4/5): "))
    if choice == 1:
        if initialized:
            print("Already initialized.")
        else:
            init()
            initialized = True
    elif choice == 2:
        if not initialized:
            print("Initialize first.")
        else:
            send_message()
    elif choice == 3:
        if not initialized:
            print("Initialize first.")
        else:
            receive_message()
    elif choice == 4:
        break
    elif choice == 5:
        s.sendto("exit server".encode(), (ip, port))
        break
    else:
        print("Invalid choice.")
```

USAGE :

-> Go to the Directory P2/InterprocessCommunication/

-> Please make sure you have all the required modules installed

-> Run the server.py via command:

python server.py <port>

-> **<port> :** This is used to specify the port number on which server will be running.

-> Server's address will be printed in the terminal

-> To connect clients with the server, run the client.py via command:

python client.py <serverIP> <serverPort>

-> **<serverIP> :** Public IP of the server

-> **<serverPort> :** Port at which server process is running

-> Server Address can be found in the terminal in which server is running

-> In the client terminal, there will be a menu which will give 5 options, i.e. to Initialize client, Send Message to some process, Receive message, Exit, or Close the server

-> Every Client will have to Initialize first with the server before it can send or receive messages to any other client. Upon Initializing, Client's IP and Port (which is randomly generated) are printed in the client's Terminal.

Part-3

Video Link **:** LINK

Core Idea :

In python, we can use **socket.addrinfo()** to check the address family of a given IP address. I take the IP address and check its family and create the socket respectively. Hence, My app supports both ***IPv4*** and ***IPv6*** addresses.

Server Side Script :

```
import socket
import sys

# Command line arguments
host=sys.argv[1]
port=sys.argv[2]
```

```
# Checking the address information
addr_info = socket.getaddrinfo(host, int(port))
# Host and Port
host = addr_info[0][4][0]
port = int(addr_info[0][4][1])
# Address Family
family = addr_info[0][0]
# Socket Type
TCP_TYPE = socket.SOCK_STREAM
# Create a TCP socket
s = socket.socket(family=family, type=TCP_TYPE)
# Bind the socket to the address
s.bind((host, port))
# Listen for incoming connections
s.listen(5)
print("Server is listening on {}:{}".format(host, port))

while(True):
    # Accept a connection
    conn, addr = s.accept()
    print('Got a connection from the address : ', addr)
    # Communicate with the client
    with conn:
        while True:
            # Receive data from the client
            data = conn.recv(1024)
            print('Received Message: ' + data.decode('utf-8'))

            if not data or data.decode('utf-8') == 'exit':
                print('Closing Connection')
                break

            # Send data to the client
            print('Sending Acknowledgement')
            conn.sendall("Acknowledgement".encode('utf-8'))
```

Client Side Script :

```
import socket
import sys

# Command line arguments
host=sys.argv[1]
port=sys.argv[2]

# Checking address information
addr_info = socket.getaddrinfo(host, int(port))

# Host and Port
host = addr_info[0][4][0]
port = int(addr_info[0][4][1])

# Address family
family = addr_info[0][0]

# Socket type
TCP_TYPE = socket.SOCK_STREAM

# Create a socket
s = socket.socket(family=family, type=TCP_TYPE)

# Connect to the server
s.connect((host, int(port)))

# Send 5 messages
numberOfMessages = 5
while numberOfMessages > 0:
    numberOfMessages -= 1
    s.sendall("Hii Server".encode('utf-8'))
    Acknowledgement = s.recv(1024)
    print('Acknowledgement Received: ' + Acknowledgement.decode('utf-8'))

# Tell the server to stop
s.sendall("exit".encode('utf-8'))
```

```
# Close the socket
s.close()
print('Connection closed')
```

USAGE :

Go to the Directory P3/

-> Please make sure you have all the required modules installed

-> Run the server.py via command:

python server.py <serverIP> <serverPort>

-> To connect clients with the server, run the client.py via command:

python client.py <serverIP> <serverPort>

-> **<serverIP> :** IP address of the server. Can be IPv6 as well as IPv4

-> **<serverPort> :** Port at which server process is running