# Distributed Computing EXP-1

## Name:Atharva Pawar

**Roll NO:9427**

**Batch:D**

**Lab Outcome:**

Develop test and debug using Message-Oriented Communication or RPC/RMIbased

client-server programs

**Theory:**

**Client-Server Model**

The client-server model is a decentralized computing architecture that involves dividing

tasks or workloads between servers and clients. The servers provide centralized resources

and services, while clients request and utilize these resources. This model enables clients to

access shared data and services, while servers can manage and control access to resources. It

helps to distribute the workload and reduce the burden on any single device or component.

The client-server model is used in a variety of applications, including web services, email,

and database systems, and is particularly useful for supporting large numbers of users or

clients. This architecture is also flexible and scalable, making it a popular choice for many

organizations.

**Socket**

Socket programming is a way of connecting two nodes on a network to communicate with

each other. One socket(node) listens on a particular port at an IP, while the other socket

reaches out to the other to form a connection. The server forms the listener socket while the

client reaches out to the server.

They are the real backbones behind web browsing. In simpler terms, there is a server and a

client. Socket programming is started by importing the socket library and making a simple

socket.

Function Call

Description

Socket()

To create a socket

Bind()

It’s a socket identification

like a telephone number to

contact

Listen()

Ready to receive a

connection

Connect()

Ready to act as a sender

Accept()

Confirmation, it is like

accepting to receive a call

from a sender

Write()

To send data

Read()

To receive data

Close()

To close a connection

**Single-threading**

Single-threading is a computer programming model that processes and executes tasks

sequentially in a single thread of execution. In this model, the processor runs one task at a

time and is unable to execute multiple tasks simultaneously. As a result, single-threaded

applications can only perform one operation at a time and cannot take advantage of multi

core processors. Single-threading is simple and easy to implement, but it can be limited in

terms of performance, particularly for complex or demanding tasks. However, it is still

commonly used in applications where the task execution time is relatively short and

predictable, or where the need for multi-threading is low. Single-threaded programming can

also be used to ensure that tasks are executed in a specific order, which can be important in

certain applications, such as financial transactions.

**Multi-threading**

Multithreading is the ability of a program or an operating system process to manage its

useby more than one user at a time and to even manage multiple requests by the same user

without having to have multiple copies of the programming running in the computer. Each

user request for a program or system service(and here a user can also be another program)

is kept track of as a thread with a separate identity. As programs work on behalf of the initial

request for that thread and are interrupted by other requests, the status of work on behalf

ofthat thread is kept track of until the work is completed.

**Developing Client-Server Application**

In this experiment we aim at developing Client–Server application using multithreading

concepts. In this application server handles the requests from multiple clients.

For this experiment socket programing is used. In client program the parameters like port

number, server name are taken in order to connect with server. In the server program it

acceptsthe connection from client and assigns them to new connection handler object. Then

one thread at server side handles new connections and another thread is for clients.

Multithread client server

example:Web client:

• Web browser scans an incoming HTML page, and finds that more files need to

befetched

• Each file is fetched by a separate thread, each doing a (blocking) HTTP request

• As files come in, the browser displays them

A multithreaded server organized in a dispatcher/worker model.

**Steps to run the application**

1. Start server program. It will be ready to accept connection from the client.

2. On another terminal start client program and send some message to server.

3. Server will display the output.**(Code & Output)**

**1 client 1 server**

**Client.py**

universe@lenovo12:~/Desktop/afdz$ cat client.py

import socket

def client\_program():

host = socket.gethostname() # as both code is running on same pc

port = 5000 # socket server port number

client\_socket = socket.socket() # instantiate

client\_socket.connect((host, port)) # connect to the server

message = input(“ -> “) # take input

while message.lower().strip() != ‘bye’:

client\_socket.send(message.encode()) # send message

data = client\_socket.recv(1024).decode() # receive response

print(‘Received from server: ‘ + data) # show in terminal

message = input(“ -> “) # again take input

client\_socket.close() # close the connection

if \_\_name\_\_ == ‘\_\_main\_\_’:

client\_program()

universe@lenovo12:~/Desktop/afdz$

**Server.py**

universe@lenovo12:~/Desktop/afdz$ cat server.py

import socket

def server\_program():

# get the hostname

host = socket.gethostname()

port = 5000 # initiate port no above 1024

server\_socket = socket.socket() # get instance

# look closely. The bind() function takes tuple as argument

server\_socket.bind((host, port)) # bind host address and port together

# configure how many client the server can listen simultaneously

server\_socket.listen(2)

conn, address = server\_socket.accept() # accept new connection

print(“Connection from: “ + str(address))

while True:

# receive data stream. It won’t accept data packet greater than 1024 bytes

data = conn.recv(1024).decode()

if not data:

# if data is not received breakbreak

print(“from connected user: “ + str(data))

data = input(‘ -> ‘)

conn.send(data.encode()) # send data to the client

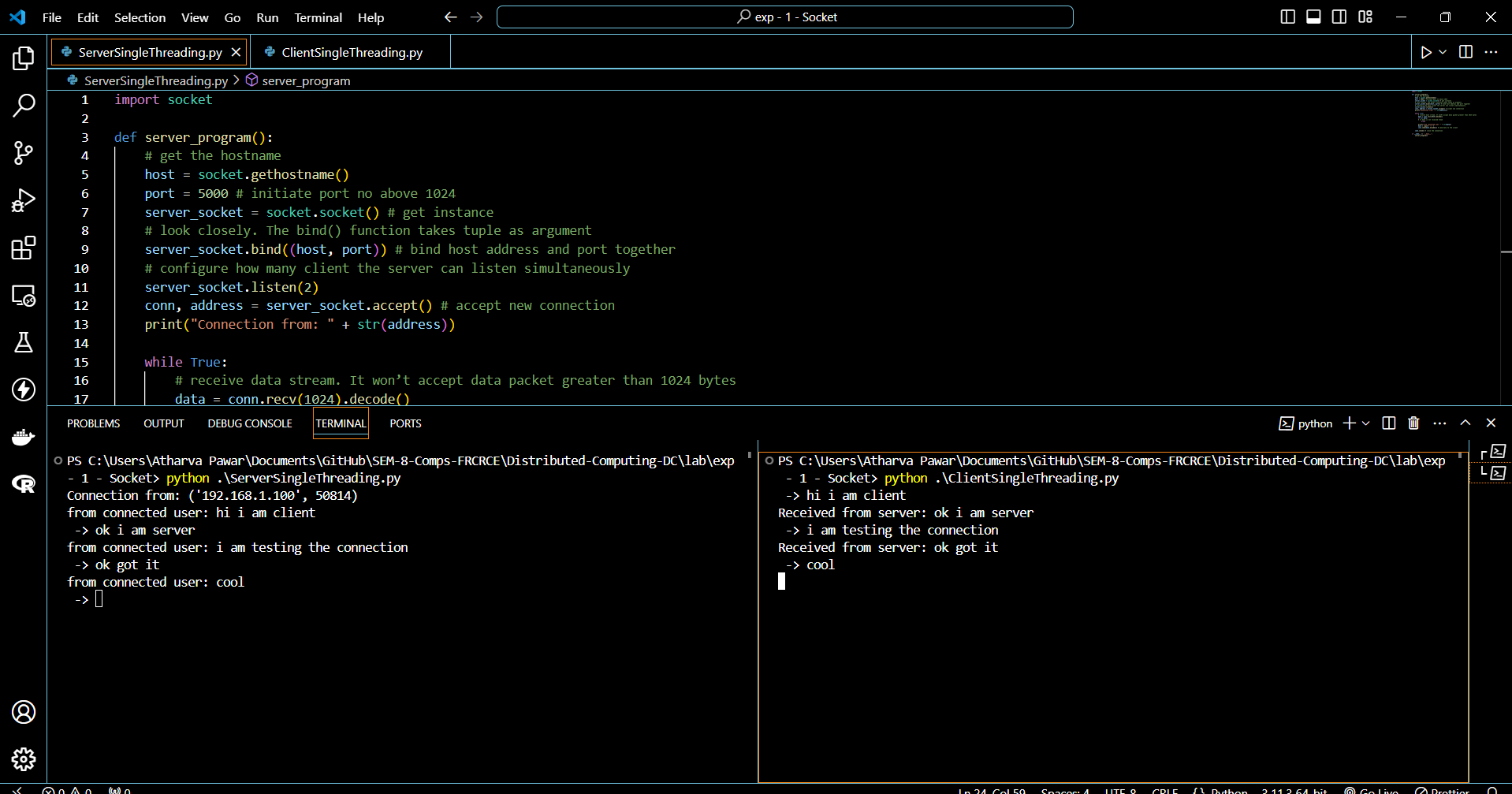
conn.close() # close the connection

if \_\_name\_\_ == ‘\_\_main\_\_’:

server\_program()

regardless of the underlying communication mechanisms or protocols used. This shields the client from the complexities of network communication and allows for easier integration of remote services.

Failure Transparency: Stubs can handle failures and errors transparently, shielding the client from the details of network failures, service unavailability, or other issues. Depending on the implementation, stubs may provide error handling and retry mechanisms to improve the robustness of the distributed system.



**Multithreading**

**2 client 1 server**

**Client1.py**

import socket

host = socket.gethostname()port = 2004

BUFFER\_SIZE = 2000

MESSAGE = input(“tcpClientA: Enter message/ Enter exit:”)

tcpClientA = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

tcpClientA.connect((host, port))

while MESSAGE != ‘exit’:

tcpClientA.send(MESSAGE.encode())

data = tcpClientA.recv(BUFFER\_SIZE).decode()

print (“ Client2 received data:”, data)

MESSAGE = input(“tcpClientA: Enter message to continue/ Enter exit:”)

tcpClientA.close()

**Client2.py**

import socket

host = socket.gethostname()

port = 2004

BUFFER\_SIZE = 2000

MESSAGE = input(“tcpClientB: Enter message/ Enter exit:”)

tcpClientB = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

tcpClientB.connect((host, port))

while MESSAGE != ‘exit’:

tcpClientB.send(MESSAGE.encode())

data = tcpClientB.recv(BUFFER\_SIZE).decode()

print (“ Client received data:”, data)

MESSAGE = input(“tcpClientB: Enter message to continue/ Enter exit:”)

tcpClientB.close()

**Server.py**

import socket

from threading import Thread

from socketserver import ThreadingMixIn

# Multithreaded Python server : TCP Server Socket Thread Pool

class ClientThread(Thread):

def \_\_init\_\_(self,ip,port):

Thread.\_\_init\_\_(self)

self.ip = ip

self.port = port

print (“[+] New server socket thread started for “ + ip + “:” + str(port))

def run(self):

while True :

data = conn.recv(2048).decode()

print (“Server received data:”, data)

MESSAGE = input(“Multithreaded Python server : Enter Response from Server/Enter exit:”)

if MESSAGE == ‘exit’:

break

conn.send(MESSAGE.encode()) # echo

# Multithreaded Python server : TCP Server Socket Program Stub

TCP\_IP = ‘0.0.0.0’

TCP\_PORT = 2004

BUFFER\_SIZE = 20 # Usually 1024, but we need quick response

tcpServer = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

tcpServer.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)tcpServer.bind((TCP\_IP, TCP\_PORT))

threads = []

while True:

tcpServer.listen(4)

print (“Multithreaded Python server : Waiting for connections from TCP clients…”)

(conn, (ip,port)) = tcpServer.accept()

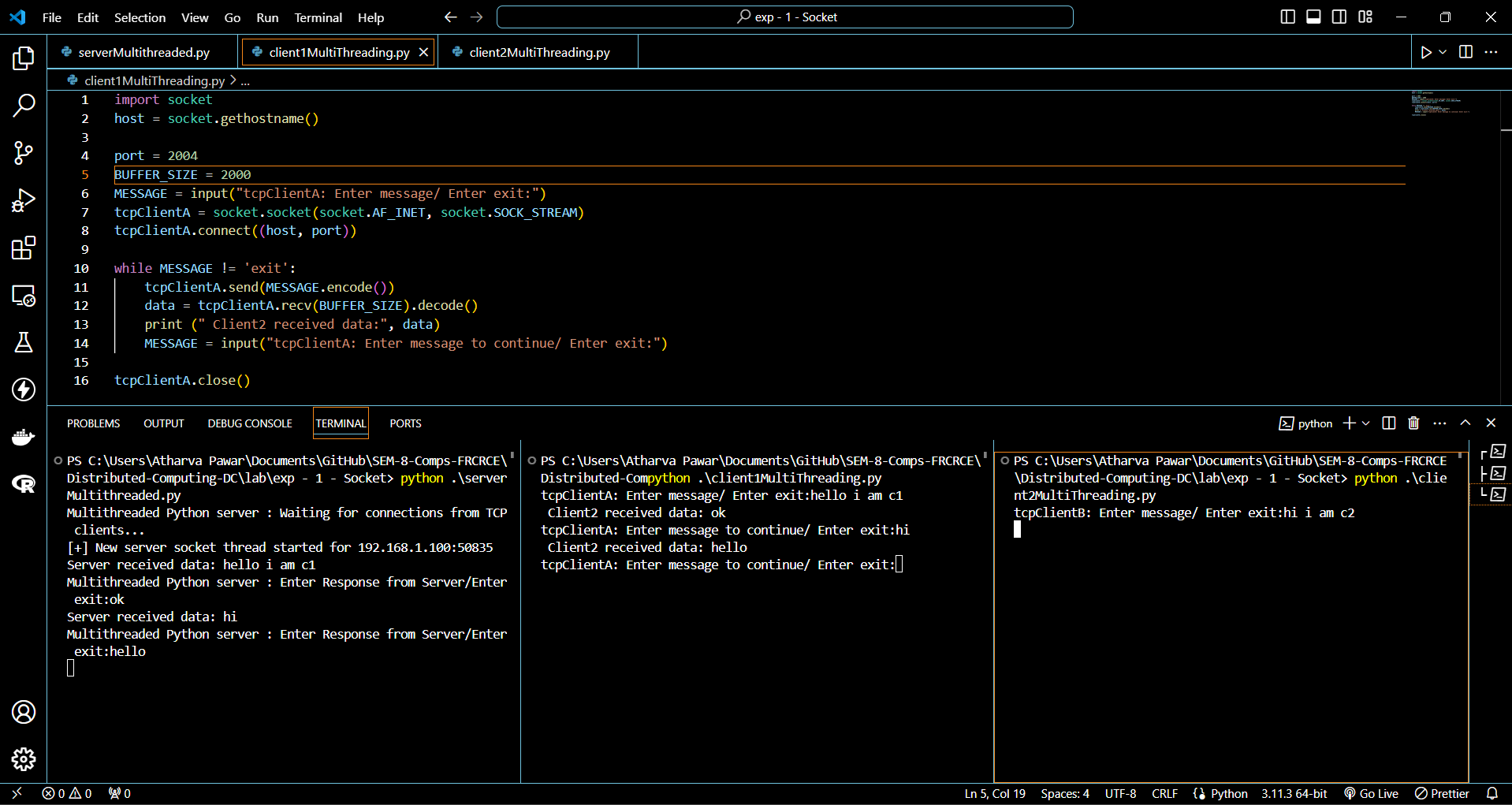
newthread = ClientThread(ip,port)

newthread.start()

threads.append(newthread)

for t in threads:

t.join()



**Conclusions :**

The experiment demonstrated the difference between single threading and multithreading in

a client-server model. Single threading showed limitations in handling multiple clients

simultaneously, whereas multithreading improved the performance by handling multiple

clients at the same time. This highlights the importance of multithreading in server design for improved efficiency and scalability.

**Post lab Questions:**

1. Enlist the socket primitives.

2. What are the advantages of a Multithreaded Server?

3. With an example, explain the concept of multithreaded clients.

4. What are the motivations for using threads.

5.What are the typical models for Organizing threads.