VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Team Member –

Rahul Mahajan – 17BCE1063

Yash Srivastava – 17BCE1069

Project title – Inter Process Communication using sockets

Abstract –

In our project we are going to implement Inter process communication (IPC) through socket programming. It 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 other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server. Socket are very easy, versatile and are basic component of IPC between two processes. Sockets are end points to communication to which name can be bound. Socket exist in communication domain. IPC socket is a data communication end point for exchanging data between process, executing on same host OS. We are going to make two application process on same as well as different machines and establish socket connection between two and establish chatting interface between two. Also we have shown process and cpu details of all the clients in a common shared area connected to the same server.

Introduction –

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data. There are numerous reasons for providing an environment or situation which allows process co-operation:

- Information sharing: Since some users may be interested in the same piece of information (for example, a shared file), you must provide a situation for allowing concurrent access to that information.
- Computation speedup: If you want a particular work to run fast, you must break it into sub-tasks where each of them will get executed in parallel with the other tasks. Note that such a speed-up can be attained only when the computer has compound or various processing elements like CPUs or I/O channels.
- Modularity: You may want to build the system in a modular way by dividing the system functions into split processes or threads.
- Convenience: Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.

Working together with multiple processes, require an interprocess communication (IPC) method which will allow them to exchange data along with various information. There are two primary models of interprocess communication:

1. Shared memory

2. Message passing

In the shared-memory model, a region of memory which is shared by cooperating processes gets established. Processes can be then able to exchange information by reading and writing all the data to the shared region. In the message-passing form, communication takes place by way of messages exchanged among the cooperating processes.

Motivation –

All modern OS (Windows 95 and later, OS X from 10.0, almost all Unix-like systems of any age) isolate memory of processes from each other, giving each process a sandbox that runs its process as if it runs within its own virtual memory with illusion of possessing physical CPUs of the machine (though they are available to process only when OS schedules that). So, one process cannot access memory of another process by default. Although processes often must exchange data or at least signals to co-operate. So some mechanisms that make that exchange possible are required.

The most obvious one is shared memory, when some areas of virtual memory of two or more processes is the same physical memory. It requires a special setup procedure and proper synchronization for data consistency.

Nowadays chatting features are trending in various applications. These chatting features are a form of IPC and so we are interested in knowing about the basics of how IPC is working in these areas.

Module–wise implementation details –

1. Server –

```
#!/usr/bin/env python3
"""Server for multithreaded (asynchronous) chat application."""

from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
```

We will be using TCP sockets for this purpose, and therefore we use AF_INET and SOCK_STREAM flags. We use them over UDP sockets because they're more telephonic, where the recipient has to approve the incoming connection before communication begins, and UDP sockets are more post-mail sort of thing (anyone can send a mail to any recipient whose address s/he knows), so they don't really require an establishment of connection before communication can happen. Clearly, TCP suit more to our purpose than UDP sockets, therefore we use them.

After imports, we set up some constants for later use:

```
clients = {}
addresses = {}
HOST = ''
PORT = 33000
BUFSIZ = 1024
ADDR = (HOST, PORT)
SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)
```

Now, we break our task of serving into accepting new connections, broadcasting messages and handling particular clients. Let's begin with accepting connections:

```
def accept_incoming_connections():
    """Sets up handling for incoming clients."""
    while True:
        client, client_address = SERVER.accept()
        print("%s:%s has connected." % client_address)
        client.send(bytes("Greetings from the cave!"+
                         "Now type your name and press enter!", "utf8"))
        addresses[client] = client_address
        Thread(target=handle_client, args=(client,)).start()
```
This is just a loop that waits forever for incoming connections and as soon as it gets one, it logs the connection (prints some of the connection details) and sends the connected client a welcome message. Then it stores the client's address in the addresses dictionary and later starts the handling thread for that client. Of course, we haven't yet defined the target function handle_client() for that, but here's how we do it:

```
def handle_client(client):  # Takes client socket as argument.
    """Handles a single client connection."""
     name = client.recv(BUFSIZ).decode("utf8")
    welcome = 'Welcome %s! If you ever want to quit, type {quit} to
exit.' % name
    client.send(bytes(welcome, "utf8"))
    msg = "%s has joined the chat!" % name
    broadcast(bytes(msg, "utf8"))
    clients[client] = name
     while True:
        msg = client.recv(BUFSIZ)
        if msg != bytes("{quit}", "utf8"):
            broadcast(msg, name+": ")
        else:
            client.send(bytes("{quit}", "utf8"))
            client.close()
            del clients[client]
            broadcast(bytes("%s has left the chat." % name, "utf8"))
            break
```
Naturally, after we send the new client the welcoming message, it will reply with the name s/he wants to use for further communication. In the handle_client() function, the first task we do is we save this name, and then send another message to the client, regarding further instructions. After this comes the main loop for communication: here we recieve further messages from the client

and if a message doesn't contain instructions to quit, we simply broadcast the messsage to other connected clients (we'll be defining the broadcast method in a moment). If we do encounter a message with exit instructions (i.e., the client sends a {quit}), we echo back the same message to the client (it triggers close action on the client side) and then we close the connection socket for it. We then do some cleanup by deleting the entry for the client, and finally give a shoutout to other connected people that this particular person has left the conversation.
Now comes our broadcast() function:

```
def broadcast(msg, prefix=""):  # prefix is for name identification.
    """Broadcasts a message to all the clients."""
     for sock in clients:
        sock.send(bytes(prefix, "utf8")+msg)
```

This is pretty much self-explanatory; it simply sends the msg to all the connected clients, and prepends an optional prefix if necessary. We do pass a prefix to broadcast() in our handle_client() function, and we do it so that people can see exactly who is the sender of a particular message.

That was all the required functionalities for our server. Finally, we put in some code for starting our server and listening for incoming connections:

```
if __name__ == "__main__":
    SERVER.listen(5)  # Listens for 5 connections at max.
    print("Waiting for connection...")
    ACCEPT_THREAD = Thread(target=accept_incoming_connections)
    ACCEPT_THREAD.start()  # Starts the infinite loop.
    ACCEPT_THREAD.join()
    SERVER.close()
```

We join() ACCEPT_THREAD so that the main script waits for it to complete and doesn't jump to the next line, which closes the server.

2. Client –

We use Tkinter, Python’s “batteries included” GUI building tool for our purpose. Let’s do some imports first:

```
#!/usr/bin/env python3
"""Script for Tkinter GUI chat client."""
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
import tkinter
```

Now we’ll write functions for handling sending and receiving of messages. We start with receive:

```
def receive():
    """Handles receiving of messages."""
    while True:
        try:
            msg = client_socket.recv(BUFSIZ).decode("utf8")
            msg_list.insert(tkinter.END, msg)
        except OSError:  # Possibly client has left the chat.
            break
```

Why an infinite loop again? Because we’ll be receiving messages quite non-deterministically, and independent of how and when we send the messages. We don’t want this to be a walkie-talkie chat app which can only either send *or* receive at a time; we want to receive messages when we can, and send them when we want. The functionality within the loop is pretty straightforward; the recv() is the blocking part. It stops execution until it receives a message, and when it does, we move ahead and append the message to msg_list. We will soon define msg_list, which is basically a Tkinter feature for displaying the list of messages on the screen.
Next, we define the send() function:

```
def send(event=None):  # event is passed by binders.
    """Handles sending of messages."""
    msg = my_msg.get()
    my_msg.set("")  # Clears input field.
    client_socket.send(bytes(msg, "utf8"))
    if msg == "{quit}":
        client_socket.close()
        top.quit()
```

We're using event as an argument because it is implicitly passed by Tkinter when the send button on the GUI is pressed. my_msg is the input field on the GUI, and therefore we extract the message to be sent usin g msg = my_msg.get(). After that, we clear the input field and then send the message to the server, which, as we've seen before, broadcasts this message to all the clients (if it's not an exit message). If it is an exit message, we close the socket and then the GUI app (via top.close())

We define one more function, which will be called when we choose to close the GUI window. It is a sort of cleanup-before-close function and shall close the socket connection before the GUI closes:

```
def on_closing(event=None):
    """This function is to be called when the window is closed."""
    my_msg.set("{quit}")
    send()
```

This sets the input field to {quit} and then calls send(), which then works as expected. Now we start building the GUI, in the main namespace (i.e., outside any function). We start by defining the top-level widget and set its title:

```
top = tkinter.Tk()
top.title("Chatter")
```

Then we create a frame for holding the list of messages. Next, we create a string variable, primarily for storing the value we get from the input field (which we shall define soon). We set that variable to "Type your messages here." to prompt the user for writing their message. After that, we create a scrollbar for scrolling through this message frame. Here's the code:

```
messages_frame = tkinter.Frame(top)
my_msg = tkinter.StringVar()  # For the messages to be sent.
my_msg.set("Type your messages here.")
scrollbar = tkinter.Scrollbar(messages_frame)  # To navigate
through past messages.
```

Now we define the message list which will be stored in messages_frame and then pack in (at the appropriate places) all the stuff we've created till now:

```
msg_list = tkinter.Listbox(messages_frame, height=15, width=50,
yscrollcommand=scrollbar.set)
scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
msg_list.pack(side=tkinter.LEFT, fill=tkinter.BOTH)
msg_list.pack()
messages_frame.pack()
```

After this, we create the input field for the user to input their message, and bind it to the string variable defined above. We also bind it to the send()function so that whenever the user presses return, the message is sent to the server. Next, we create the send button if the user wishes to send their messages by clicking on it. Again, we bind the clicking of this button to the send() function. And yes, we also pack all this stuff we created just now. Furthermore, don't forget to make use of the cleanup function on_closing()which should be called when the user wishes to close the GUI window. We do that by using the protocol method of top. Here's the code for all of this:

```
entry_field = tkinter.Entry(top, textvariable=my_msg)
entry_field.bind("<Return>", send)
entry_field.pack()
send_button = tkinter.Button(top, text="Send", command=send)
send_button.pack()
top.protocol("WM_DELETE_WINDOW", on_closing)
```

3. Connection –

For connection, we have to ask the user for the server's address. We have done that by simply using input(), so the user is greeted with some command line prompt asking for host address before the GUI begins.

Here's my code:

```
HOST = input('Enter host: ')
PORT = input('Enter port: ')
if not PORT:
    PORT = 33000  # Default value.
else:
    PORT = int(PORT)
BUFSIZ = 1024
ADDR = (HOST, PORT)
client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)
```

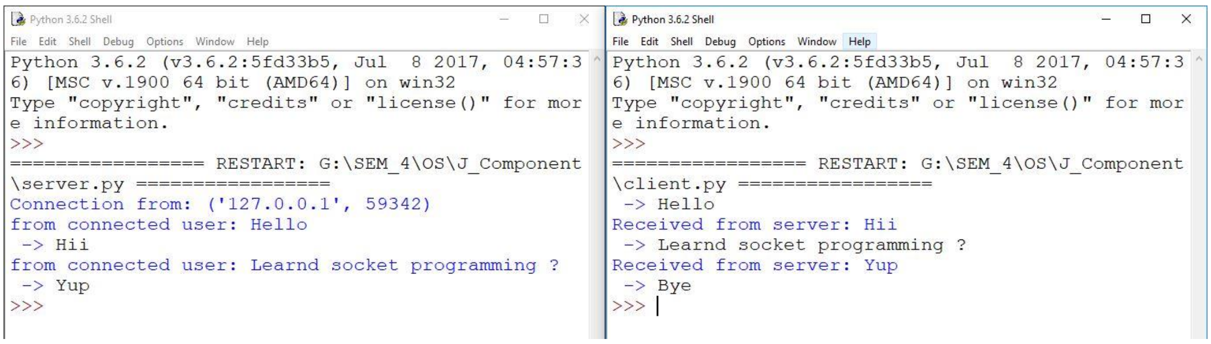
Once we get the address and create a socket to connect to it, we start the thread for receiving messages, and then the main loop for our GUI application:

receive_thread = Thread(target=receive)
receive_thread.start()
tkinter.mainloop() # Starts GUI execution.

4. Communication – First we connect server and all the client through the same network. After that first we run server and then all the client runs their program and ask for host IP address to join the shared area for communication. After entering the host IP address the clients will be connected and the GUI will appear in each client and process communication continues.

Experimental results & Discussions

1. Communication in same machine –



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:3
6) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for mor
e information.
>>>
================= RESTART: G:\SEM_4\OS\J_Component
\server.py =================
Connection from: ('127.0.0.1', 59342)
from connected user: Hello
 -> Hii
from connected user: Learnd socket programming ?
 -> Yup
>>>
```

```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:3
6) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for mor
e information.
>>>
================= RESTART: G:\SEM_4\OS\J_Component
\client.py =================
 -> Hello
Received from server: Hii
 -> Learnd socket programming ?
Received from server: Yup
 -> Bye
>>>
```

Server.py –

import socket

def server_program():

    host = "localhost" # get the hostname

    port = 5000 # initiate port no above 1024

    server_socket = socket.socket()

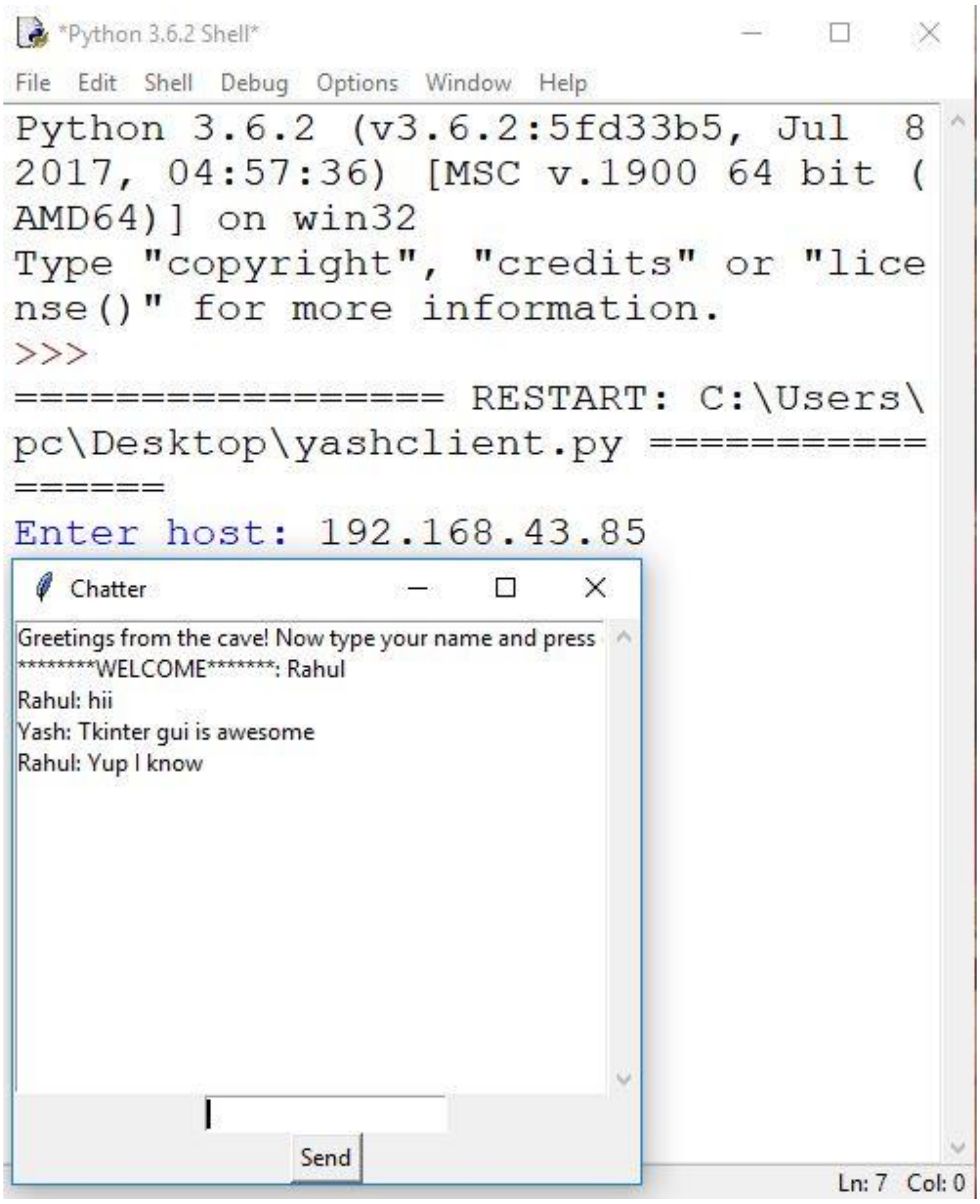
    server_socket.bind((host, port)) # bind host address and port together

```
    # 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. (data packet <= 1024 bytes)
        data = conn.recv(1024).decode()
        if not data:
            # if data is not received break
            break
        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()
```

Client.py –

```
import socket
def client_program():
    host = "localhost"  # 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()
```

2. Communication between different machine providing a chat room –



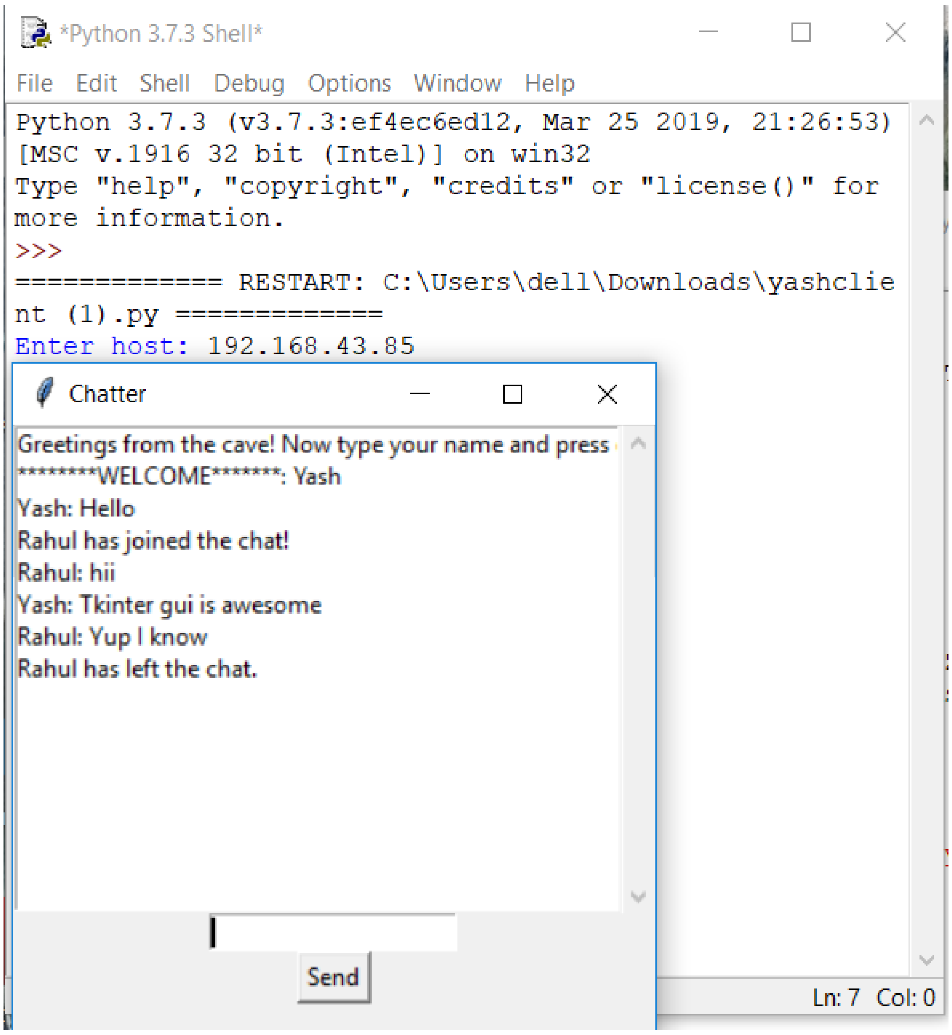
Python 3.6.2 Shell

File Edit Shell Debug Options Window Help

```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8
2017, 04:57:36) [MSC v.1900 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "lice
nse()" for more information.
>>>
================= RESTART: C:\Users\
pc\Desktop\yashclient.py ===========
======
Enter host: 192.168.43.85
```

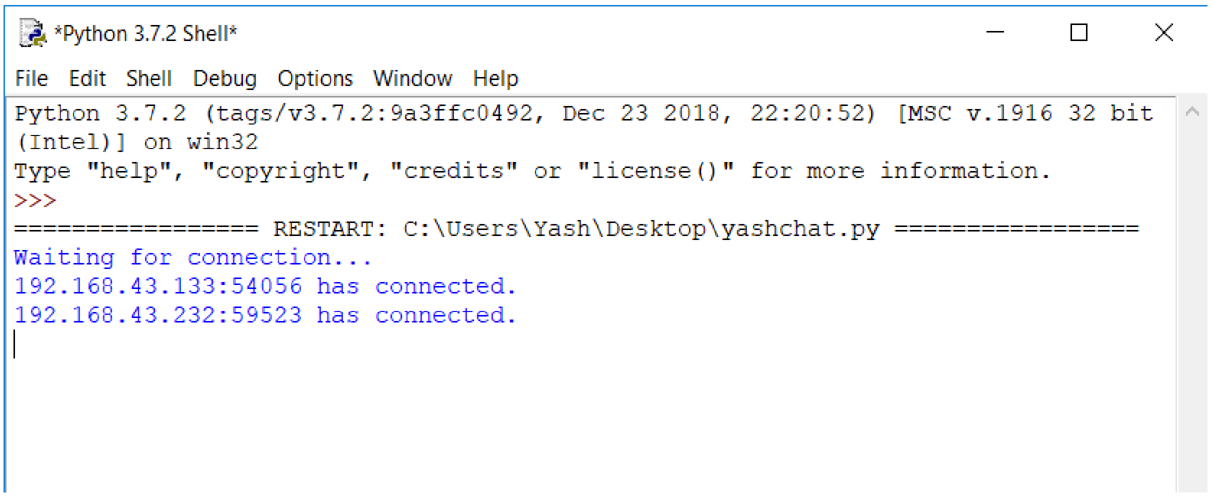
Chatter

```
Greetings from the cave! Now type your name and press
********WELCOME*******: Rahul
Rahul: hii
Yash: Tkinter gui is awesome
Rahul: Yup I know
```



```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for
more information.
>>>
============= RESTART: C:\Users\dell\Downloads\yashclie
nt (1).py =============
Enter host: 192.168.43.85
```

```
Greetings from the cave! Now type your name and press
********WELCOME*******: Yash
Yash: Hello
Rahul has joined the chat!
Rahul: hii
Yash: Tkinter gui is awesome
Rahul: Yup I know
Rahul has left the chat.
```



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================= RESTART: C:\Users\Yash\Desktop\yashchat.py =================
Waiting for connection...
192.168.43.133:54056 has connected.
192.168.43.232:59523 has connected.
```

Server_code.py -

```
#!/usr/bin/env python3
"""Server for multithreaded (asynchronous) chat application."""
```

```
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread



def accept_incoming_connections():
    """Sets up handling for incoming clients."""
    while True:
        client, client_address = SERVER.accept()
        print("%s:%s has connected." % client_address)
        client.send(bytes("Greetings from the cave! Now type your
name and press enter!", "utf8"))
        addresses[client] = client_address
        Thread(target=handle_client, args=(client,)).start()



def handle_client(client):  # Takes client socket as argument.
    """Handles a single client connection."""

    name = client.recv(BUFSIZ).decode("utf8")
    welcome = '********WELCOME*******\n: %s ' % name
    client.send(bytes(welcome, "utf8"))
    msg = "%s has joined the chat!" % name
    broadcast(bytes(msg, "utf8"))
    clients[client] = name
    while True:
        msg = client.recv(BUFSIZ)
        if msg != bytes("{quit}", "utf8"):
            broadcast(msg, name+": ")
        else:
            client.send(bytes("{quit}", "utf8"))
            client.close()
            del clients[client]
            broadcast(bytes("%s has left the chat." % name, "utf8"))
            break


def broadcast(msg, prefix=""):  # prefix is for name identification.
    """Broadcasts a message to all the clients."""

    for sock in clients:
        sock.send(bytes(prefix, "utf8")+msg)


clients = {}
addresses = {}

HOST = ''
```

```
PORT = 33000
BUFSIZ = 1024
ADDR = (HOST, PORT)

SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)

if __name__ == "__main__":
    SERVER.listen(5)
    print("Waiting for connection...")
    ACCEPT_THREAD = Thread(target=accept_incoming_connections)
    ACCEPT_THREAD.start()
    ACCEPT_THREAD.join()
```

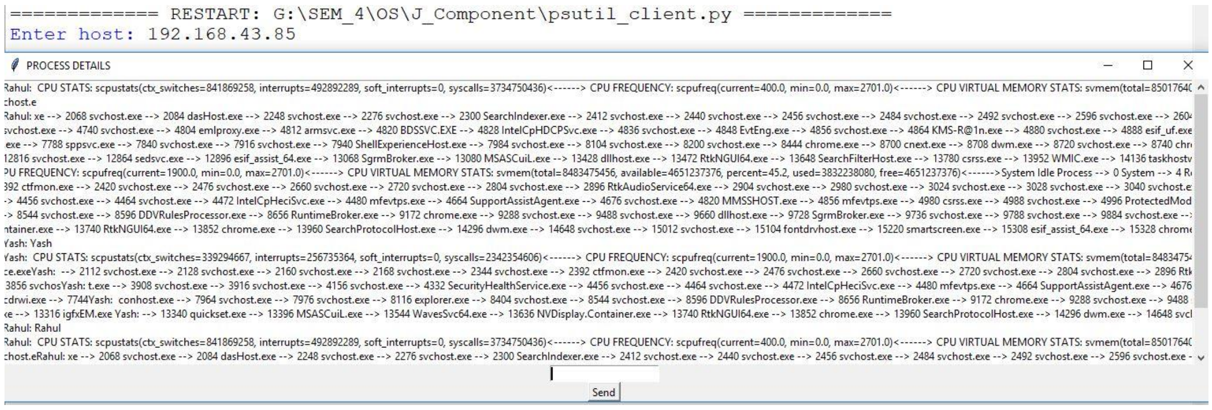
Client_code.py –

```
#!/usr/bin/env python3
"""Script for Tkinter GUI chat client."""
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
import tkinter


def receive():
    """Handles receiving of messages."""
    while True:
        try:
            msg = client_socket.recv(BUFSIZ).decode("utf8")
            msg_list.insert(tkinter.END, msg)
        except OSError:  # Possibly client has left the chat.
            break


def send(event=None):  # event is passed by binders.
    """Handles sending of messages."""
    msg = my_msg.get()
    my_msg.set("")  # Clears input field.
    client_socket.send(bytes(msg, "utf8"))
    if msg == "{quit}":
        client_socket.close()
        top.quit()



def on_closing(event=None):
    """This function is to be called when the window is closed."""
    my_msg.set("{quit}")
    send()

top = tkinter.Tk()
top.title("Chatter")
```

```
messages_frame = tkinter.Frame(top)
my_msg = tkinter.StringVar()  # For the messages to be sent.
my_msg.set("Type your messages here.")
scrollbar = tkinter.Scrollbar(messages_frame)  # To navigate
through past messages.
# Following will contain the messages.
msg_list = tkinter.Listbox(messages_frame, height=15, width=50,
yscrollcommand=scrollbar.set)
scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
msg_list.pack(side=tkinter.LEFT, fill=tkinter.BOTH)
msg_list.pack()
messages_frame.pack()

entry_field = tkinter.Entry(top, textvariable=my_msg)
entry_field.bind("<Return>", send)
entry_field.pack()
send_button = tkinter.Button(top, text="Send", command=send)
send_button.pack()

top.protocol("WM_DELETE_WINDOW", on_closing)

#----Now comes the sockets part----
HOST = input('Enter host: ')
```

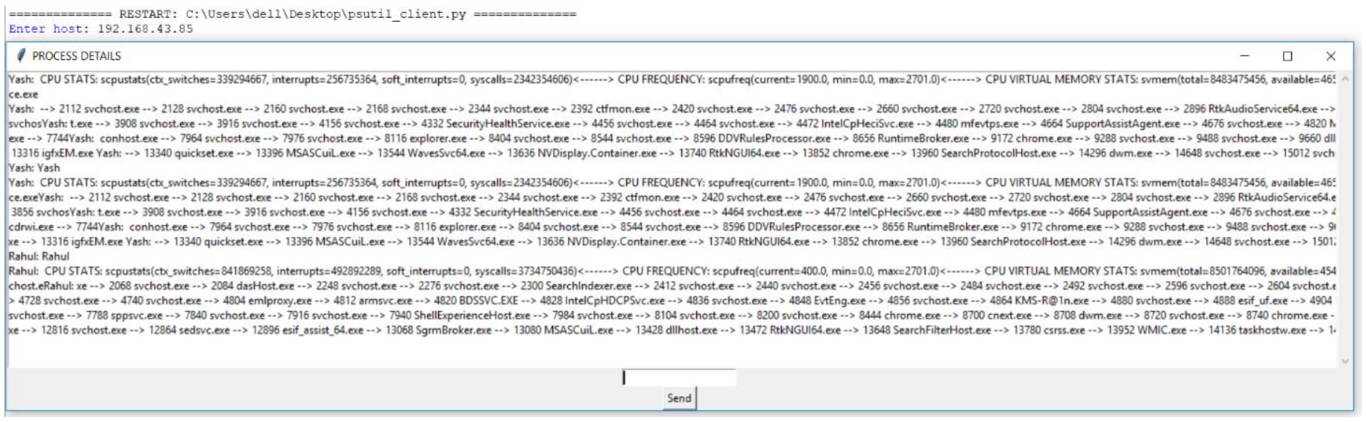
```
PORT = input('Enter port: ')
if not PORT:
    PORT = 33000
else:
    PORT = int(PORT)

BUFSIZ = 1024
ADDR = (HOST, PORT)

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)

receive_thread = Thread(target=receive)
receive_thread.start()
tkinter.mainloop()  # Starts GUI execution.
```

3. Displaying process and cpu details of all clients connected to a server –



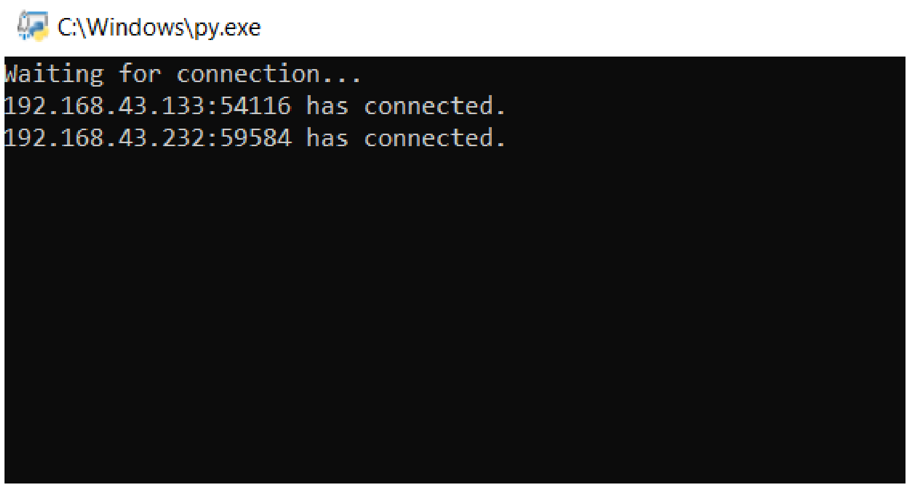
```
============= RESTART: G:\SEM_4\OS\J_Component\psutil_client.py =============
Enter host: 192.168.43.85
```

```
Rahul:  CPU STATS: scpustats(ctx_switches=841869258, interrupts=492892289, soft_interrupts=0, syscalls=3734750436)<------> CPU FREQUENCY: scpufreq(current=400.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=85017640
chost.e
Rahul: xe --> 2068 svchost.exe --> 2084 dasHost.exe --> 2248 svchost.exe --> 2276 svchost.exe --> 2300 SearchIndexer.exe --> 2412 svchost.exe --> 2440 svchost.exe --> 2456 svchost.exe --> 2484 svchost.exe --> 2492 svchost.exe --> 2596 svchost.exe --> 260
svchost.exe --> 4740 svchost.exe --> 4804 emlproxy.exe --> 4812 armsvc.exe --> 4820 BDSSVC.EXE --> 4828 IntelCpHDCPSvc.exe --> 4836 svchost.exe --> 4848 EvtEng.exe --> 4856 svchost.exe --> 4864 KMS-R@1n.exe --> 4880 svchost.exe --> 4888 esif_uf.exe
.exe --> 7788 sppsvc.exe --> 7840 svchost.exe --> 7916 svchost.exe --> 7940 ShellExperienceHost.exe --> 7984 svchost.exe --> 8104 svchost.exe --> 8200 svchost.exe --> 8444 chrome.exe --> 8700 cnext.exe --> 8708 dwm.exe --> 8720 svchost.exe --> 8740 chr
12816 svchost.exe --> 12864 sedsvc.exe --> 12896 esif_assist_64.exe --> 13068 SgrmBroker.exe --> 13080 MSASCuiL.exe --> 13428 dllhost.exe --> 13472 RtkNGUI64.exe --> 13648 SearchFilterHost.exe --> 13780 csrss.exe --> 13952 WMIC.exe --> 14136 taskhostw
PU FREQUENCY: scpufreq(current=1900.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=8483475456, available=4651237376, percent=45.2, used=3832238080, free=4651237376)<------>System Idle Process --> 0 System --> 4 R
392 ctfmon.exe --> 2420 svchost.exe --> 2476 svchost.exe --> 2660 svchost.exe --> 2720 svchost.exe --> 2804 svchost.exe --> 2896 RtkAudioService64.exe --> 2904 svchost.exe --> 2980 svchost.exe --> 3024 svchost.exe --> 3028 svchost.exe --> 3040 svchost.e
-> 4456 svchost.exe --> 4464 svchost.exe --> 4472 IntelCpHeciSvc.exe --> 4480 mfevtps.exe --> 4664 SupportAssistAgent.exe --> 4676 svchost.exe --> 4820 MMSSHOST.exe --> 4856 mfevtps.exe --> 4980 csrss.exe --> 4988 svchost.exe --> 4996 ProtectedMod
-> 8544 svchost.exe --> 8596 DDVRulesProcessor.exe --> 8656 RuntimeBroker.exe --> 9172 chrome.exe --> 9288 svchost.exe --> 9488 svchost.exe --> 9660 dllhost.exe --> 9728 SgrmBroker.exe --> 9736 svchost.exe --> 9788 svchost.exe --> 9884 svchost.exe -->
ntainer.exe --> 13740 RtkNGUI64.exe --> 13852 chrome.exe --> 13960 SearchProtocolHost.exe --> 14296 dwm.exe --> 14648 svchost.exe --> 15012 svchost.exe --> 15104 fontdrvhost.exe --> 15220 smartscreen.exe --> 15308 esif_assist_64.exe --> 15328 chrome
Yash: Yash
Yash:  CPU STATS: scpustats(ctx_switches=339294667, interrupts=256735364, soft_interrupts=0, syscalls=2342354606)<------> CPU FREQUENCY: scpufreq(current=1900.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=8483475
ce.exeYash:  --> 2112 svchost.exe --> 2128 svchost.exe --> 2160 svchost.exe --> 2168 svchost.exe --> 2344 svchost.exe --> 2392 ctfmon.exe --> 2420 svchost.exe --> 2476 svchost.exe --> 2660 svchost.exe --> 2720 svchost.exe --> 2804 svchost.exe --> 2896 Rtk
3856 svchosYash: t.exe --> 3908 svchost.exe --> 3916 svchost.exe --> 4156 svchost.exe --> 4332 SecurityHealthService.exe --> 4456 svchost.exe --> 4464 svchost.exe --> 4472 IntelCpHeciSvc.exe --> 4480 mfevtps.exe --> 4664 SupportAssistAgent.exe --> 4676
cdrwi.exe --> 7744Yash:  conhost.exe --> 7964 svchost.exe --> 7976 svchost.exe --> 8116 explorer.exe --> 8404 svchost.exe --> 8544 svchost.exe --> 8596 DDVRulesProcessor.exe --> 8656 RuntimeBroker.exe --> 9172 chrome.exe --> 9288 svchost.exe --> 9488
xe --> 13316 igfxEM.exe Yash: --> 13340 quickset.exe --> 13396 MSASCuiL.exe --> 13544 WavesSvc64.exe --> 13636 NVDisplay.Container.exe --> 13740 RtkNGUI64.exe --> 13852 chrome.exe --> 13960 SearchProtocolHost.exe --> 14296 dwm.exe --> 14648 svcl
Rahul: Rahul
Rahul:  CPU STATS: scpustats(ctx_switches=841869258, interrupts=492892289, soft_interrupts=0, syscalls=3734750436)<------> CPU FREQUENCY: scpufreq(current=400.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=85017640
chost.eRahul: xe --> 2068 svchost.exe --> 2084 dasHost.exe --> 2248 svchost.exe --> 2276 svchost.exe --> 2300 SearchIndexer.exe --> 2412 svchost.exe --> 2440 svchost.exe --> 2456 svchost.exe --> 2484 svchost.exe --> 2492 svchost.exe --> 2596 svchost.exe -
```



```
============== RESTART: C:\Users\dell\Desktop\psutil_client.py ==============
Enter host: 192.168.43.85
```

```
Yash:  CPU STATS: scpustats(ctx_switches=339294667, interrupts=256735364, soft_interrupts=0, syscalls=2342354606)<------> CPU FREQUENCY: scpufreq(current=1900.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=8483475456, available=46
ce.exe
Yash:  --> 2112 svchost.exe --> 2128 svchost.exe --> 2160 svchost.exe --> 2168 svchost.exe --> 2344 svchost.exe --> 2392 ctfmon.exe --> 2420 svchost.exe --> 2476 svchost.exe --> 2660 svchost.exe --> 2720 svchost.exe --> 2804 svchost.exe --> 2896 RtkAudioService64.exe -->
svchosYash: t.exe --> 3908 svchost.exe --> 3916 svchost.exe --> 4156 svchost.exe --> 4332 SecurityHealthService.exe --> 4456 svchost.exe --> 4464 svchost.exe --> 4472 IntelCpHeciSvc.exe --> 4480 mfevtps.exe --> 4664 SupportAssistAgent.exe --> 4676 svchost.exe --> 4820 M
exe --> 7744Yash:  conhost.exe --> 7964 svchost.exe --> 7976 svchost.exe --> 8116 explorer.exe --> 8404 svchost.exe --> 8544 svchost.exe --> 8596 DDVRulesProcessor.exe --> 8656 RuntimeBroker.exe --> 9172 chrome.exe --> 9288 svchost.exe --> 9488 svchost.exe --> 9660 dll
 13316 igfxEM.exe Yash: --> 13340 quickset.exe --> 13396 MSASCuiL.exe --> 13544 WavesSvc64.exe --> 13636 NVDisplay.Container.exe --> 13740 RtkNGUI64.exe --> 13852 chrome.exe --> 13960 SearchProtocolHost.exe --> 14296 dwm.exe --> 14648 svchost.exe --> 15012 svch
Yash: Yash
Yash:  CPU STATS: scpustats(ctx_switches=339294667, interrupts=256735364, soft_interrupts=0, syscalls=2342354606)<------> CPU FREQUENCY: scpufreq(current=1900.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=8483475456, available=46
ce.exeYash:  --> 2112 svchost.exe --> 2128 svchost.exe --> 2160 svchost.exe --> 2168 svchost.exe --> 2344 svchost.exe --> 2392 ctfmon.exe --> 2420 svchost.exe --> 2476 svchost.exe --> 2660 svchost.exe --> 2720 svchost.exe --> 2804 svchost.exe --> 2896 RtkAudioService64.e
 3856 svchosYash: t.exe --> 3908 svchost.exe --> 3916 svchost.exe --> 4156 svchost.exe --> 4332 SecurityHealthService.exe --> 4456 svchost.exe --> 4464 svchost.exe --> 4472 IntelCpHeciSvc.exe --> 4480 mfevtps.exe --> 4664 SupportAssistAgent.exe --> 4676 svchost.exe --> 4
cdrwi.exe --> 7744Yash:  conhost.exe --> 7964 svchost.exe --> 7976 svchost.exe --> 8116 explorer.exe --> 8404 svchost.exe --> 8544 svchost.exe --> 8596 DDVRulesProcessor.exe --> 8656 RuntimeBroker.exe --> 9172 chrome.exe --> 9288 svchost.exe --> 9488 svchost.exe --> 9
xe --> 13316 igfxEM.exe Yash: --> 13340 quickset.exe --> 13396 MSASCuiL.exe --> 13544 WavesSvc64.exe --> 13636 NVDisplay.Container.exe --> 13740 RtkNGUI64.exe --> 13852 chrome.exe --> 13960 SearchProtocolHost.exe --> 14296 dwm.exe --> 14648 svchost.exe --> 1501
Rahul: Rahul
Rahul:  CPU STATS: scpustats(ctx_switches=841869258, interrupts=492892289, soft_interrupts=0, syscalls=3734750436)<------> CPU FREQUENCY: scpufreq(current=400.0, min=0.0, max=2701.0)<------> CPU VIRTUAL MEMORY STATS: svmem(total=8501764096, available=454
chost.eRahul: xe --> 2068 svchost.exe --> 2084 dasHost.exe --> 2248 svchost.exe --> 2276 svchost.exe --> 2300 SearchIndexer.exe --> 2412 svchost.exe --> 2440 svchost.exe --> 2456 svchost.exe --> 2484 svchost.exe --> 2492 svchost.exe --> 2596 svchost.exe --> 2604 svchost.e
> 4728 svchost.exe --> 4740 svchost.exe --> 4804 emlproxy.exe --> 4812 armsvc.exe --> 4820 BDSSVC.EXE --> 4828 IntelCpHDCPSvc.exe --> 4836 svchost.exe --> 4848 EvtEng.exe --> 4856 svchost.exe --> 4864 KMS-R@1n.exe --> 4880 svchost.exe --> 4888 esif_uf.exe --> 4904
svchost.exe --> 7788 sppsvc.exe --> 7840 svchost.exe --> 7916 svchost.exe --> 7940 ShellExperienceHost.exe --> 7984 svchost.exe --> 8104 svchost.exe --> 8200 svchost.exe --> 8444 chrome.exe --> 8700 cnext.exe --> 8708 dwm.exe --> 8720 svchost.exe --> 8740 chrome.exe -
xe --> 12816 svchost.exe --> 12864 sedsvc.exe --> 12896 esif_assist_64.exe --> 13068 SgrmBroker.exe --> 13080 MSASCuiL.exe --> 13428 dllhost.exe --> 13472 RtkNGUI64.exe --> 13648 SearchFilterHost.exe --> 13780 csrss.exe --> 13952 WMIC.exe --> 14136 taskhostw.exe --> 1
```



```
Waiting for connection...
192.168.43.133:54116 has connected.
192.168.43.232:59584 has connected.
```

Psutil_server –

```
#!/usr/bin/env python3
"""Server for multithreaded (asynchronous) chat application."""
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread


def accept_incoming_connections():
    """Sets up handling for incoming clients."""
    while True:
        client, client_address = SERVER.accept()
        print("%s:%s has connected." % client_address)
        addresses[client] = client_address
        Thread(target=handle_client, args=(client,)).start()


def handle_client(client):  # Takes client socket as argument.
    """Handles a single client connection."""
    name = client.recv(BUFSIZ).decode("utf8")
    msg = "%s has joined the chat!" % name
    broadcast(bytes(msg, "utf8"))
    clients[client] = name
    while True:
        msg = client.recv(BUFSIZ)
        if msg != bytes("{quit}", "utf8"):
            broadcast(msg, name+": ")

        else:
            client.send(bytes("{quit}", "utf8"))
            client.close()
            del clients[client]
            broadcast(bytes("%s has left the chat." % name, "utf8"))
            break



def broadcast(msg, prefix=""):  # prefix is for name identification.
    """Broadcasts a message to all the clients."""
    for sock in clients:
        sock.send(bytes(prefix, "utf8")+msg)



clients = {}
addresses = {}

HOST = ''
PORT = 33000
BUFSIZ = 1024
ADDR = (HOST, PORT)
SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)
if __name__ == "__main__":
    SERVER.listen(5)
    print("Waiting for connection...")
    ACCEPT_THREAD = Thread(target=accept_incoming_connections)
    ACCEPT_THREAD.start()
    ACCEPT_THREAD.join()
```

Psutil_client –

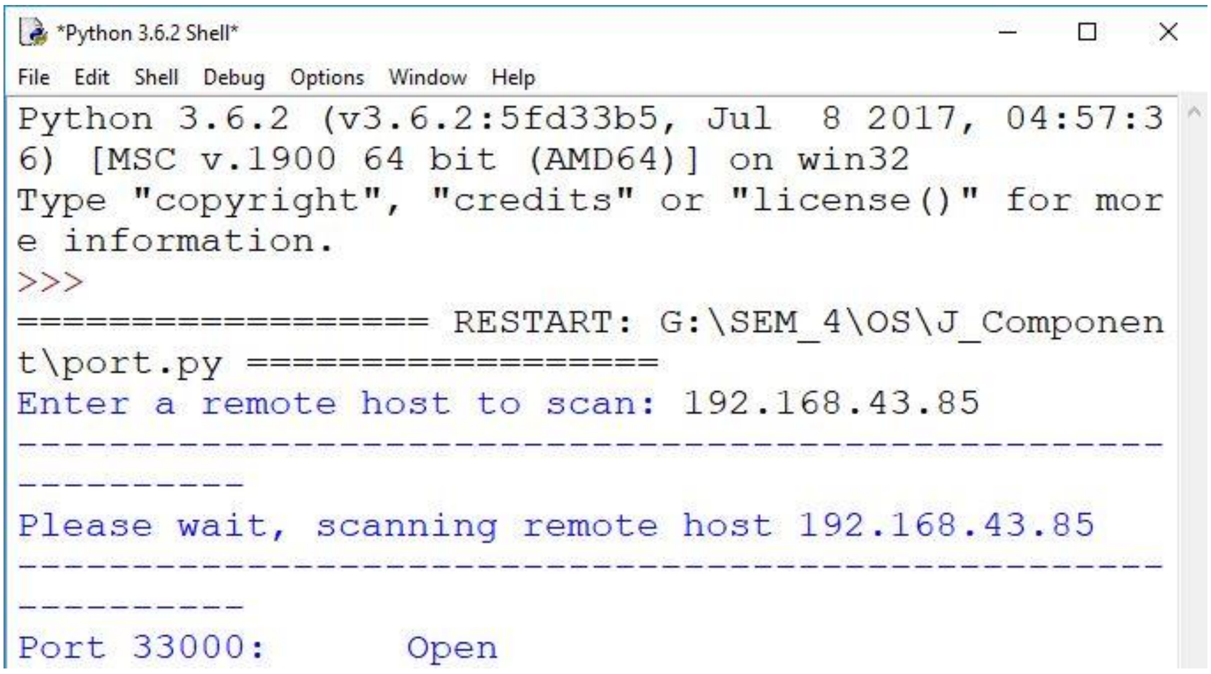
```
"""psutil (python system and process utilities) is a cross-platform library for retrieving
information on running processes and system utilization (CPU, memory, disks, network, sensors)
```

```
in Python."""
#!/usr/bin/env python3
"""Script for Tkinter GUI chat client."""
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
import tkinter
import psutil




def getListOfProcessSortedByMemory():
    '''
    Get list of running process sorted by Memory Usage
    '''
    listOfProcObjects = []
    # Iterate over the list
    for proc in psutil.process_iter():
       try:
           # Fetch process details as dict
           pinfo = proc.as_dict(attrs=['pid', 'name', 'username'])
           pinfo['vms'] = proc.memory_info().vms / (1024 * 1024)
           # Append dict to list
           listOfProcObjects.append(pinfo);
```

```
        except (psutil.NoSuchProcess, psutil.AccessDenied,
psutil.ZombieProcess):
           pass

    # Sort list of dict by key vms i.e. memory usage
    listOfProcObjects = sorted(listOfProcObjects, key=lambda
procObj: procObj['vms'], reverse=True)

    return listOfProcObjects



processName=""
processName+=" CPU STATS: "+str(psutil.cpu_stats())+"<------>"
processName+=" CPU FREQUENCY:
"+str(psutil.cpu_freq(percpu=False))+"<------>"
processName+=" CPU VIRTUAL MEMORY STATS:
"+str(psutil.virtual_memory())+"<------>"

for proc in psutil.process_iter():
    processName += str(proc.name())+" --> "+str(proc.pid)+" "




def receive():
    """Handles receiving of messages."""
    while True:
        try:
            msg = client_socket.recv(BUFSIZ).decode("utf8")
            msg_list.insert(tkinter.END, msg)
        except OSError:  # Possibly client has left the chat.
            break


def send(event=None):  # event is passed by binders.
    """Handles sending of messages."""
    msg = my_msg.get()
    my_msg.set("")  # Clears input field.
    client_socket.send(bytes(msg, "utf8"))
    msg = str(processName)
    # Clears input field.
    client_socket.send(bytes(msg, "utf8"))
    if msg == "{quit}":
        client_socket.close()
        top.quit()


def on_closing(event=None):
    """This function is to be called when the window is closed."""
    my_msg.set("{quit}")
    send()

top = tkinter.Tk()
top.title("PROCESS DETAILS")

messages_frame = tkinter.Frame(top)
my_msg = tkinter.StringVar()  # For the messages to be sent.
my_msg.set("Type your name here.")
scrollbar = tkinter.Scrollbar(messages_frame)  # To navigate
through past messages.
# Following will contain the messages.
msg_list = tkinter.Listbox(messages_frame, height=20, width=240,
yscrollcommand=scrollbar.set, xscrollcommand=scrollbar.set)
scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
msg_list.pack(side=tkinter.LEFT, fill=tkinter.BOTH)
msg_list.pack()
messages_frame.pack()

entry_field = tkinter.Entry(top, textvariable=my_msg)
entry_field.bind("<Return>", send)
entry_field.pack()
send_button = tkinter.Button(top, text="Send", command=send)
send_button.pack()
```

```
top.protocol("WM_DELETE_WINDOW", on_closing)

#----Now comes the sockets part----
HOST = input('Enter host: ')
PORT = 33000



BUFSIZ = 1024
ADDR = (HOST, PORT)

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)

receive_thread = Thread(target=receive)
receive_thread.start()
tkinter.mainloop()  # Starts GUI execution.
```

4. Sensing the connected ports –



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:3
6) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for mor
e information.
>>>
================== RESTART: G:\SEM_4\OS\J_Componen
t\port.py ==================
Enter a remote host to scan: 192.168.43.85
------------------------------------------------------
----------
Please wait, scanning remote host 192.168.43.85
------------------------------------------------------
----------
Port 33000:      Open
```

Port.py –

```
#!/usr/bin/env python
import socket
import subprocess
import sys
from datetime import datetime

# Clear the screen
subprocess.call('clear', shell=True)

# Ask for input
remoteServer    = input("Enter a remote host to scan: ")
remoteServerIP  = socket.gethostbyname(remoteServer)
```

```
# Print a nice banner with information on which host we are about to
scan

print ("-" * 60)

print ("Please wait, scanning remote host", remoteServerIP)

print ("-" * 60)


# Check what time the scan started

t1 = datetime.now()


# Using the range function to specify ports (here it will scans all
ports between 1 and 1024)


# We also put in some error handling for catching errors


try:

    for port in range(33000,33120):

        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        result = sock.connect_ex((remoteServerIP, port))

        if result == 0:

            print ("Port {}:         Open".format(port))

        sock.close()


except KeyboardInterrupt:

    print ("You pressed Ctrl+C")
    sys.exit()

except socket.gaierror:
    print ('Hostname could not be resolved. Exiting')
    sys.exit()

except socket.error:
    print ("Couldn't connect to server")
    sys.exit()

# Checking the time again
t2 = datetime.now()

# Calculates the difference of time, to see how long it took to run
the script
total =  t2 - t1

# Printing the information to screen
print ('Scanning Completed in: ', total)
```

Conclusion and future works –

In the final analysis we successfully established a inter process communication between server machine and client machines to chat as well as to share process and CPU details in an easy and compatible GUI format. For this we have use one important OS library of python named psutil to achieve our goal. Also we have used tkinter GUI for communication purpose. In future we are looking forward to add a feature that will share the log file stored in the pc that contain the CPU information.