SRM INSTITUTE OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# 18CSC302J - COMPUTER NETWORKS

SEMESTER – 5

BATCH – 1

| | |
|---|---|
| **REGISTRATION NUMBER** | **RA1811028010049** |
| **NAME** | **SHUSHRUT KUMAR** |

**B.tech-CSE-CC, Third Year (Section: J2)**

**Faculty Incharge: Vaishnavi Moorthy, Asst Prof/Dept of CSE**

**Year :- 2020-2021**

# INDEX

**Exercise: #1**

**Date : 05 August 2020**

## Study of Basic Functions of Socket Programming

**Aim:** Study of necessary header files with respect to socket programming and of Basic Functions of Socket Programming. Standard Library for Socket IO

**Technical Objective:** There are a lot of libraries in the socket IO so figure out what they do using the command line interface into linux based OS

**Given Requirements:** Using your linux based OS find out what different libraries are present in the standard library for socket IO

**1.man string** : The string header defines one variable type, one macro, and various functions for manipulating arrays of characters.

**2.man netdb** : This manual page is part of the POSIX Programmer's Manual. The Linux Implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

**3.man time** : The time.h header defines four variable types, two macro and various functions for manipulating date and time. **time**() returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).If *tloc* is non-NULL, the return value is also stored in the memory pointed to by *tloc*.

**4.man stdio** : The stdio header defines three variable types, several macros, and various functions for performing input and output.

**5.man ioctl** : The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument fd must be an open file descriptor.

**6.man stat** : The <sys/stat.h> header defines the structure of the data returned by the functions fstat(), lstat(), and stat().

**7.man sys** : The **sysfs** filesystem is a pseudo-filesystem which provides an interface to kernel data structures. (More precisely, the files and directories in **sysfs** provide a view of the *kobject* structures defined internally within the kernel.) The files under **sysfs** provide information about devices, kernel modules, filesystems, and other kernel components.

**8.man ioctl** : The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument fd must be an open file descriptor.

**9.man errno** : The *<errno.h>* header file defines the integer variable *errno*, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

**10.man pcap-filter** : The Packet Capture library provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism. It also supports saving captured packets to a ``save file'', and reading packets from a ``save file''.

**11.man write : write()** writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the

**12.man inet : inet_aton()** converts the Internet host address *cp* from the IPv4 numbers-and-dots notation into binary form (in network byte order) and stores it in the structure that *inp* points to. **inet_aton**() returns nonzero if the address is valid, zero if not.

**13.man system** : The **sysfs** filesystem is a pseudo-filesystem which provides an interface to kernel data structures. (More precisely, the files and directories in **sysfs** provide a view of the *kobject* structures defined internally within the kernel.) The files under **sysfs** provide information about devices, kernel modules, filesystems, and other kernel components.

**14.man gethostname** : These system calls are used to access or to change the system hostname. More precisely, they operate on the hostname associated with the calling process's UTS namespace.

**15.man read** : read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.

**16.man htonl** : These functions shall convert 16-bit and 32-bit quantities between network byte order and host byte order.On some implementations, these functions are defined as macros.The uint32_t and uint16_t types are defined in <*inttypes.h*>.

**17.man htons** : The htonl() function converts the unsigned integer *hostlong* from host byte order to network byte order. The htons() function converts the unsigned short integer *hostshort* from host byte order to network byte order. The ntohl() function converts the unsigned integer *netlong* from network byte order to host byte order.

**18.man gethostbyname** : The gethostbyname() function returns a structure of type *hostent* for the given host *name*. Here *name* is either a hostname or an IPv4 address in standard dot notation (as for inet_addr(3)). If *name* is an IPv4 address, no lookup is performed and gethostbyname() simply copies *name* into the *h_name* field and its *struct in_addr* equivalent
into the *h_addr_list[0]* field of the returned *hostent* structure. If *name* doesn't end in a dot and the environment variable HOSTALIASES is set, the alias file pointed to by HOSTALIASES will first be searched for *name* (see hostname(7) for the file format). The current domain and its parents are searched unless the name ends in a dot.

**19.man socket** : socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

**20.man bind** : When a socket is created with a socket, it exists in a namespace (address family) but has no address assigned to it. bind() assigns the address specified by *addr* to the socket referred to by the file descriptor *sockfd*. *addrlen* specifies the size, in bytes, of the address structure pointed to by *addr*. Traditionally, this operation is called "assigning a name to a socket".

**21.man if_config** : Ifconfig is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.\

**22.man accept** : The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, *sockfd*, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket *sockfd* is unaffected by this call.

**23.man recv** : The recv(), recvfrom(), and recvmsg() calls are used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls.

**RESULT:** Various Functions related to socket programming were studied. Those are the part of Socket IO and Basic Functions of Socket Programming.

**Exercise: #2**

**Date : 12 August 2020**

## TCP/IP Client Server Communication

**Aim:** Setting up of simple TCP/IP Client Server Communication. Where the TCP connection will help the server and client to communicate to each other.

**Technical Objective:** The TCP/IP protocol allows systems to communicate even if they use different types of network hardware. ... IP, or Internet Protocol, performs the actual data transfer between different systems on the network or Internet. Using TCP binding, you can create both client and server portions of client-server systems. Demonstrate the same.

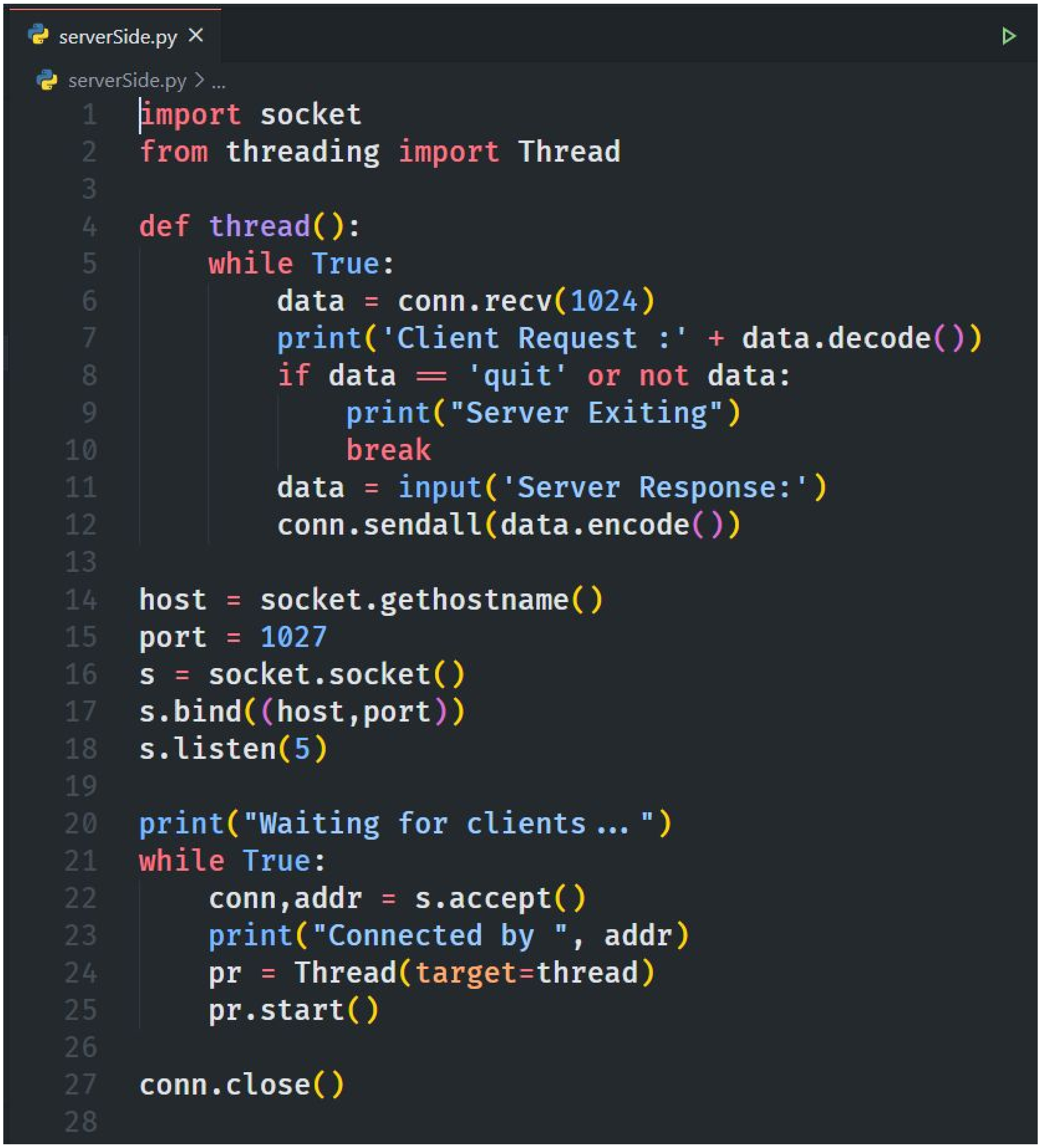
**Given Requirements:** TCP/IP client server communication helps in connecting two devices as client and server, so make a TCP connection to demonstrate the same.

**TCP SERVER :**

**Algorithm -**

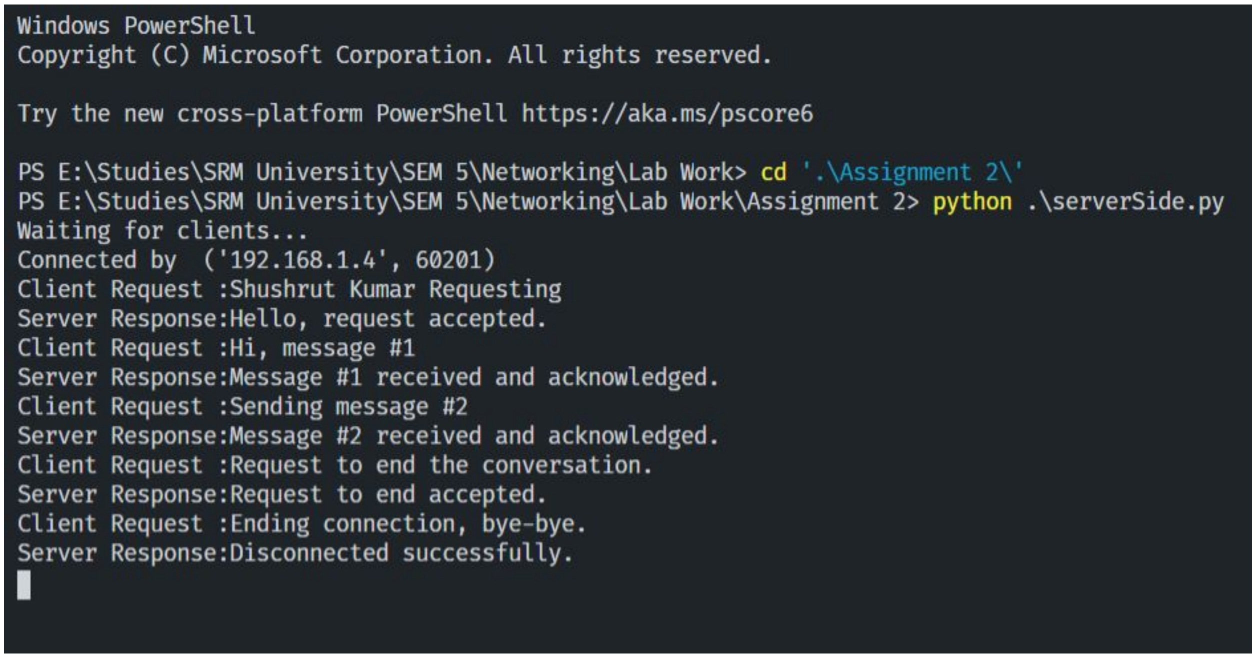
1. Create() - Create TCP socket.
2. Bind() - Bind the socket to the server address.
3. Listen() - put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. Accept() - At this point, connection is established between client and server, and they are ready to transfer data.
5. Go back to Step 3.

**Code-**



```python
import socket
from threading import Thread

def thread():
    while True:
        data = conn.recv(1024)
        print('Client Request :' + data.decode())
        if data == 'quit' or not data:
            print("Server Exiting")
            break
        data = input('Server Response:')
        conn.sendall(data.encode())

host = socket.gethostname()
port = 1027
s = socket.socket()
s.bind((host,port))
s.listen(5)

print("Waiting for clients ... ")
while True:
    conn,addr = s.accept()
    print("Connected by ", addr)
    pr = Thread(target=thread)
    pr.start()

conn.close()
```

**Output-**

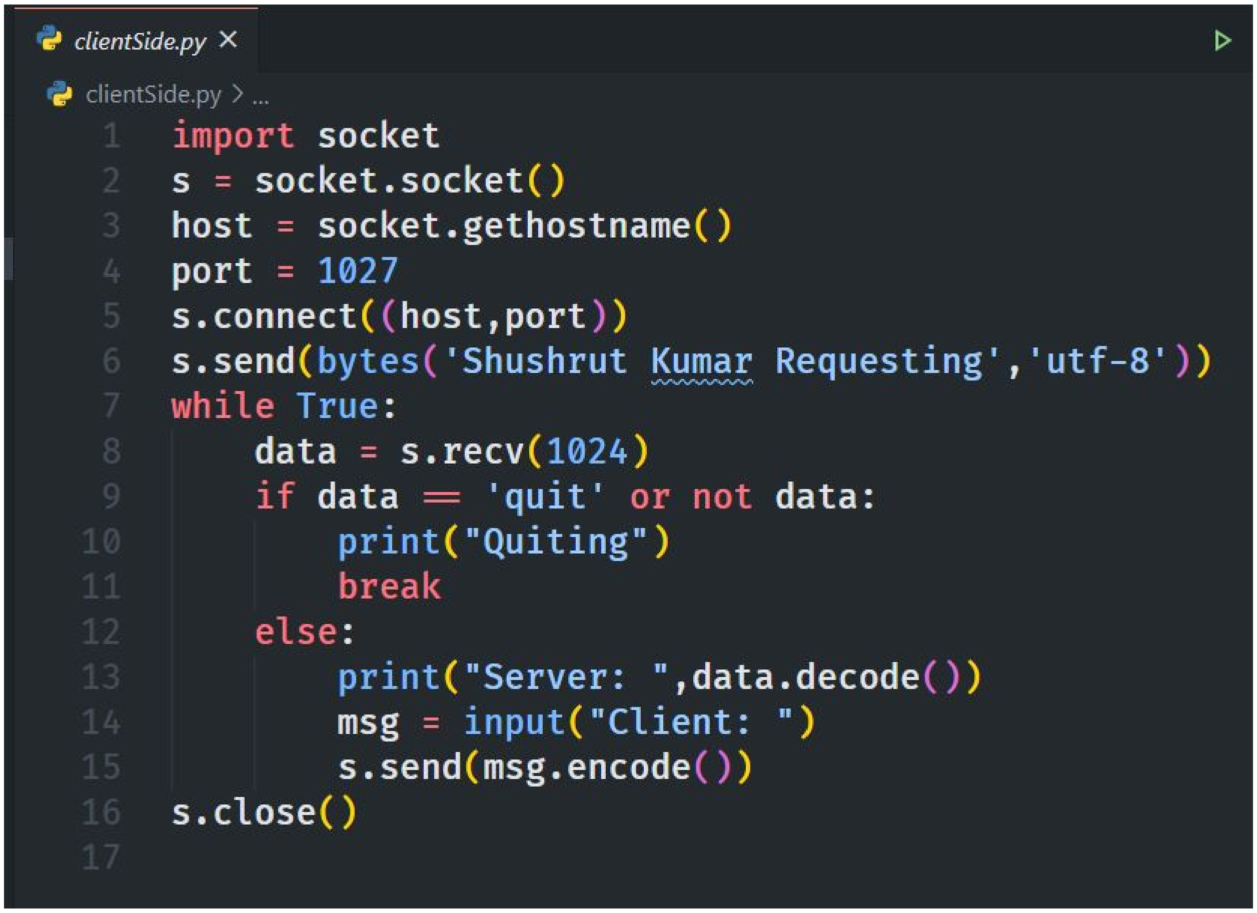


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\Studies\SRM University\SEM 5\Networking\Lab Work> cd '.\Assignment 2\'
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 2> python .\serverSide.py
Waiting for clients...
Connected by  ('192.168.1.4', 60201)
Client Request :Shushrut Kumar Requesting
Server Response:Hello, request accepted.
Client Request :Hi, message #1
Server Response:Message #1 received and acknowledged.
Client Request :Sending message #2
Server Response:Message #2 received and acknowledged.
Client Request :Request to end the conversation.
Server Response:Request to end accepted.
Client Request :Ending connection, bye-bye.
Server Response:Disconnected successfully.
```

**Algorithm Client :**

1. Create TCP socket.
2. Bind the socket to server address.
3. Put the server socket in a passive mode, where it waits for the client to approach the
4. server to make a connection
5. At this point, connection is established between client and server, and they are ready to
6. transfer data.
7. Go back to Step 3.

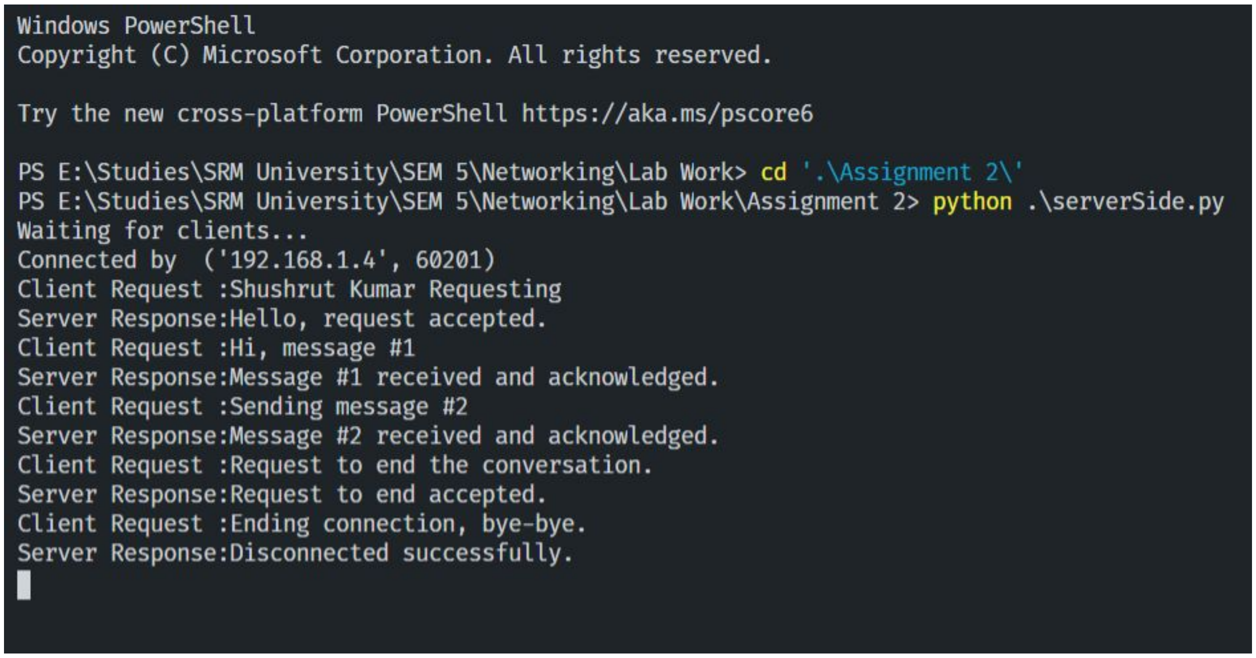
**Code-**



clientSide.py

```
import socket
s = socket.socket()
host = socket.gethostname()
port = 1027
s.connect((host,port))
s.send(bytes('Shushrut Kumar Requesting','utf-8'))
while True:
    data = s.recv(1024)
    if data == 'quit' or not data:
        print("Quiting")
        break
    else:
        print("Server: ",data.decode())
        msg = input("Client: ")
        s.send(msg.encode())
s.close()
```

**Output-**



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\Studies\SRM University\SEM 5\Networking\Lab Work> cd '.\Assignment 2\'
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 2> python .\serverSide.py
Waiting for clients...
Connected by  ('192.168.1.4', 60201)
Client Request :Shushrut Kumar Requesting
Server Response:Hello, request accepted.
Client Request :Hi, message #1
Server Response:Message #1 received and acknowledged.
Client Request :Sending message #2
Server Response:Message #2 received and acknowledged.
Client Request :Request to end the conversation.
Server Response:Request to end accepted.
Client Request :Ending connection, bye-bye.
Server Response:Disconnected successfully.
█
```

**Result :** A simple TCP/IP Server/Client Communication was set-up. Where there was connection and communication made between server and client.

**Exercise: #3**

**Date: 20 August 2020**

## UDP Server Client Implementation

**Aim:** To set up a UDP Echo Client Server Communication. In computer networking, the User Datagram Protocol is one of the core members of the Internet protocol suite.

**Technical Objective:** The User Datagram Protocol, or UDP, is a communication protocol used across the Internet for especially time-sensitive transmissions such as video playback or DNS lookups. It speeds up communications by not formally establishing a connection before data is transferred. Demonstrate the same using server client connection.

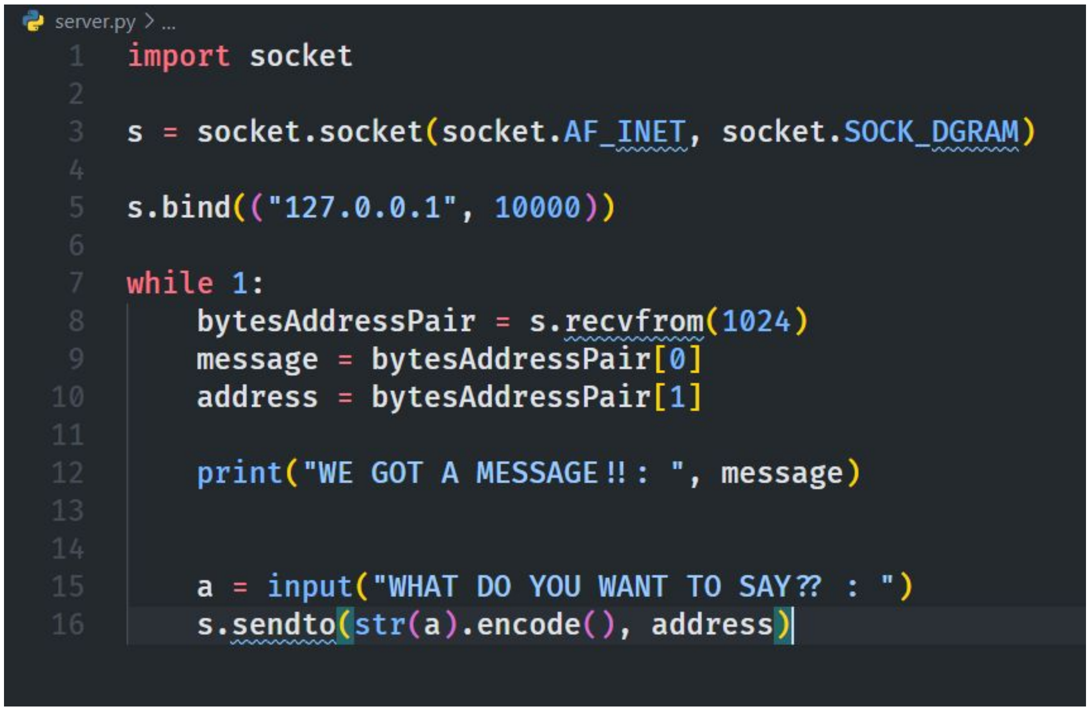
**Given Requirements:** User Datagram Protocol (UDP) is a Transport Layer protocol. UDP is a part of the Internet Protocol suite, referred to as the UDP/IP suite. Unlike TCP, it is unreliable and connectionless protocol. So, there is no need to establish connection prior to data transfer.

**SERVER-**

ALGORITHM:

1. Create UDP socket.
2. Bind the sockets to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet and send a reply to the client.
5. Go back to step 3.

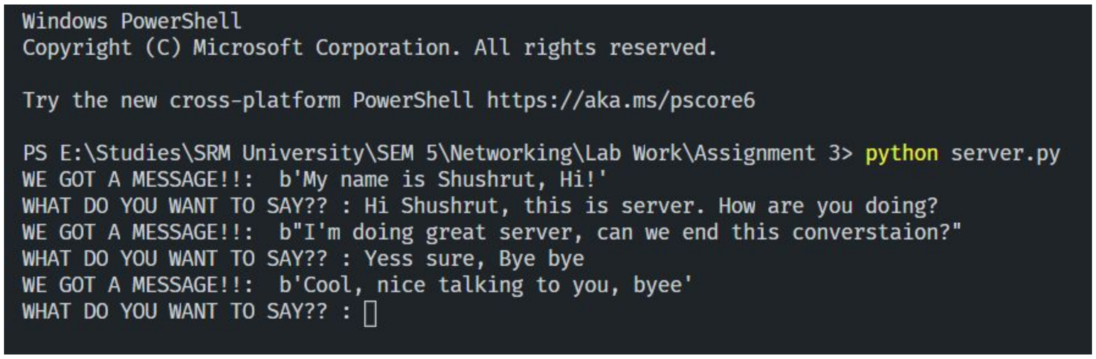
CODE



server.py > ...

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

s.bind(("127.0.0.1", 10000))

while 1:
    bytesAddressPair = s.recvfrom(1024)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]

    print("WE GOT A MESSAGE!!: ", message)


    a = input("WHAT DO YOU WANT TO SAY?? : ")
    s.sendto(str(a).encode(), address)
```

OUTPUT:

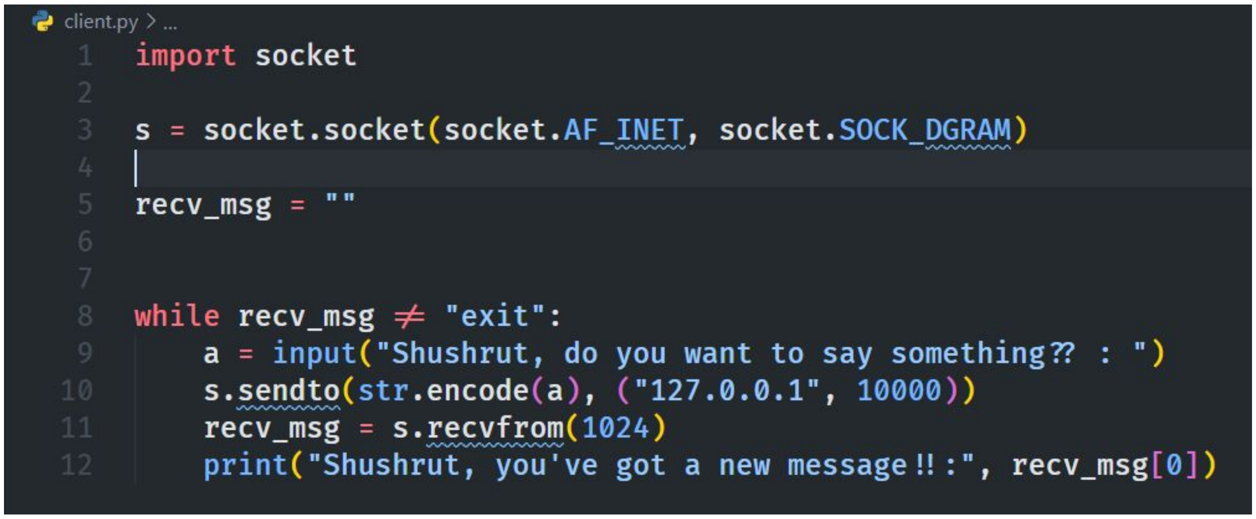


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 3> python server.py
WE GOT A MESSAGE!!:  b'My name is Shushrut, Hi!'
WHAT DO YOU WANT TO SAY?? : Hi Shushrut, this is server. How are you doing?
WE GOT A MESSAGE!!:  b"I'm doing great server, can we end this converstaion?"
WHAT DO YOU WANT TO SAY?? : Yess sure, Bye bye
WE GOT A MESSAGE!!:  b'Cool, nice talking to you, byee'
WHAT DO YOU WANT TO SAY?? : 
```

**Algorithm Client :**

1. Create UDP socket.
2. Bind the socket to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet and send a reply to the client.
5. Go back to Step 3.

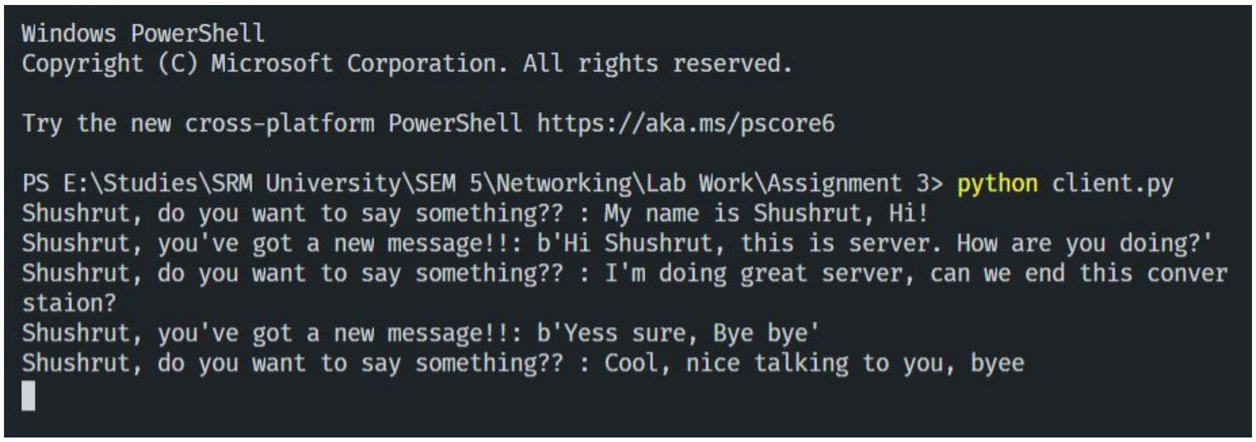
CODE



client.py > ...

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

recv_msg = ""


while recv_msg ≠ "exit":
    a = input("Shushrut, do you want to say something⁇ : ")
    s.sendto(str.encode(a), ("127.0.0.1", 10000))
    recv_msg = s.recvfrom(1024)
    print("Shushrut, you've got a new message‼:", recv_msg[0])
```

OUTPUT:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 3> python client.py
Shushrut, do you want to say something?? : My name is Shushrut, Hi!
Shushrut, you've got a new message!!: b'Hi Shushrut, this is server. How are you doing?'
Shushrut, do you want to say something?? : I'm doing great server, can we end this conver
staion?
Shushrut, you've got a new message!!: b'Yess sure, Bye bye'
Shushrut, do you want to say something?? : Cool, nice talking to you, byee
█
```

**Result :** A UDP Echo Client/Server Communication was established. And the connectionless setup is made between client and server to communicate with each other.

**Exercise: #4**

**Date: 28 August 2020**

## Date and time request - TCP/IP communication

**Aim:** To set up a TCP/IP Client - Server Communication and request for date and time from the server. To get the accurate time and date from the server to the client.

**Technical Objective:** The creators of TCP/IP recognized that certain applications might not work properly if there was too much differential between the system clocks of a pair of devices. To support this requirement, they created a pair of ICMP messages that allow devices to exchange system time information. The initiating device creates a Timestamp message and sends it to the device with which it wishes to synchronize.

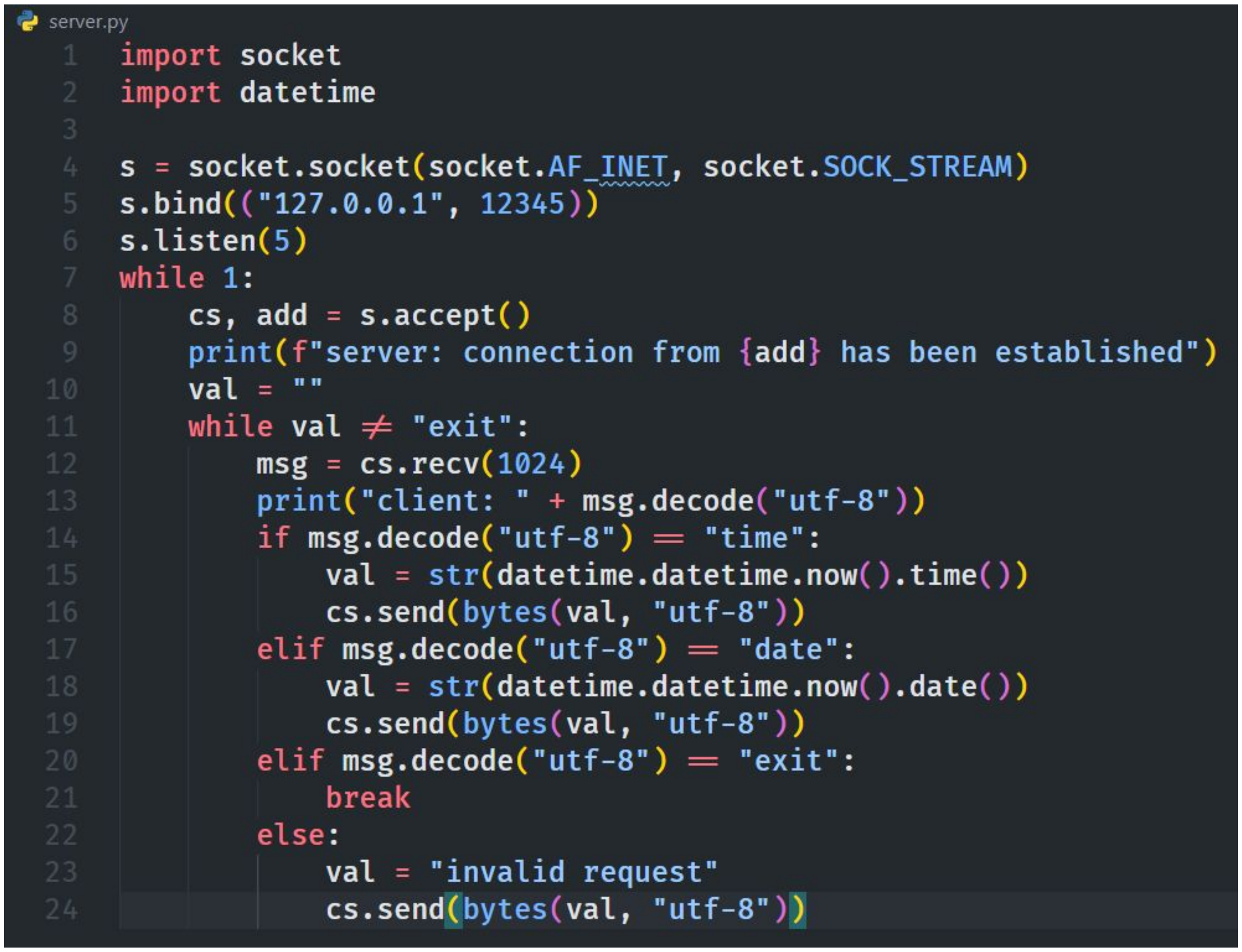
**Given Requirements:** One aspect of this autonomy is that each device maintains a separate system clock. Since even highly-accurate clocks have slight differences in how accurately they keep time, as well as the time they are initialized with, this means that under normal circumstances, no two devices on an internetwork are guaranteed to have exactly the same time.

**Algorithm:**

1. Create a TCP server socket.
2. Bind the socket to a server address.
3. Start a client socket while the server socket is passive and waiting for the client to connect.
4. Once connection is established between the client and server socket, send a request for the date or time to the server.
5. Interpret the message from the client and send the required output while the client waits.

SERVER:

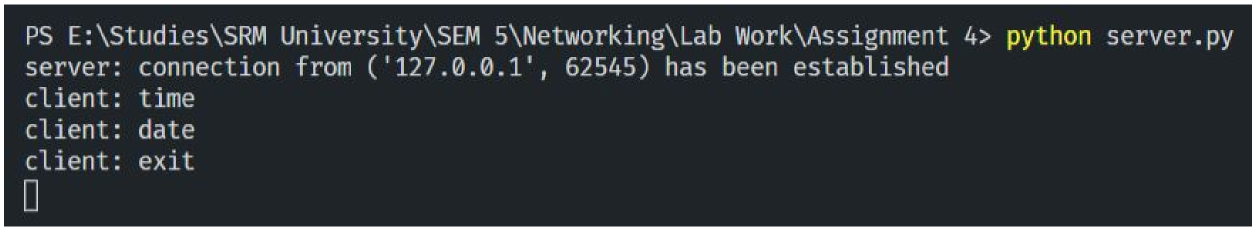
CODE:



server.py

```python
import socket
import datetime

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1", 12345))
s.listen(5)
while 1:
    cs, add = s.accept()
    print(f"server: connection from {add} has been established")
    val = ""
    while val != "exit":
        msg = cs.recv(1024)
        print("client: " + msg.decode("utf-8"))
        if msg.decode("utf-8") == "time":
            val = str(datetime.datetime.now().time())
            cs.send(bytes(val, "utf-8"))
        elif msg.decode("utf-8") == "date":
            val = str(datetime.datetime.now().date())
            cs.send(bytes(val, "utf-8"))
        elif msg.decode("utf-8") == "exit":
            break
        else:
            val = "invalid request"
            cs.send(bytes(val, "utf-8"))
```

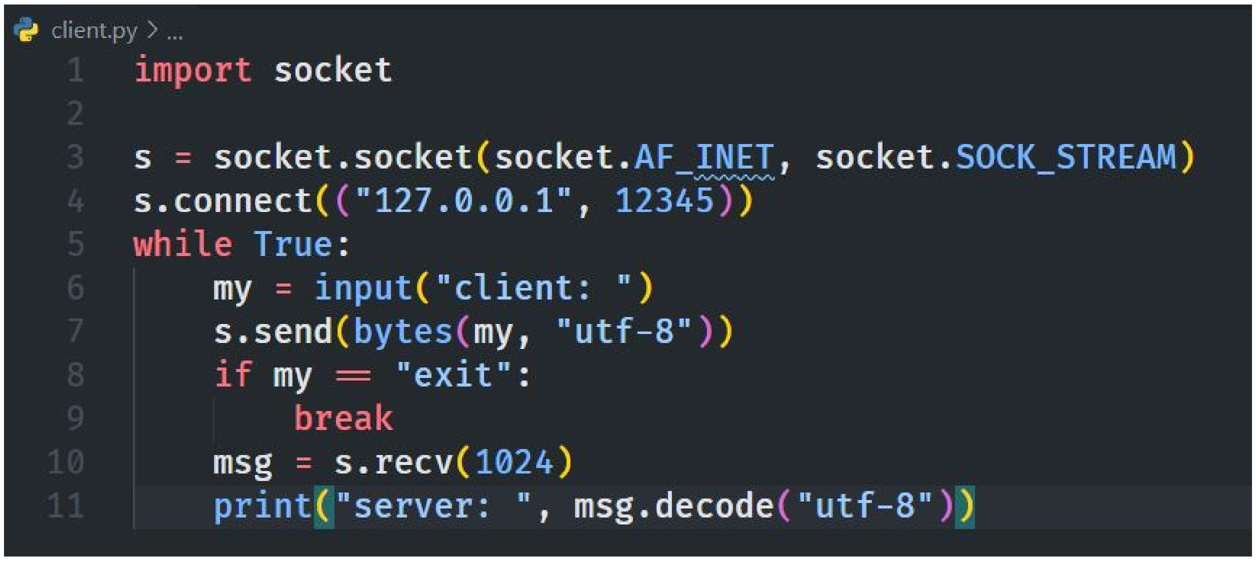
OUTPUT:



```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 4> python server.py
server: connection from ('127.0.0.1', 62545) has been established
client: time
client: date
client: exit
```

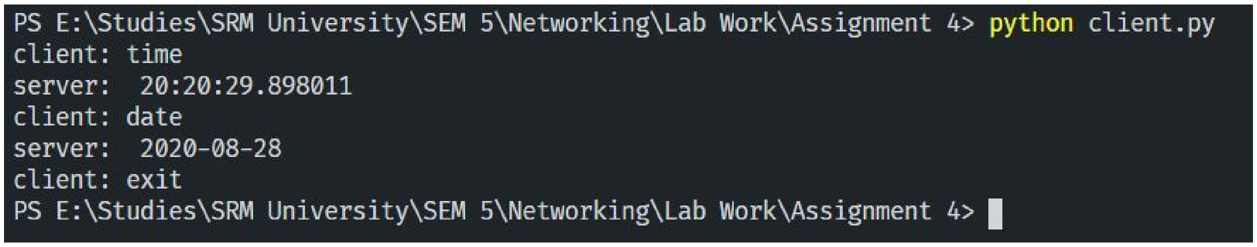
CLIENT:

CODE:



```
client.py > ...
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 12345))
while True:
    my = input("client: ")
    s.send(bytes(my, "utf-8"))
    if my == "exit":
        break
    msg = s.recv(1024)
    print("server: ", msg.decode("utf-8"))
```

OUTPUT:



```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 4> python client.py
client: time
server:  20:20:29.898011
client: date
server:  2020-08-28
client: exit
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 4>
```

**Result :** A TCP/IP connection was established successfully and the date and time was returned as required. The client will receive the date and time from the server.

**Experiment #5**

**Date : 04 September 2020**

## Half Duplex Chat Using TCP/IP

**Aim :** The application is a rudimentary chat application that enables one to chat with the server in half-duplex mode until the connection is explicitly closed.

**Technical Objectives :** Technologies that employ half-duplex operation are capable of sending information in both directions between two nodes, but only one direction or the other can be utilized at a time. This is a fairly common mode of operation when there is only a single network medium (cable, radio frequency and so forth) between devices**.**

**Given Requirements :** While this term is often used to describe the behavior of a pair of devices, it can more generally refer to any number of connected devices that take turns transmitting. For example, in conventional Ethernet networks, any device can transmit, but only one may do so at a time. For this reason, regular (unswitched) Ethernet networks are often said to be "half-duplex", even though it may seem strange to describe a LAN that way.

Algorithm SERVER-

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Initialize server address to 0 using the bzero function.
4. Assign the sin_family to AF_INET, sin_addr to INADDR_ANY, sin_port to dynamically assigned port number.
5. Bind the local host address to the socket using the bind function.

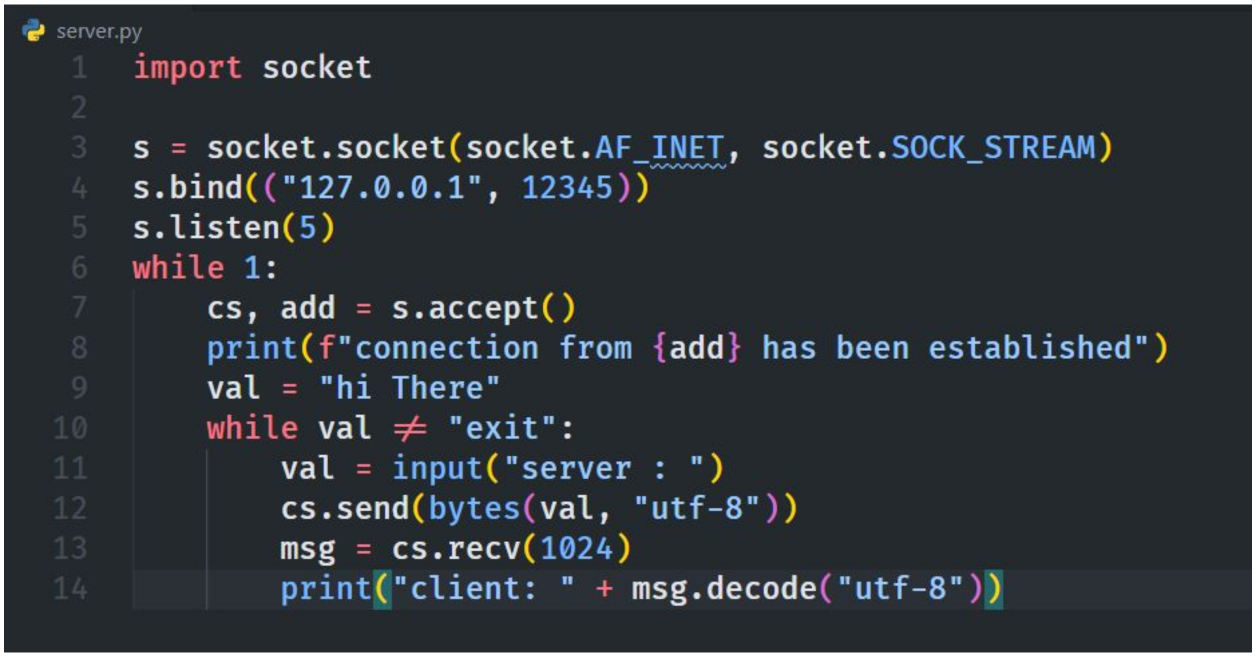
6. Listen on the socket for connection requests from the client.
7. Accept connection requests from the Client using accept function.
8. Fork the process to receive a message from the client and print it on the console.
9. Read messages from the console and send it to the client.

Algorithm CLIENT-

1. Include the necessary header files.
2. Create a socket using a socket function with the family AF_INET, type as SOCK_STREAM.
3. Initialize server address to 0 using the bzero function.
4. Assign the sin_family to AF_INET.
5. Get the server IP address and the Port number from the console.
6. Using gethostbyname function assign it to a hostent structure, and assign it to sin_addr of the server address structure.
7. Request a connection from the server using the connect function.
8. Fork the process to receive a message from the server and print it on the console.
9. Read messages from the console and send it to the server.

Code

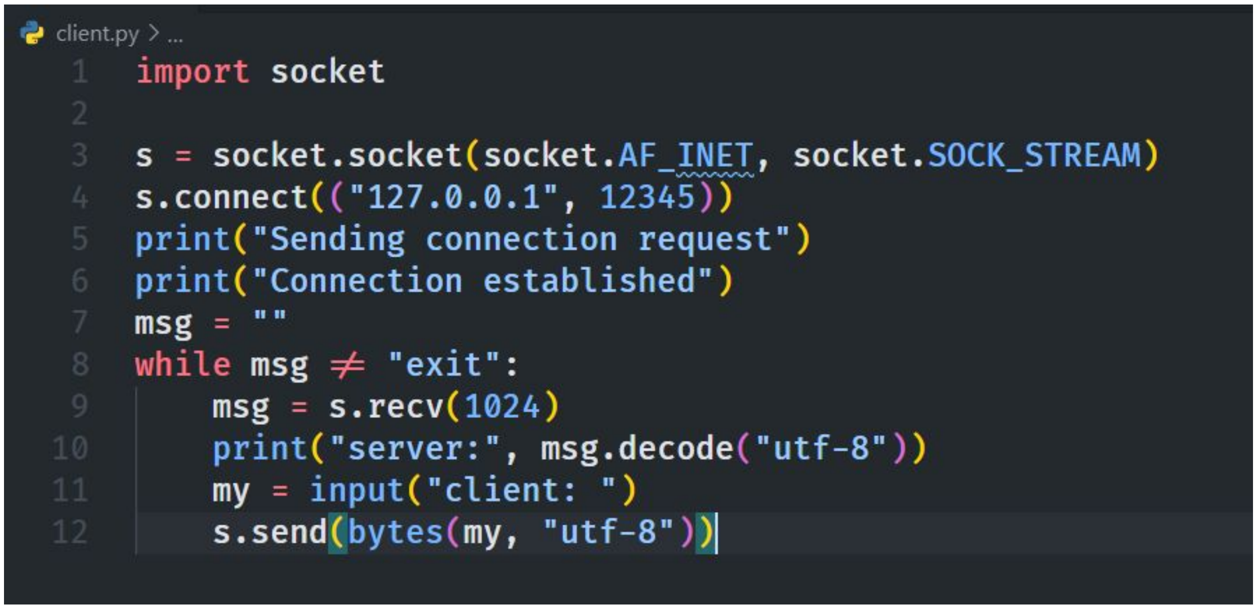
SERVER-



server.py

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1", 12345))
s.listen(5)
while 1:
    cs, add = s.accept()
    print(f"connection from {add} has been established")
    val = "hi There"
    while val ≠ "exit":
        val = input("server : ")
        cs.send(bytes(val, "utf-8"))
        msg = cs.recv(1024)
        print("client: " + msg.decode("utf-8"))
```

Code CLIENT-

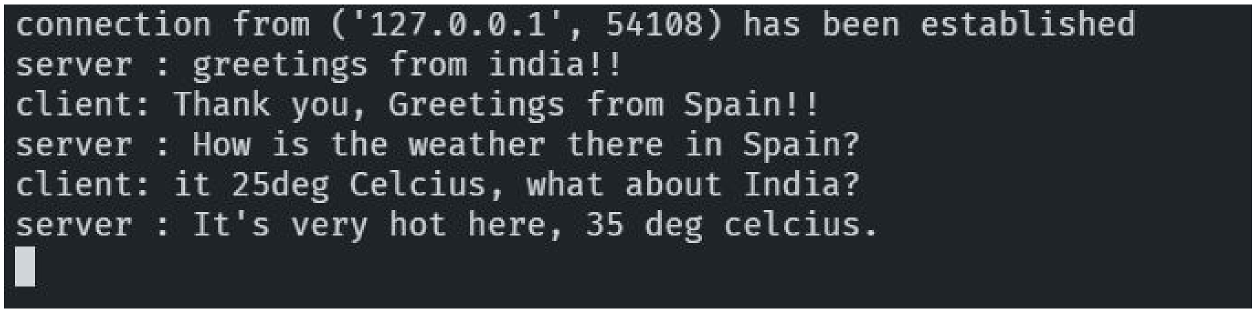


client.py > ...

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 12345))
print("Sending connection request")
print("Connection established")
msg = ""
while msg ≠ "exit":
    msg = s.recv(1024)
    print("server:", msg.decode("utf-8"))
    my = input("client: ")
    s.send(bytes(my, "utf-8"))
```

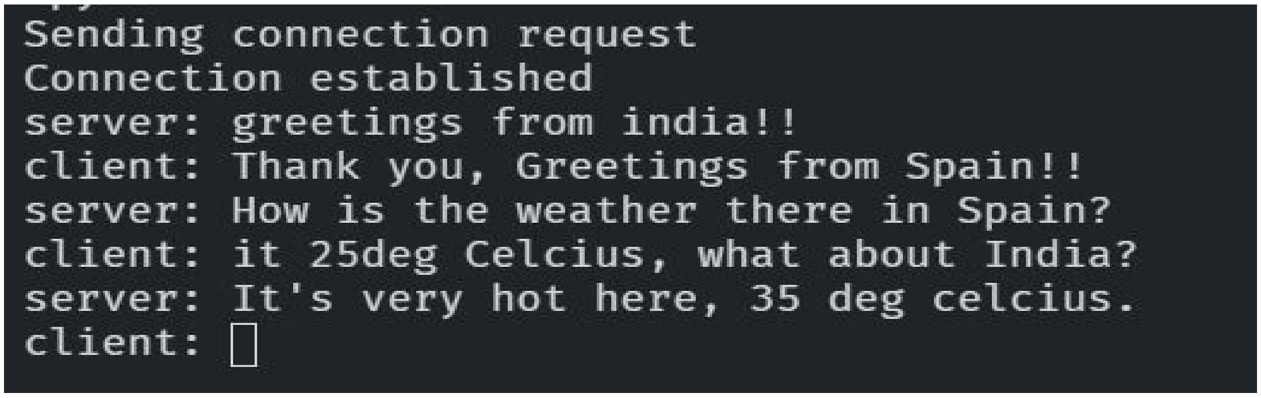
Output

Server-



```
connection from ('127.0.0.1', 54108) has been established
server : greetings from india!!
client: Thank you, Greetings from Spain!!
server : How is the weather there in Spain?
client: it 25deg Celcius, what about India?
server : It's very hot here, 35 deg celcius.
```

Output Client-



```
Sending connection request
Connection established
server: greetings from india!!
client: Thank you, Greetings from Spain!!
server: How is the weather there in Spain?
client: it 25deg Celcius, what about India?
server: It's very hot here, 35 deg celcius.
client:
```

**Result :** Half duplex application, where the Client establishes a connection with the Server. The Client can send and the server well receive messages at the same time, successfully implemented.

**Lab Experiment #6**

**Date : 11 September 2020**

## Full Duplex Chat Using TCP/IP

**Aim :** Creating a full duplex connection. Implementing a simple tcp chat between a server and a client. I'm using multi-threading so the server and the client can send and receive data at the same time (full duplex).

**Technical Objective :** A full-duplex link can only connect two devices, so many such links are required if multiple devices are to be connected together. Note that the term "full-duplex" is somewhat redundant; "duplex" would suffice, but everyone still says "full-duplex" (likely, to differentiate this mode from half-duplex).

**Given Requirements :** In full-duplex operation, a connection between two devices is capable of sending data in both directions simultaneously. Full-duplex channels can be constructed either as a pair of simplex links or using one channel designed to permit bidirectional simultaneous transmissions.

**Algorithm Server-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Bind the local host address to the socket using the bind function.
4. Listen on the socket for connection requests from the client.
5. Accept connection requests from the Client using accept function.
6. Fork the process to receive a message from the client and print it on the console and another process to read messages from the console and send it to the client

**Algorithm Client-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Get the server IP address and the Port number from the console.
4. Using gethostbyname function assign it to a hostent structure, and assign it to sin_addr of the server address structure.
5. Request a connection from the server using the connect function.
6. Fork the process to receive a message from the server and print it on the console and another process to read messages from the console and send it to the server simultaneously.

## Code Server-

```python
import socket
import threading
import sys

FLAG = False

def recv_from_client(conn):
    global FLAG
    try:
        while True:
            if FLAG == True:
                break
            message = conn.recv(1024).decode()

            if message == 'quit':
                conn.send('quit'.encode())
                conn.close()
                print('Connection Closed')
                FLAG = True
                break
            print('Client: ' + message)
    except:
        conn.close()

def send_to_client(conn):
    global FLAG
    try:
        while True:
            if FLAG == True:
                break
            send_msg = input('')
            if send_msg == 'quit':
                conn.send('quit'.encode())
                conn.close()
                print('Connection Closed')
                FLAG = True
                break
            conn.send(send_msg.encode())
    except:
        conn.close()

def main():
    threads = []
    global FLAG

    HOST = 'localhost'
    serverPort = 6500

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

    serverSocket.bind((HOST, serverPort))
    print("Socket binded.")

    serverSocket.listen(1)
    print("Listening.....")

    connectionSocket, addr = serverSocket.accept()
    print('Connection Established with a Client on ',
addr, '\n')

    t_rcv = threading.Thread(target=recv_from_client,
args=(connectionSocket,))
    t_send = threading.Thread(target=send_to_client,
args=(connectionSocket,))

    threads.append(t_rcv)
    threads.append(t_send)
    t_rcv.start()
```

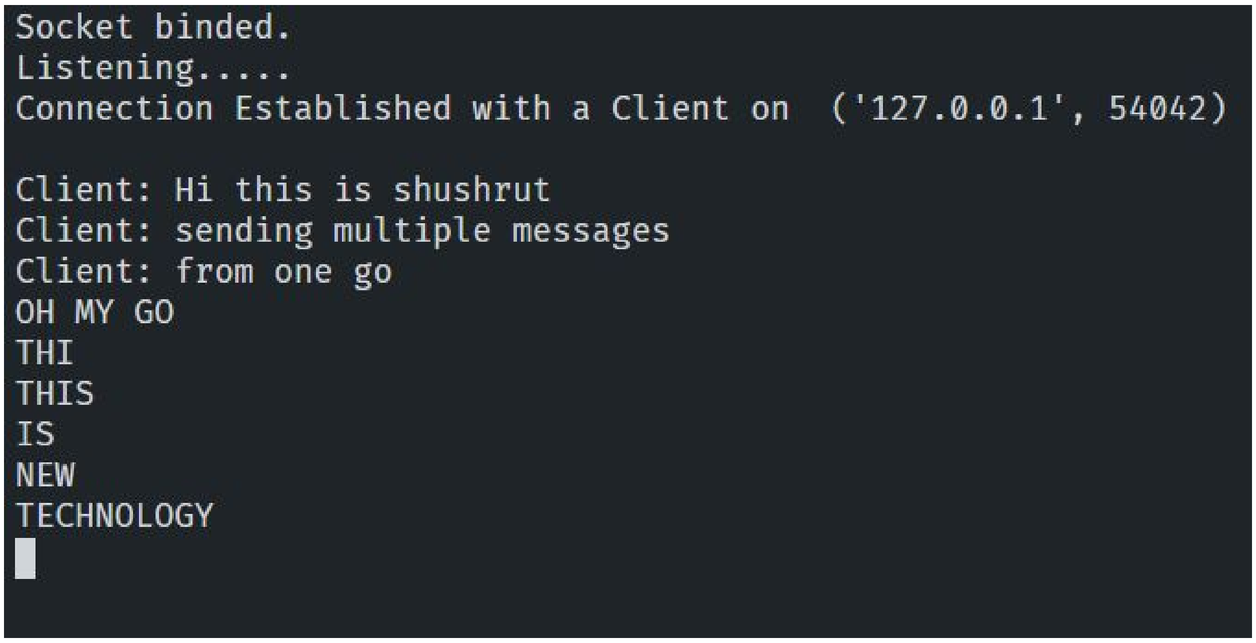
```python
    t_send.start()

    t_rcv.join()
    t_send.join()

    print('EXITING')
    serverSocket.close()

    sys.exit()

if _name_ == '_main_':
    main()
```

**Code Client-**

```python
import socket
import threading
import sys

FLAG = False

def send_to_server(clsock):
    global FLAG
    while True:
        if FLAG == True:
            break
        send_msg = input('')
        clsock.sendall(send_msg.encode())



def recv_from_server(clsock):
    global FLAG
    while True:
        data = clsock.recv(1024).decode()
        if data == 'quit':
            print('Closing connection')
            FLAG = True
            break
        print('Server: ' + data)

# this is main function
def main():
    threads = []
    HOST = 'localhost'
    PORT = 6500

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

    clientSocket.connect((HOST, PORT))
    print('Client is connected to the Server\n')

        t_send  =  threading.Thread(target=send_to_server,
args=(clientSocket,))

        t_rcv  =  threading.Thread(target=recv_from_server,
args=(clientSocket,))

    threads.append(t_send)
    threads.append(t_rcv)
    t_send.start()
    t_rcv.start()

    t_send.join()
    t_rcv.join()

    print('EXITING')
    sys.exit()

if _name_ == '_main_':
    main()
```

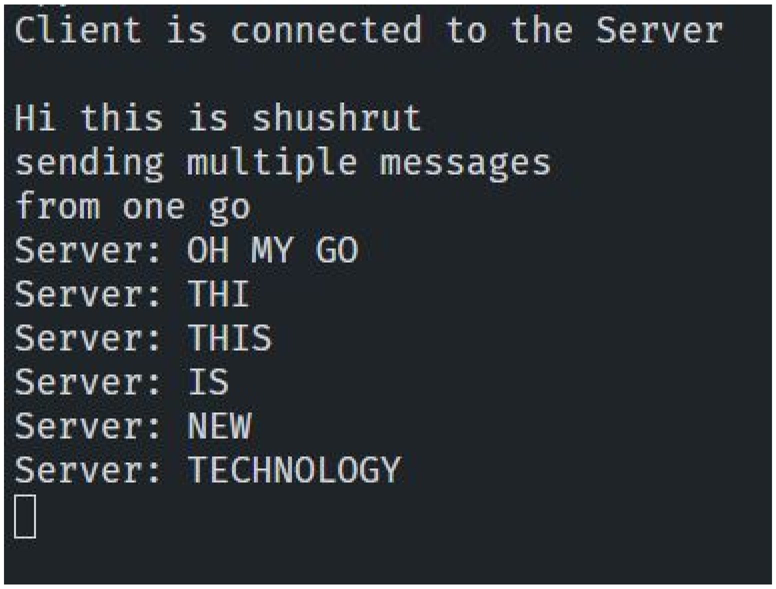
**Output**

**Server-**



```
Socket binded.
Listening.....
Connection Established with a Client on  ('127.0.0.1', 54042)

Client: Hi this is shushrut
Client: sending multiple messages
Client: from one go
OH MY GO
THI
THIS
IS
NEW
TECHNOLOGY
```

**Output Client-**



```
Client is connected to the Server

Hi this is shushrut
sending multiple messages
from one go
Server: OH MY GO
Server: THI
Server: THIS
Server: IS
Server: NEW
Server: TECHNOLOGY
```

**Result :** full duplex application, where the Client establishes a connection with the Server. The Client and Server can send as well as receive messages at the same time. Both the Client and Server exchange messages, successfully implemented.

**Experiment #7**

**Date: 25 September 2020**

## Implementation Of File Transfer Protocol

**Aim :** There are two hosts, Client and Server. The Client sends the name of the file it needs from the Server and the Server sends the contents of the file to the Client, where it is stored in a file.

**Technical Objective :** To implement FTP application, where the Client on establishing a connection with the Server sends the name of the file it wishes to access remotely. The Server then sends the contents of the file to the Client, where it is stored.

**Give Requirements :** There are two hosts, Client and Server. The Client sends the name of the file it needs from the Server and the Server sends the contents of the file to the Client, where it is stored in a file.

**Algorithm Server-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Bind the local host address to the socket using the bind function.
4. Listen on the socket for connection  requests from the client.
5. Accept connection requests from the Client using accept function.
6. Fork the process to receive a message from the client and print it on the console and another process to read messages from the console and send it to the client simultaneously

**Algorithm Client-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Get the server IP address and the Port number from the console.
4. Using gethostbyname function assign it to a hostent structure, and assign it to sin_addr of the server address structure.
5. Request a connection from the server using the connect function.
6. Fork the process to receive a message from the server and print it on the console and another process to read messages from the console and send it to the server.

**Code : Server -**

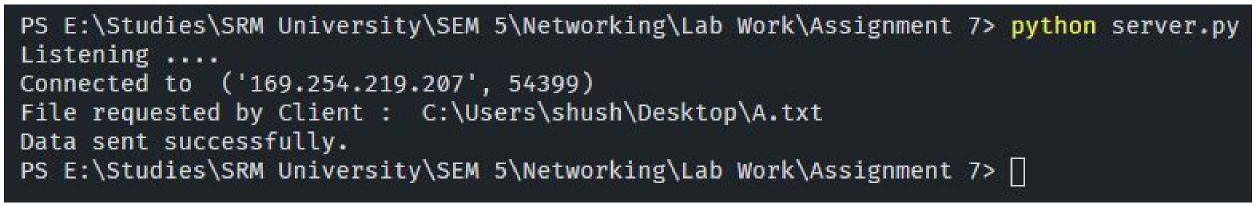
```
import socket
s = socket.socket()
host = socket.gethostname()
port = 5004
s.connect((host,port))
print("Connected.")
sentence = input(">> ")
s.send(sentence.encode())

filename = input(str("Enter incoming file name: "))
file = open(filename, 'a')
file_data = s.recv(1024)
file.write("\n")
file.write(file_data.decode("utf-8"))
file.close()
```

```
print("File received successfully.")
```

**Code : Client-**

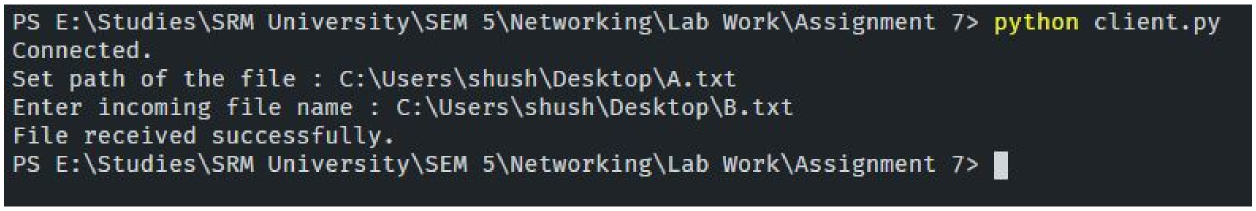
```
import socket
s = socket.socket()
host = socket.gethostname()
port = 5004
s.connect((host,port))
print("Connected.")
sentence = input(">> ")
s.send(sentence.encode())

filename = input(str("Enter incoming file name: "))
file = open(filename, 'a')
file_data = s.recv(1024)
file.write("\n")
file.write(file_data.decode("utf-8"))
file.close()
print("File received successfully.")

s.close()
```

**Output Client -**



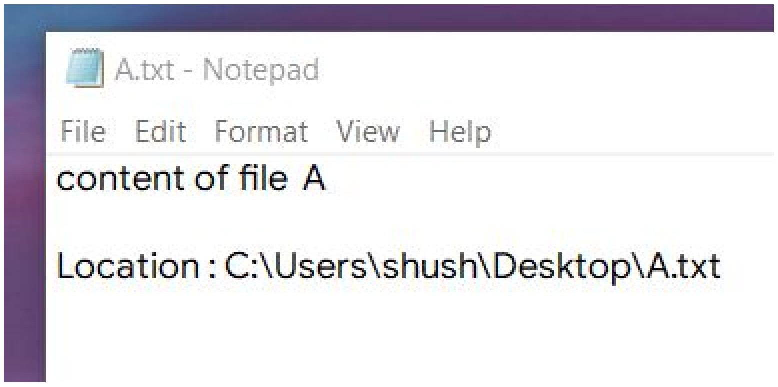
```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 7> python server.py
Listening ....
Connected to  ('169.254.219.207', 54399)
File requested by Client :  C:\Users\shush\Desktop\A.txt
Data sent successfully.
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 7>
```

**Output Server -**

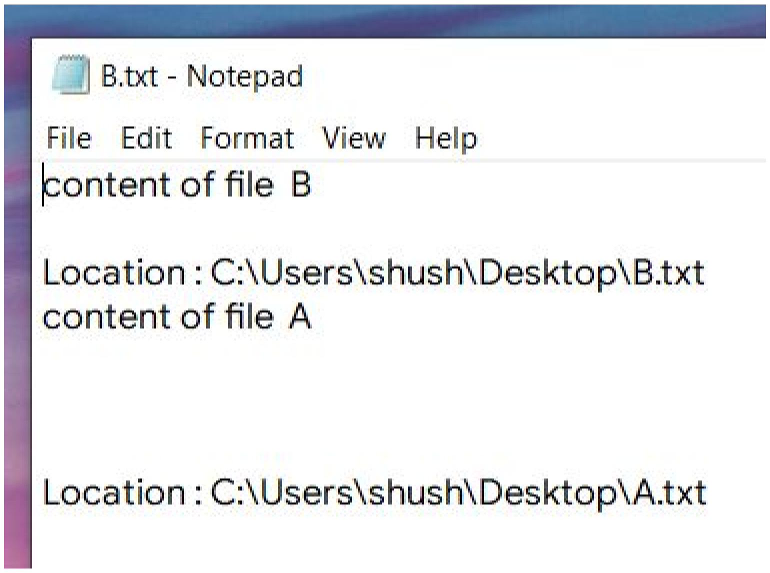


```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 7> python client.py
Connected.
Set path of the file : C:\Users\shush\Desktop\A.txt
Enter incoming file name : C:\Users\shush\Desktop\B.txt
File received successfully.
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 7>
```

**Output**



```
content of file A

Location : C:\Users\shush\Desktop\A.txt
```



```
content of file B

Location : C:\Users\shush\Desktop\B.txt
content of file A



Location : C:\Users\shush\Desktop\A.txt
```

**Result :** FTP application, where the Client on establishing a connection with the Server sends the name of the file it wishes to access remotely. The Server then sends the contents of the file to the Client, where it is stored. Successfully implemented

**Experiment #8**

**Date: 17-10-2020**

## Remote Command Execution Using UDP

**Aim :** Remote command server using TCP and UDP as transport protocol. Clients interact with the server sending some specific commands and the server holds that process, and sends back to client an response. The server logs all the commands that were sent by the client.

**Technical Objective :** Remote Command execution is implemented through this program using which Client is able to execute commands at the Server. Here, the Client sends the command to the Server for remote execution. The Server executes the command and the send result of the execution

**Given Requirements :** There are two hosts, Client and Server. The Client sends a command to the Server, which executes the command and sends the result back to the Client.

**Algorithm Server-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Bind the local host address to the socket using the bind function.
4. Listen on the socket for connection requests from the client.
5. Accept connection requests from the Client using accept function.
6. Fork the process to receive a message from the client and print it on the console and another process to read messages from the console and send it to the client simultaneously

**Algorithm Client-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Get the server IP address and the Port number from the console.
4. Using gethostbyname function assign it to a hostent structure, and assign it to sin_addr of the server address structure.
5. Request a connection from the server using the connect function.
6. Fork the process to receive a message from the server and print it on the console and another process to read messages from the console and send it to the server.

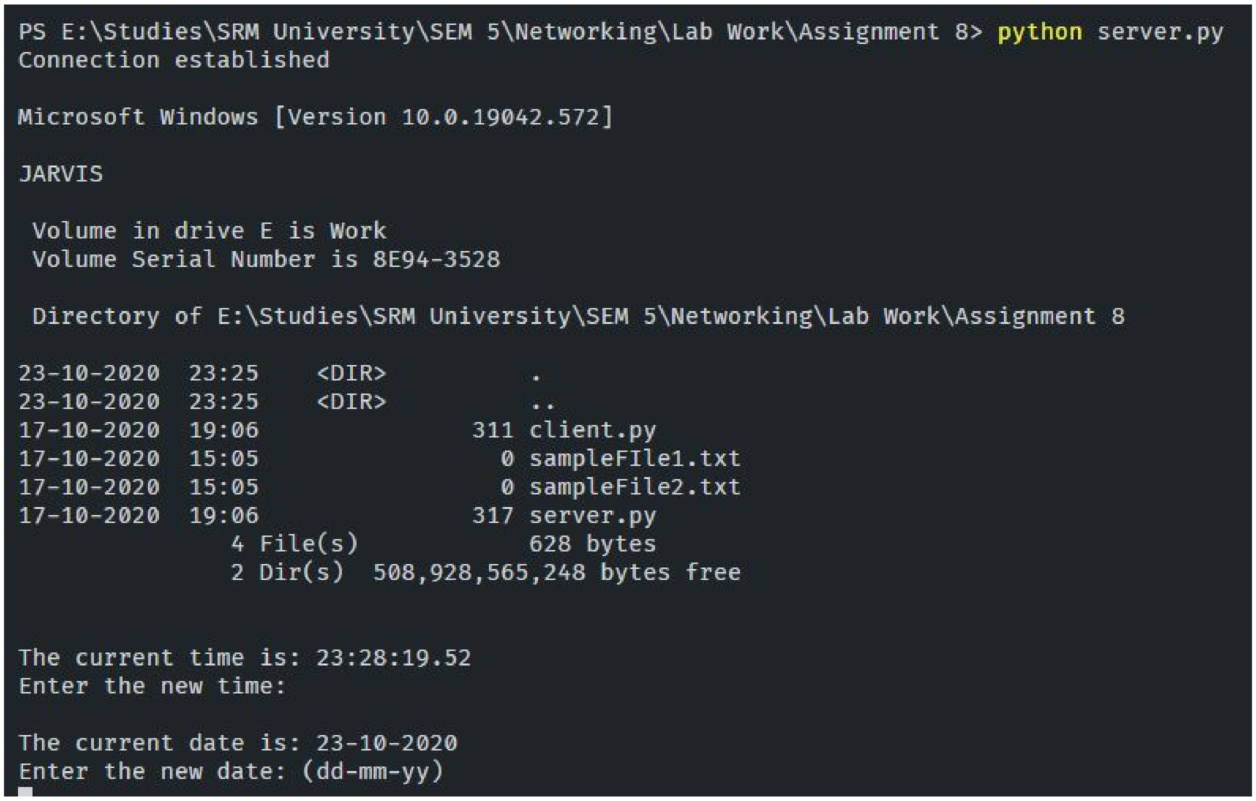
**Code Server-**

```
import socket

import os

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

s.bind(("", 6789))

print(s.recvfrom(1024)[0].decode())

while True:

    x = s.recvfrom(1024)

    if (x[0].decode() == "stop"):
        break
    ops = os.popen(x[0].decode())
    op = ops.read()
    print(op)
    s.sendto(op.encode(), x[1])
```

**Code Client-**

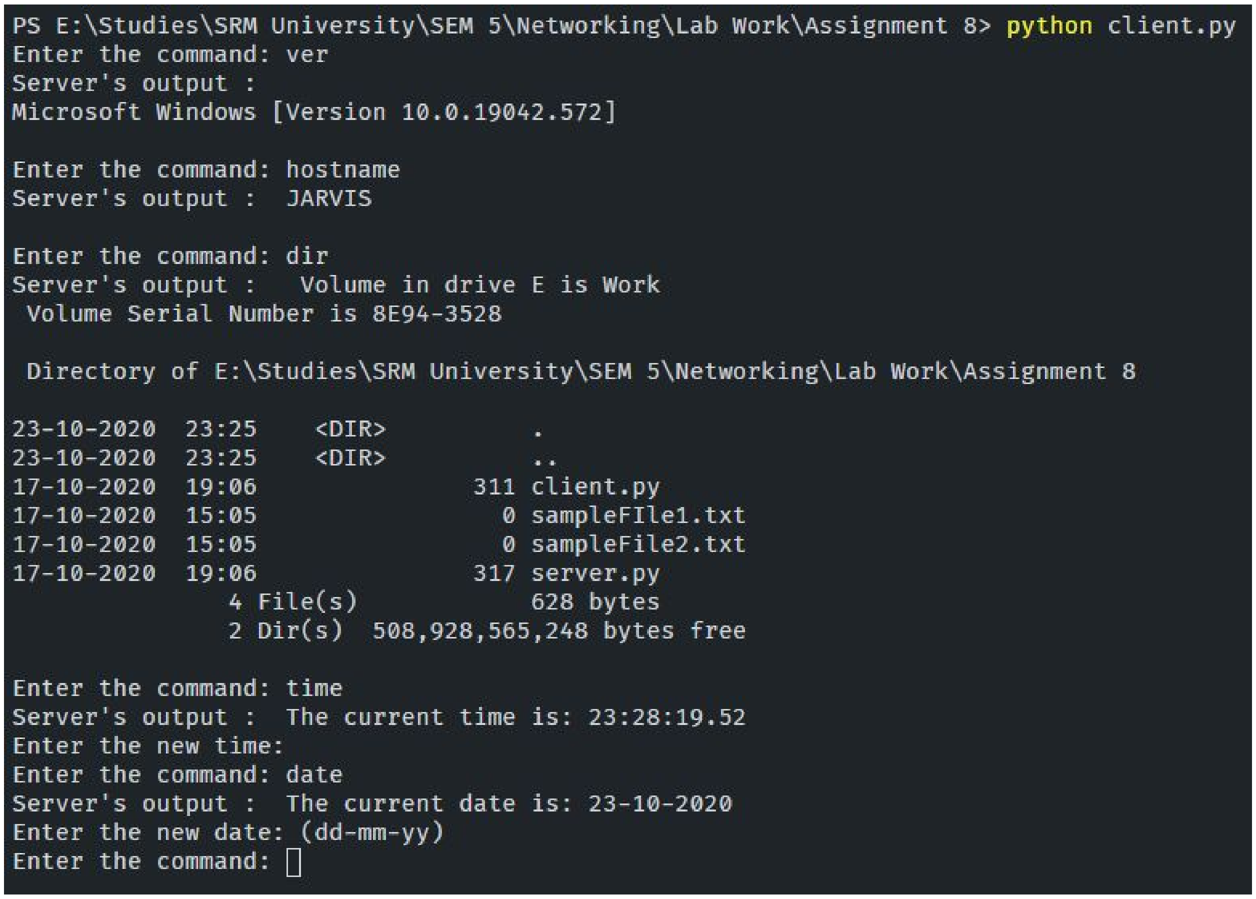
```
import socket
import os
cl = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
addr = ("192.168.1.2", 6789)
cl.sendto("Connection established".encode(), addr)
while True:
    x = input("Enter the command: ")
    cl.sendto(x.encode(), addr)
    y = cl.recvfrom(1024)
    print("Server's output : ", y[0].decode())
```

## Output Server-



```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 8> python server.py
Connection established

Microsoft Windows [Version 10.0.19042.572]

JARVIS

 Volume in drive E is Work
 Volume Serial Number is 8E94-3528

 Directory of E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 8

23-10-2020  23:25    <DIR>          .
23-10-2020  23:25    <DIR>          ..
17-10-2020  19:06               311 client.py
17-10-2020  15:05                 0 sampleFIle1.txt
17-10-2020  15:05                 0 sampleFile2.txt
17-10-2020  19:06               317 server.py
               4 File(s)            628 bytes
               2 Dir(s)  508,928,565,248 bytes free


The current time is: 23:28:19.52
Enter the new time:

The current date is: 23-10-2020
Enter the new date: (dd-mm-yy)
```

**Output Server-**



```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 8> python client.py
Enter the command: ver
Server's output :
Microsoft Windows [Version 10.0.19042.572]

Enter the command: hostname
Server's output :  JARVIS

Enter the command: dir
Server's output :   Volume in drive E is Work
 Volume Serial Number is 8E94-3528

 Directory of E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 8

23-10-2020  23:25    <DIR>          .
23-10-2020  23:25    <DIR>          ..
17-10-2020  19:06               311 client.py
17-10-2020  15:05                 0 sampleFIle1.txt
17-10-2020  15:05                 0 sampleFile2.txt
17-10-2020  19:06               317 server.py
               4 File(s)            628 bytes
               2 Dir(s)  508,928,565,248 bytes free

Enter the command: time
Server's output :  The current time is: 23:28:19.52
Enter the new time:
Enter the command: date
Server's output :  The current date is: 23-10-2020
Enter the new date: (dd-mm-yy)
Enter the command: 
```

**Result :** Remote Command execution is implemented through this program using which Client is able to execute commands at the Server. Here, the Client sends the command to the Server for remote execution. The Server executes the command and the send result of the execution

**Experiment #9**

**Date: 17-10-2020**

## Arp Implementation Using UDP

**Aim :** Performing ARP implementation using UDP. Where we get the MAC address from the provided IP address in the input field when the network is the same.

**Technical Objective :** Address Resolution Protocol (ARP) is implemented through this program. The IP address of any Client is given as the input. The ARP cache is looked up for the corresponding hardware address. This is returned as the output. Before compiling that Client is pinged.

**Given Requirements :** There is a single host. The IP address of any Client in the network is given as input and the corresponding hardware address is got as the output.

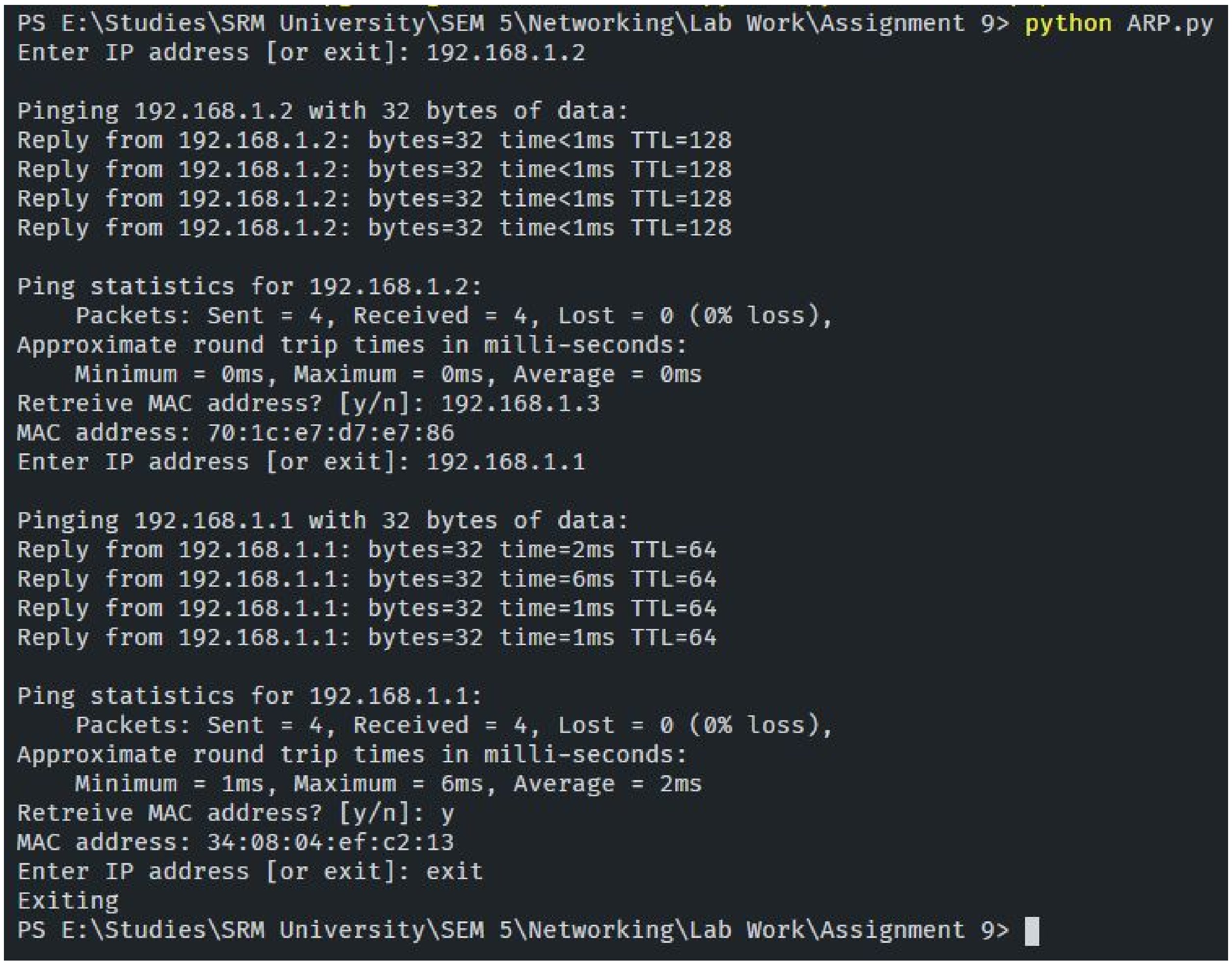
**Algorithm-**

1. Include the necessary header files.
2. Create a socket using a socket function with family AF_INET, type as SOCK_STREAM.
3. Bind the local host address to the socket using the bind function.
4. Listen on the socket for connection requests from the client.
5. Accept connection requests from the Client using accept function.
6. Fork the process to receive a message from the client and print it on the console and another process to read messages from the console and send it to the client simultaneously.

**Code-**

```python
import os
import getmac

ip = ""

while ip != "exit":
    ip = input("Enter IP address [or exit]: ")
    if ip == "exit":
        continue
    os.system("ping {}".format(ip))
    msg = input("Retreive MAC address? [y/n]: ")
    if msg == "n":
        continue
    mac = getmac.get_mac_address(ip = ip)
    print("MAC address: {}".format(mac))
print("Exiting")
```

**Output-**



```
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 9> python ARP.py
Enter IP address [or exit]: 192.168.1.2

Pinging 192.168.1.2 with 32 bytes of data:
Reply from 192.168.1.2: bytes=32 time<1ms TTL=128
Reply from 192.168.1.2: bytes=32 time<1ms TTL=128
Reply from 192.168.1.2: bytes=32 time<1ms TTL=128
Reply from 192.168.1.2: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.1.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
Retreive MAC address? [y/n]: 192.168.1.3
MAC address: 70:1c:e7:d7:e7:86
Enter IP address [or exit]: 192.168.1.1

Pinging 192.168.1.1 with 32 bytes of data:
Reply from 192.168.1.1: bytes=32 time=2ms TTL=64
Reply from 192.168.1.1: bytes=32 time=6ms TTL=64
Reply from 192.168.1.1: bytes=32 time=1ms TTL=64
Reply from 192.168.1.1: bytes=32 time=1ms TTL=64

Ping statistics for 192.168.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 6ms, Average = 2ms
Retreive MAC address? [y/n]: y
MAC address: 34:08:04:ef:c2:13
Enter IP address [or exit]: exit
Exiting
PS E:\Studies\SRM University\SEM 5\Networking\Lab Work\Assignment 9>
```

**Result :** Address Resolution Protocol is implemented through this program. The IP address of any Client is given as the input. The ARP cache is looked up for the corresponding hardware address. This is returned as the output. Before compiling that Client is pinged.

**Experiment #10**

**Date: 03-11-2020**

## Study Of IPv6 Addressing And Subnetting

**Aim**- To study IPv6 addressing and subnetting IPv6 Addressing Modes Addressing mode refers to the mechanism of hosting an address on the network. IPv6 offers several types of modes by which a single host can be addressed. More than one host can be addressed at once or the host at the closest distance can be addressed.

**Technical Objective:** IPv6 addresses use 128 bits to represent an address which includes bits to be used for subnetting. ... The second half of the address (least significant 64 bits) is always used for hosts only. Therefore, there is no compromise if we subnet the network. Simulate the same.

**Given Requirements:** IPv6 protocol has to be studied in detail and then simulated using routers and PC in Cisco packet tracer simulation software with the packet ICMPv6 to demonstrate subnet mask feature of IPv6 protocol.

**Unicast:**

In unicast mode of addressing, an IPv6 interface (host) is uniquely identified in a network segment. The IPv6 packet contains both source and destination IP addresses. A host interface is equipped with an IP address which is unique in that network segment. When a network switch or a router receives a unicast IP packet, destined to a single host, it sends out one of its outgoing interfaces which connects to that particular host.

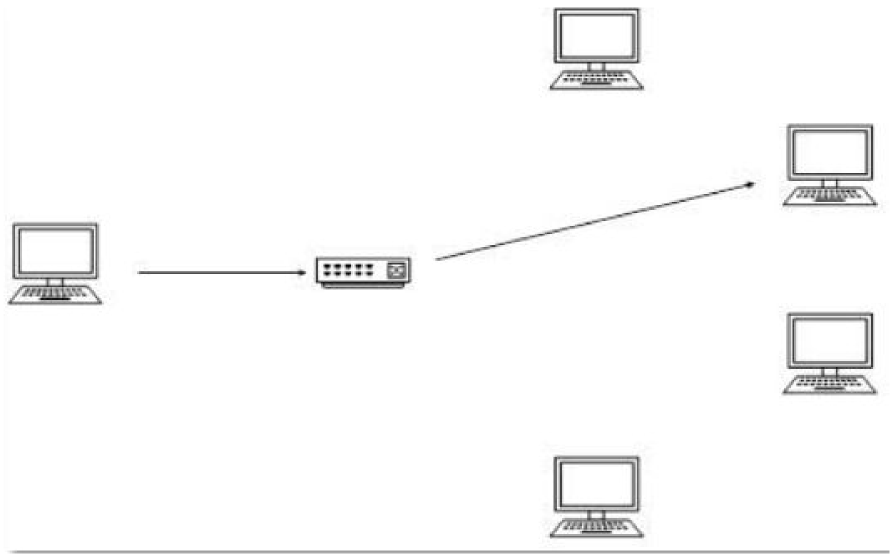


Fig 10.1 :**Unicast**

**Multicast :**

The IPv6 multicast mode is the same as that of IPv4. The packet destined to multiple hosts is sent on a special multicast address. All the hosts interested in that multicast information, need to join that multicast group first. All the interfaces that joined the group receive the multicast packet and process it, while other hosts not interested in multicast packets ignore the multicast information.

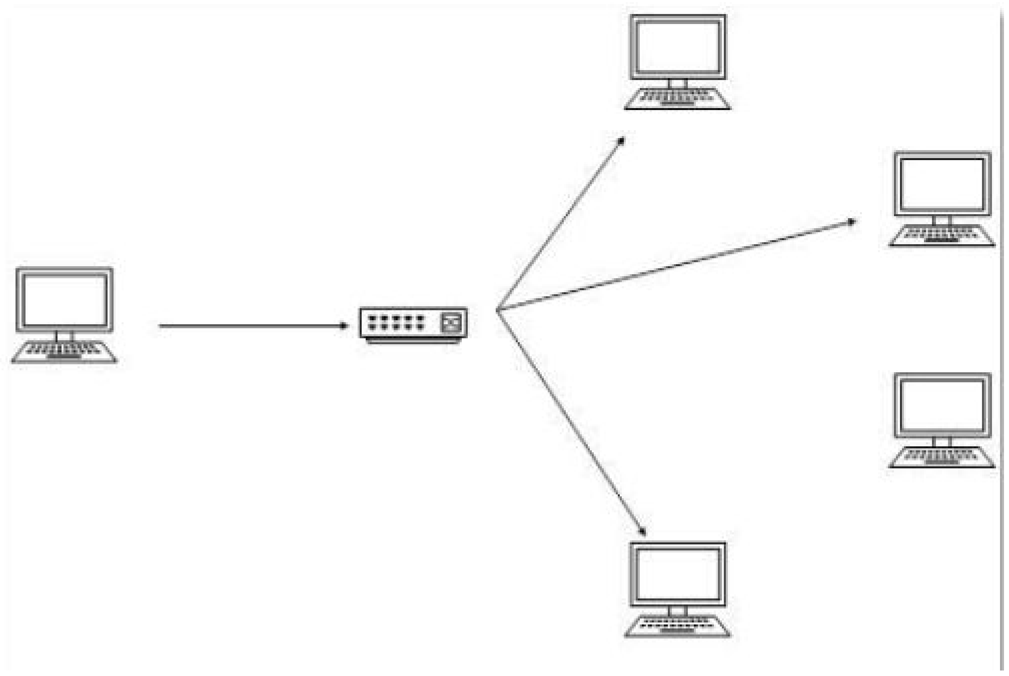


Fig 10.2 :**Multicast**

**Anycast :**

IPv6 has introduced a new type of addressing, which is called Anycast addressing. In this addressing mode, multiple interfaces (hosts) are assigned the same Anycast IP address. When a host wishes to communicate with a host equipped with an Anycast IP address, it sends a Unicast message. With the help of a complex routing mechanism, that Unicast message is delivered to the host closest to the Sender in terms of Routing cost.

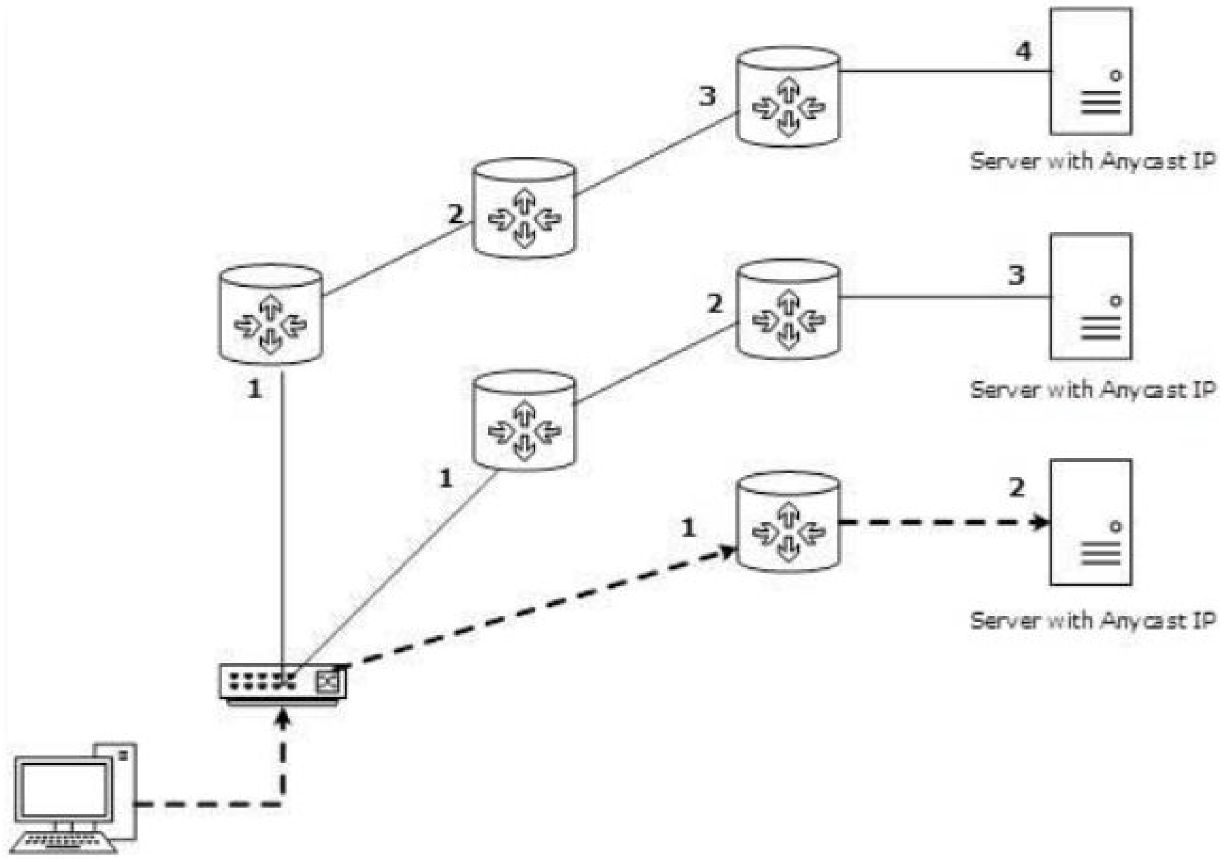


Fig 10.3 : **Anycast**

**Address Structure:**

An IPv6 address is made of 128 bits divided into eight 16-bits blocks. Each block is then converted into 4- digit Hexadecimal numbers separated by colon symbols. For example, given below is a 128-bit IPv6 address represented in binary format and divided into eight 16- bits blocks:

0010000000000001 0000000000000000 0011001000111000 1101111111100001 0000000001100011 0000000000000000 0000000000000000 1111111011111011

Each block is then converted into Hexadecimal and separated by ':' symbol: 2001:0000:3238:DFE1:0063:0000:0000:FEFB

**IPv6 Subnetting :**

In IPv4, addresses were created in classes. Classful IPv4 addresses clearly define the bits used for network prefixes and the bits used for hosts on that network. To subnet in IPv4, we play with the default classful netmask which allows us to borrow host bits to be used as subnet bits. This results in multiple subnets but less hosts per subnet. That is, when we borrow host bits to create a subnet, it costs us in lesser bit to be used for host addresses. IPv6 addresses use 128 bits to represent an address which includes bits to be used for subnetting. The second half of the address (least significant 64 bits) is always used for hosts only. Therefore, there is no compromise if we subnet the network.

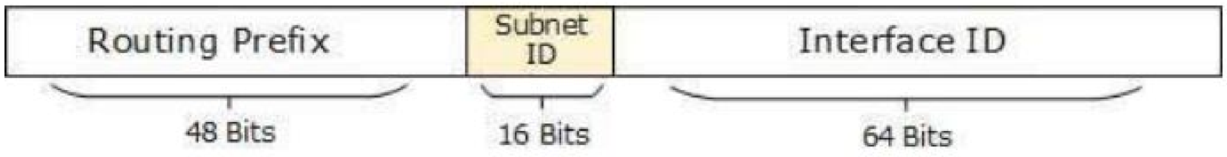


FIG 10.4 : **IPv6 Subnetting**

16 bits of subnet is equivalent to IPv4's Class B Network. Using these subnet bits, an organization can have another 65 thousand of subnets which is by far, more than enough. Thus, routing prefix is /64 and host portion is 64 bits. We can further subnet the network beyond 16 bits of Subnet ID, by borrowing host bits; but it is recommended that 64 bits should always be used for hosts addresses because auto-configuration requires 64 bits. IPv6 subnetting works on the same concept as Variable Length Subnet Masking in IPv4. /48 prefix can be allocated to an organization providing it the benefit of having up to /64 subnet prefixes, which is 65535

sub-networks, each having 264 hosts. A /64 prefix can be assigned to a point-to-point connection where there are only two hosts (or IPv6 enabled devices) on a link.

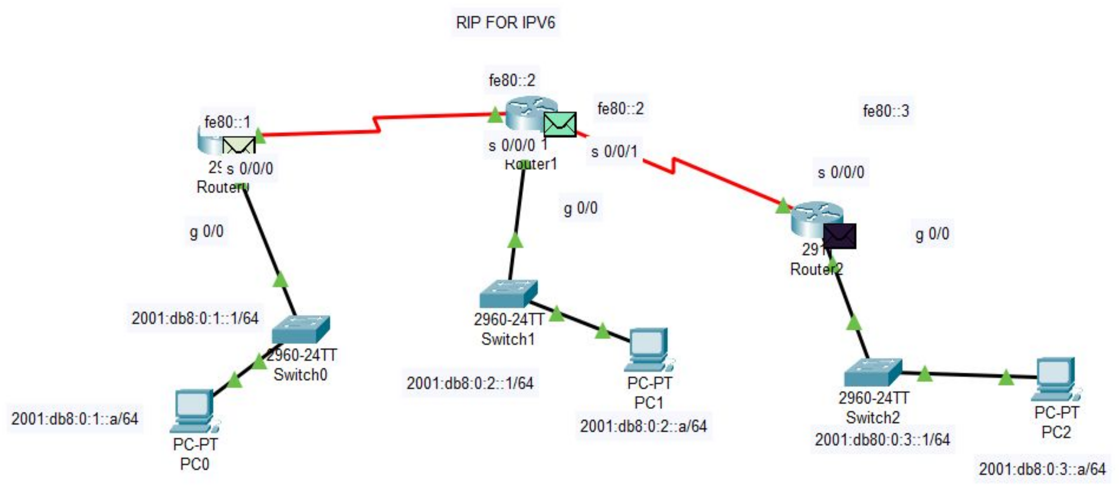


FIG 10.5 : **CISCO Packet Tracer Simulation**

**Result-** Study IPv6 addressing and subnetting IPv6 Addressing Modes Addressing mode refers to the mechanism of hosting an address on the network. IPv6 offers several types of modes by which a single host can be addressed. More than one host can be addressed at once or the host at the closest distance can be addressed.

**Lab Experiment #11**

**Date: 03-11-2020**

## Communication Using HDLC/PPP

**Aim-** To study about HDLC (High Level Data Link Control) and PPP (point to point): protocols in cisco packet tracer.

**Technical Objective:** The encapsulation protocols of HDLC and PPP are two different ways in which this is implemented within the connections, so after studying and understanding the differences between them, demonstrate the differences between them using Cisco packet tracer.

**Given Requirements:** HDLC and PPP are two different types of communication methods, so give a detailed description and analysis of both and show there difference in cisco packet tracer simulation software.

HDLC (High Level Data Link Control):

HDLC is a data link protocol used on synchronous serial data links. Because the standardized HDLC cannot support multiple protocols on a single link (lack of a mechanism to indicate which protocol is carried), Cisco developed a proprietary version of HDLC, called cHDLC, with a proprietary field acting as a protocol field. This field makes it possible for a single serial link to accommodate multiple network-layer protocols.

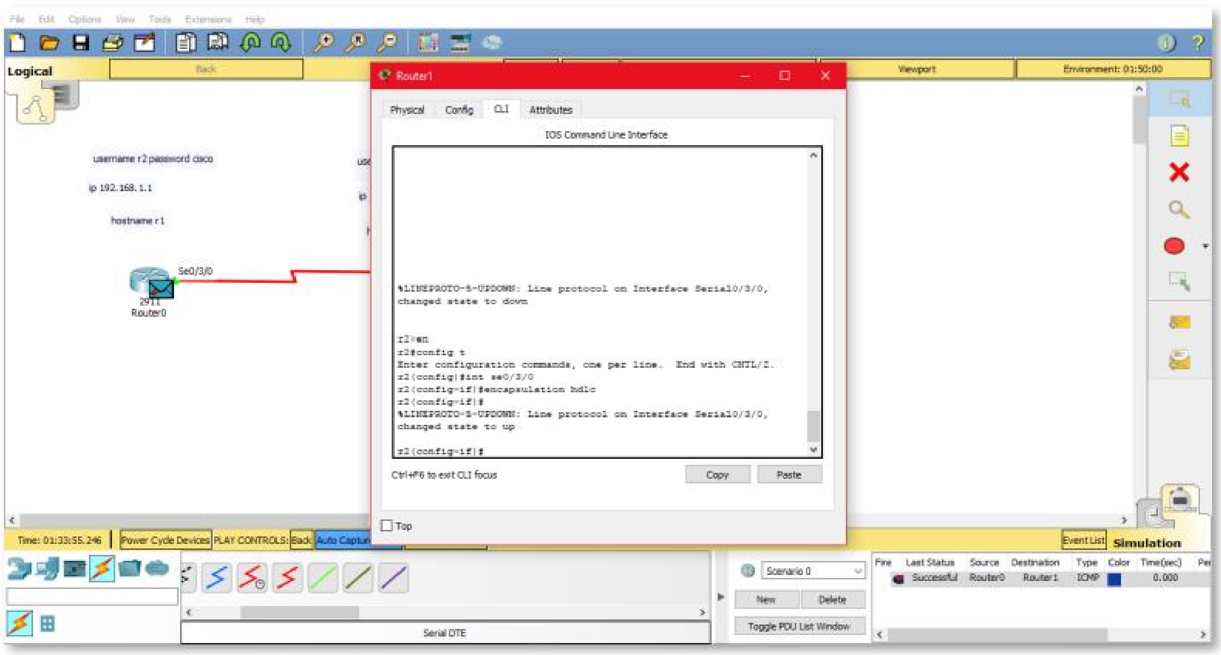
HDLC IN CISCO:

• By default, in the cisco packet tracer it is already HDLC.

• In case if it's not we can use the CLI commands to enable it the commands are

```
Router-B(config)#interface serial 0/0/0
```

```
Router-B(config-if)#encapsulation hdlc

Router-B(config-if)#ip address 192.168.10.5 255.255.255.252

Router-B(config-if)#no shutdown
```



username r2 password cisco

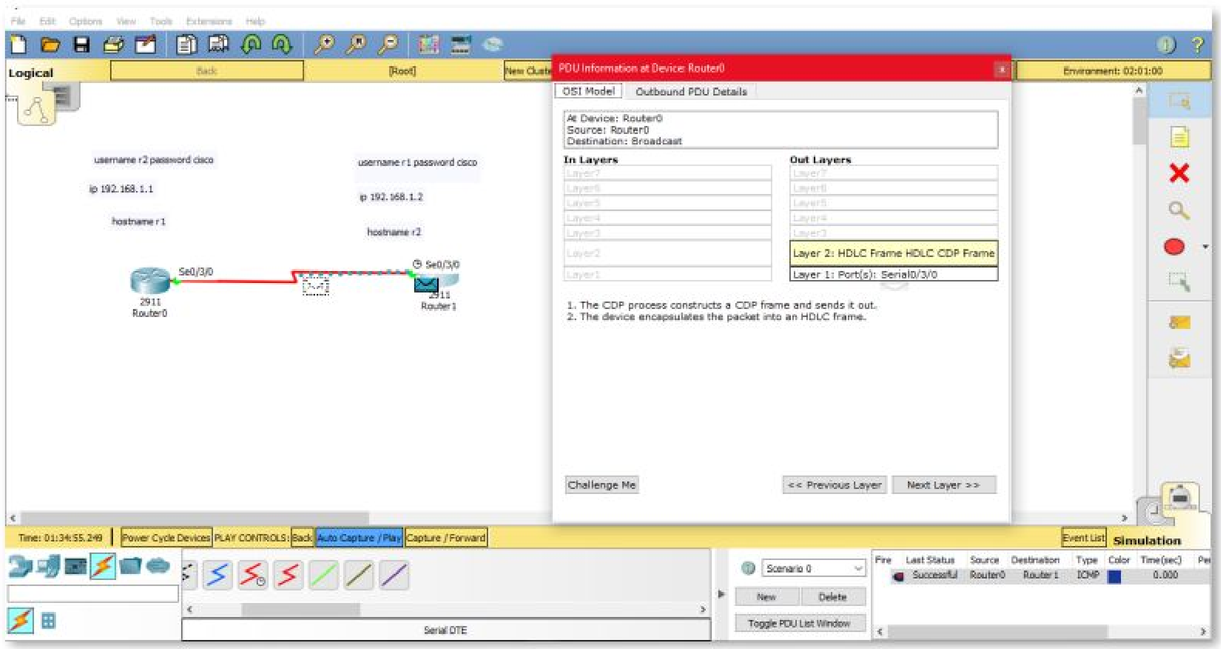
ip 192.168.1.1

hostname r1

```
%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/3/0,
changed state to down


r2>en
r2#config t
Enter configuration commands, one per line.  End with CNTL/Z.
r2(config)#int se0/3/0
r2(config-if)#encapsulation hdlc
r2(config-if)#
%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/3/0,
changed state to up

r2(config-if)#
```

| Fire | Last Status | Source | Destination | Type | Color | Time(sec) |
|---|---|---|---|---|---|---|
| ● | Successful | Router0 | Router1 | ICMP | 🟦 | 0.000 |

FIG 11.1 : **CISCO Packet Tracer Simulation CLI**



PDU Information at Device: Router0

OSI Model | Outbound PDU Details

```
At Device: Router0
Source: Router0
Destination: Broadcast

In Layers        Out Layers
Layer7           Layer7
Layer6           Layer6
Layer5           Layer5
Layer4           Layer4
Layer3           Layer3
Layer2           Layer 2: HDLC Frame HDLC CDP Frame
Layer1           Layer 1: Port(s): Serial0/3/0

1. The CDP process constructs a CDP frame and sends it out.
2. The device encapsulates the packet into an HDLC frame.
```

| Fire | Last Status | Source | Destination | Type | Color | Time(sec) |
|---|---|---|---|---|---|---|
| ● | Successful | Router0 | Router1 | ICMP | 🟦 | 0.000 |

FIG 11.2 : **CISCO Packet Tracer Simulation IPV6 Subnet masking**

**PPP (point to point):**

• The username of one router should be the hostname of other and if there are two routers this happens.

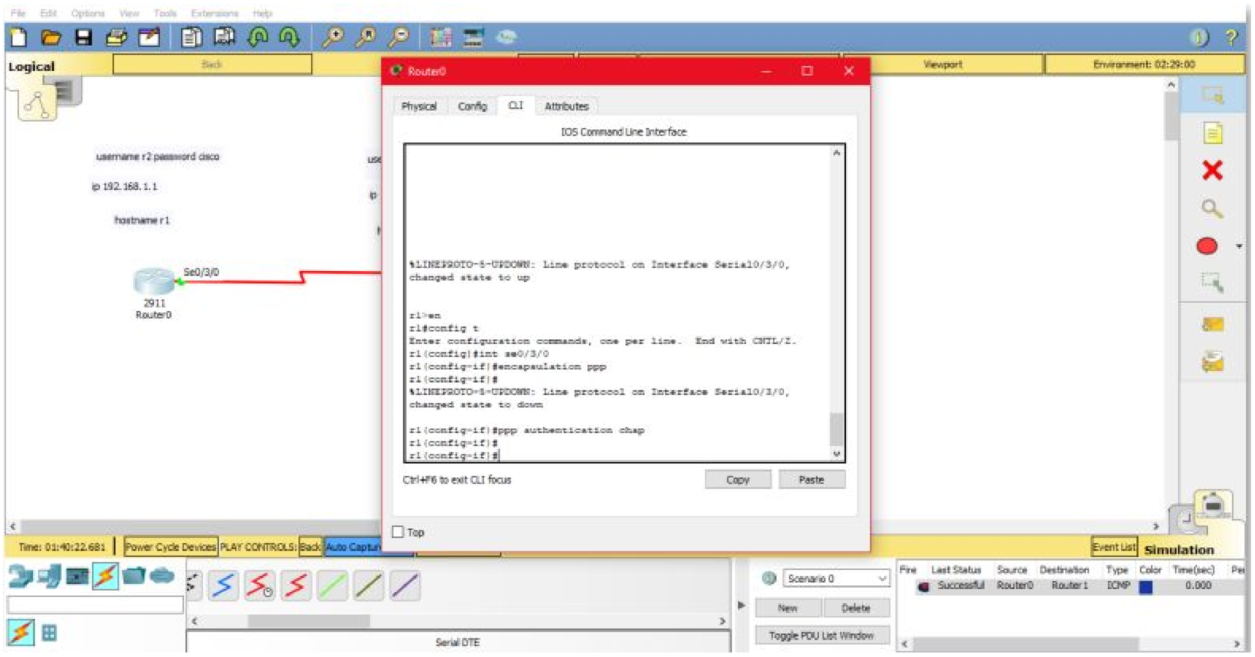
• And the passwords should be same for both routers in case if we take any two routers.

• And we are using chap in ppp to test this ppp The commands are:-

r1(config)# int se0/3/0

R1(config t)# encapsulation ppp

R1(config t)#ppp authentication chap.



username r2 password cisco

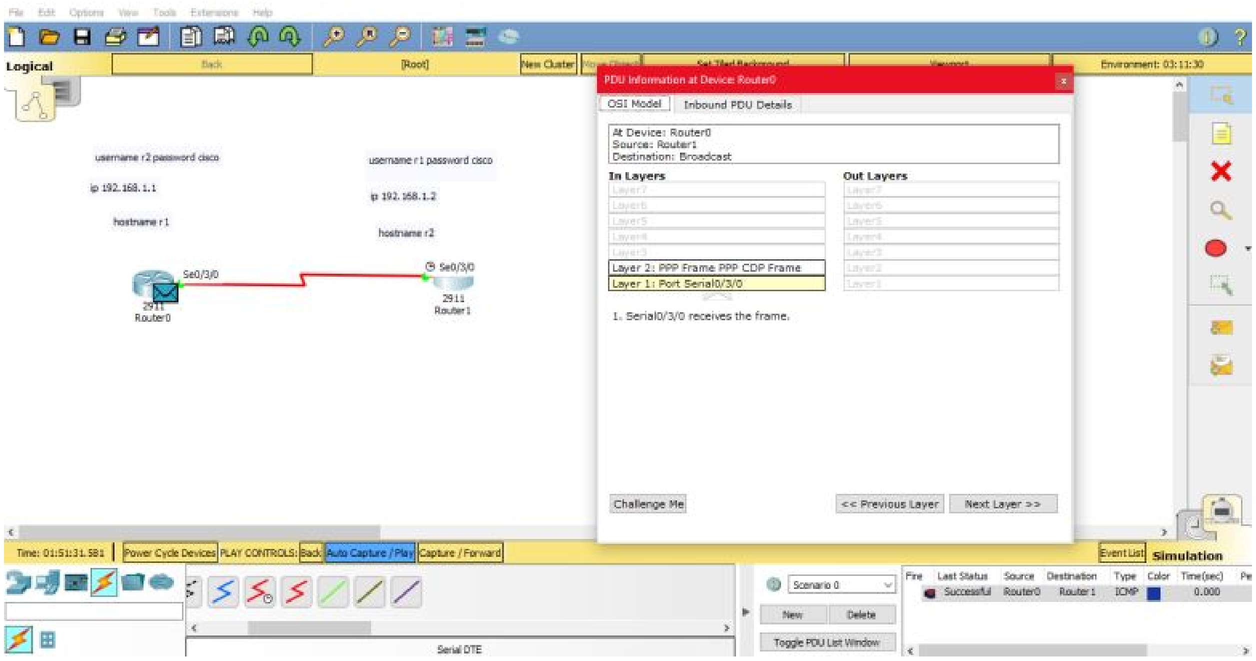
ip 192.168.1.1

hostname r1

```
%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/3/0,
changed state to up


r1>en
r1#config t
Enter configuration commands, one per line.  End with CNTL/Z.
r1(config)#int se0/3/0
r1(config-if)#encapsulation ppp
r1(config-if)#
%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/3/0,
changed state to down

r1(config-if)#ppp authentication chap
r1(config-if)#
r1(config-if)#
```

| Fire | Last Status | Source | Destination | Type | Color | Time(sec) |
|---|---|---|---|---|---|---|
| ● | Successful | Router0 | Router1 | ICMP | 🟦 | 0.000 |

FIG 11.3 : **CISCO Packet Tracer Simulation CLI**



username r2 password cisco

ip 192.168.1.1

hostname r1

username r1 password cisco

ip 192.168.1.2

hostname r2

PDU Information at Device: Router0

OSI Model | Inbound PDU Details

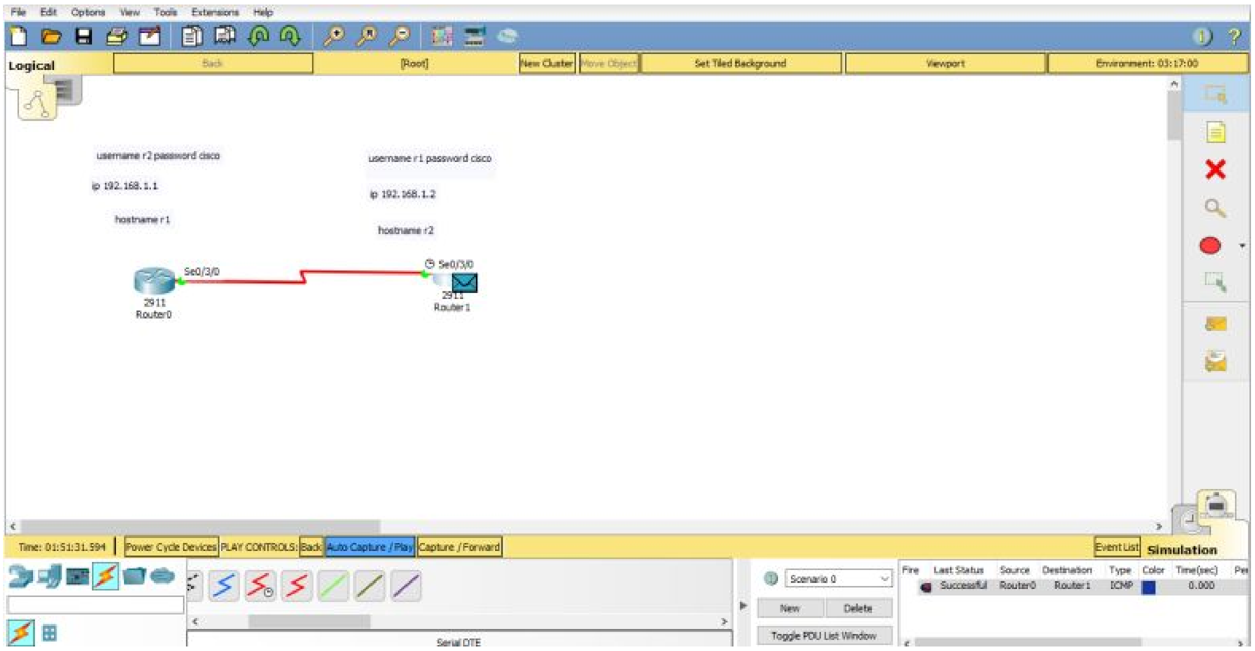
At Device: Router0
Source: Router1
Destination: Broadcast

| In Layers | Out Layers |
|---|---|
| Layer7 | Layer7 |
| Layer6 | Layer6 |
| Layer5 | Layer5 |
| Layer4 | Layer4 |
| Layer3 | Layer3 |
| Layer 2: PPP Frame PPP CDP Frame | Layer2 |
| Layer 1: Port Serial0/3/0 | Layer1 |

1. Serial0/3/0 receives the frame.

| Fire | Last Status | Source | Destination | Type | Color | Time(sec) |
|---|---|---|---|---|---|---|
| ● | Successful | Router0 | Router1 | ICMP | 🟦 | 0.000 |

FIG 11.4 : **CISCO Packet Tracer Simulation IPV6 Subnet Mask**



username r2 password cisco

ip 192.168.1.1

hostname r1

username r1 password cisco

ip 192.168.1.2

hostname r2

| Fire | Last Status | Source | Destination | Type | Color | Time(sec) | Pe |
|---|---|---|---|---|---|---|---|
| ● | Successful | Router0 | Router1 | ICMP | 🟦 | 0.000 | |

FIG 11.4 : **CISCO Packet Tracer Simulation**

**Result:** HDLC and PPP commands with encapsulation were studied successfully, using Cisco Packet Tracer and the simulation successfully completed.