Exercise 1: Study of necessary header files with respect to socket programming

Aim

To study the necessary header files with respect to socket programming.

**An Overview of The Sockets Module**

To use sockets in Python we will need to import the socket module.

s = socket.socket(socket\_family, socket\_type)

**Socket Family**

The primary way of using the sockets module is through the sockets() function which returns a socket object with methods to implement various system socket calls. Python sockets supports a number of address families under the network layer protocol, mainly:

1. AF\_INET — this is the most common, and uses IPv4 for network addressing. Most of the internet networking is presently done using IPv4.
2. AF\_INET6 — this is the next generation of the internet protocol using IPv6 and provides a number of features not available under IPv4.
3. AF\_UNIX — finally, this is the address family for Unix Domain Sockets (UDS), an inter-process communication protocol available on POSIX-compliant systems. This implementation allows passing of data between processes on an operating system without going through the network.

**Socket Types**

Depending on the transport layer protocol used, sockets can either be:

1. SOCK\_DGRAM for message-oriented datagram transport. These sockets are usually associated with the User Datagram Protocol (UDP) which provides an unreliable delivery of individual messages. Datagram sockets are commonly used when the order of messages is not important, such as when sending out the same data to multiple clients.
2. SOCK\_STREAM for stream-oriented transport often associated with the Transmission Control Protocol (TCP). TCP provides a reliable and ordered delivery of byte between two host, with error handling and control, making it useful for implementing applications that involve transfer of large amounts of data.

Result

The necessary header files of socket programming was studied.

Exercise 2: Study of Basic Functions of Socket programming

Aim

To study Basic Functions of Socket programming.

s = socket.socket(socket\_family, socket\_type)

1)s.bind()

This method binds address (hostname, port number pair) to socket.

s.bind((HOST, PORT))

Host - hostname or the ip address of host

Port - port number to which host socket is connected

2)s.listen()

This method sets up and start TCP listener.

s.listen(5)

Here, listen tells the socket library that we want it to queue up as many as 5 connect requests.

3)s.accept()

This passively accept TCP client connection, waiting until connection arrives (blocking).

conn, addr = s.accept()

The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

4)s.connect()

This method actively initiates TCP server connection.

s.connect(server\_address)

Server\_address is the address of the server.

5)s.recv()

This method receives TCP message

S.recv(16)

Calling recv(16) on the socket object allows us to receive the message from our server in chunks of 16

6)s.send()

This method transmits TCP message

clientSocket.send(data)

data send to client from server in binary format

7)s.recvfrom()

This method receives UDP message

socket.recvfrom(bufsize)

bufsize - The number of bytes to be read from the UDP socket.

8)s.sendto()

This method transmits UDP message

sendto(bytes, address)

bytes - The data to be sent in bytes format.

address -  A tuple consisting of IP address and port number.

9)s.close()

This method closes socket

10)socket.gethostname()

Returns the hostname.

Result

The functions of socket programming was studying.

Exercise 3 Simple Tcp/IP Client Server communication

Aim

To write a code for client server communication using socket programming.

TCP server

import socket

s = socket.socket()

s.bind(('localhost', 9999))

s.listen(1)

print('Server is listening')

c, addr = s.accept()

print('Connected with ', addr)

d = c.recv(1024)

while (True):

data = input("Enter message:")

c.sendall(data.encode())

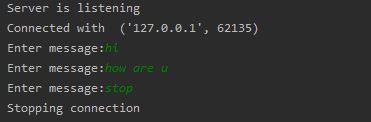
if(data=="stop"):

print("Stopping connection")

break

s.close()

Server Output



TCP client

import socket

c = socket.socket()

c.connect(('localhost',9999))

print('Client waiting for connection')

c.sendall('Client Connected'.encode())

while(True):

msg = c.recv(1024)

if (msg.decode() == 'stop'):

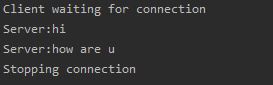
print("Stopping connection")

break

print('Server:'+msg.decode())

c.close()

Client Output



UDP server

import socket

from datetime import date

today = date.today()

s = socket.socket(type=socket.SOCK\_DGRAM)

s.bind(('localhost', 9999))

#s.listen(1)

print('Server is listening')

#c, addr = s.accept()

#print('Connected with ', addr)

d = s.recvfrom(1024)

while (True):

#data = input("Enter message:")

s.sendto(str(today).encode(),d[1])

command = s.recvfrom(1024)

if command[0].decode() == "stop":

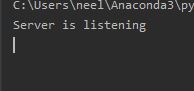
break

elif command[0].decode() == "continue":

continue

s.close()

Server Output



UDP client

import socket

c = socket.socket(type=socket.SOCK\_DGRAM)

c.connect(('localhost',9999))

print('Client waiting for connection')

c.sendto('Client Connected'.encode(),('localhost',9999))

while(True):

msg = c.recvfrom(1024)

if (msg[0].decode() == ''):

break

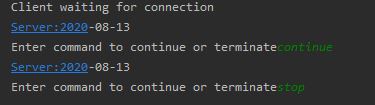
print('Server:'+msg[0].decode())

command = input("Enter command to continue or terminate")

c.sendto(command.encode(),('localhost',9999))

c.close()

Client Output



TCP server

import socket

from datetime import date

today = date.today()

s = socket.socket()

s.bind(('localhost', 9999))

s.listen(1)

print('Server is listening')

c, addr = s.accept()

print('Connected with ', addr)

d = c.recv(1024)

while (True):

#data = input("Enter message:")

c.sendall(str(today).encode())

command = c.recv(1024)

if command.decode() == "stop":

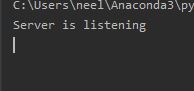
break

elif command.decode() == "continue":

continue

s.close()

Server Output



TCP client

import socket

c = socket.socket()

c.connect(('localhost',9999))

print('Client waiting for connection')

c.sendall('Client Connected'.encode())

while(True):

msg = c.recv(1024)

if (msg.decode() == ''):

break

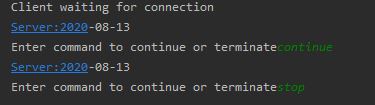
print('Server:'+msg.decode())

command = input("Enter command to continue or terminate")

c.sendall(command.encode())

c.close()

Client output



Result

TCP/IP Client Server communication was performed using socket programming.

Exercise 4 UDP echo client communication

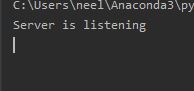
Aim

To write a code for UDP client echo communication using socket programming.

UDP Server

import socket  
  
s = socket.socket(type=socket.SOCK\_DGRAM)  
s.bind(('localhost', 9999))  
print('Server is listening')  
while (True):  
 d=s.recvfrom(1024)  
 if(d=="stop"):  
 print("Stopping connection")  
 break  
 s.sendto(d[0],d[1])  
s.close()

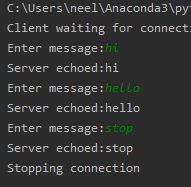
Server Output



UDP Client

import socket  
c = socket.socket(type=socket.SOCK\_DGRAM)  
c.connect(('localhost',9999))  
print('Client waiting for connection')  
  
while(True):  
 data = input("Enter message:")  
 c.sendto(data.encode(),('localhost',9999))  
 msg = c.recvfrom(1024)  
 print("Server echoed:"+msg[0].decode())  
 if (msg[0].decode() == 'stop'):  
 print("Stopping connection")  
 break  
c.close()

Client Output



Result

UDP echo client communication was performed using socket programming.

Exercise 5: Concurrent TCP/IP Day-Time Server

Aim

To create a concurrent TCP/IP day-time server.

Code

server

#5. Conccurent clients program  
import socket  
from \_thread import \*  
import datetime  
s = socket.socket()  
ThreadCount=0  
s.bind((**'localhost'**, 9999))  
print(**'Waitiing for a Connection..'**)  
s.listen(2)  
  
def threaded\_client(connection):  
 global ThreadCount  
 try:  
 while True:  
  
 data = connection.recv(1024).decode()  
 msg =str(datetime.datetime.now())  
 connection.sendall(msg.encode())  
  
 if data==**"stop"**:  
 ThreadCount-=1  
 print(**"Threat Number:"**+str(ThreadCount))  
  
 if ThreadCount==0:  
  
 s.close()  
  
 except:  
 print(**"Closing Server"**)  
 connection.close()  
  
  
try:  
 while True:  
 c, addr = s.accept()  
 print(**'Connected to: Client'**,ThreadCount+1)  
 start\_new\_thread(threaded\_client, (c, ))  
 ThreadCount += 1  
  
 print(**'Thread Number: '** + str(ThreadCount))  
except:  
 s.close()

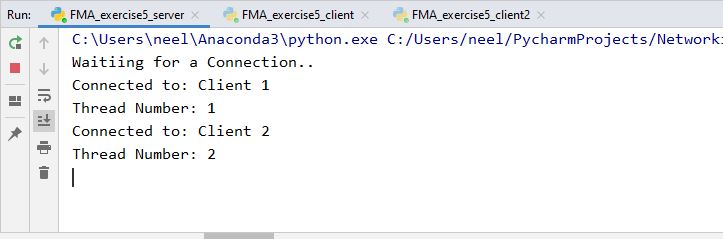
Client-1

import socket  
  
c = socket.socket()  
  
print(**'Waiting for connection'**)  
  
c.connect((**'localhost'**, 9999))  
  
while True:  
 data = input(**'Say Something: '**)  
 c.send(data.encode())  
 if data==**"stop"**:  
 break  
  
 msg = c.recv(1024)  
 print(msg.decode())  
  
c.close()

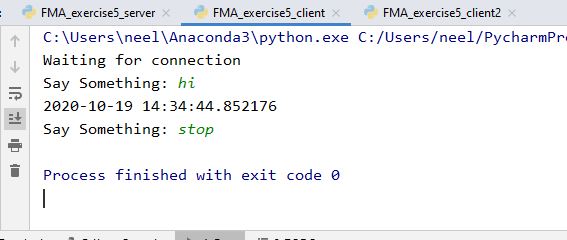
Client-2

import socket  
  
c = socket.socket()  
  
print(**'Waiting for connection'**)  
  
c.connect((**'localhost'**, 9999))  
  
while True:  
 data = input(**'Say something:'**)  
  
 c.send(data.encode())  
 if data == **"stop"**:  
 break  
 msg = c.recv(1024)  
 print(**"server date:"**,msg.decode())  
  
c.close()

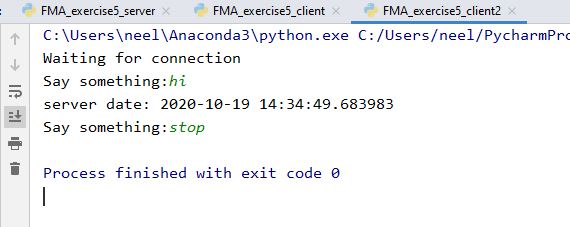
Output



Client-1



Client 2



Result

Concurrent TCP/IP day-time server was created.

Exercise 6: Half Duplex Chat Using TCP/IP

Aim

To create Half Duplex Chat Using TCP/IP.

Code

Server

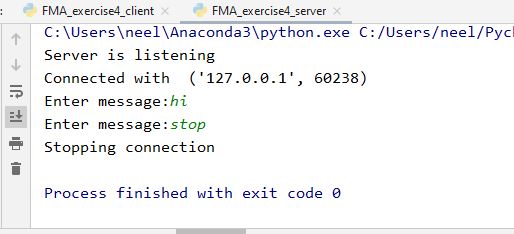
#6. Half Duplex chat  
import socket  
  
s = socket.socket()  
s.bind((**'localhost'**, 9999))  
  
s.listen(1)  
print(**'Server is listening'**)  
  
c, addr = s.accept()  
print(**'Connected with '**, addr)  
d = c.recv(1024)  
while (True):  
 data = input(**"Enter message:"**)  
 c.sendall(data.encode())  
 if(data==**"stop"**):  
 print(**"Stopping connection"**)  
 break  
 msg = c.recv(1024)  
s.close()

Client

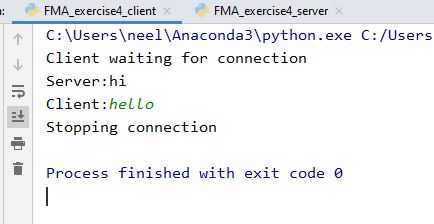
import socket  
c = socket.socket()  
c.connect((**'localhost'**,9999))  
print(**'Client waiting for connection'**)  
c.sendall(**'Client Connected'**.encode())  
while(True):  
 msg = c.recv(1024)  
 if (msg.decode() == **'stop'**):  
 print(**"Stopping connection"**)  
 break  
 print(**'Server:'**+msg.decode())  
 data = input(**"Client:"**)  
 c.sendall(data.encode())  
c.close()

Output

Server



Client



Result

Half Duplex Chat was created Using TCP/IP.

Exercise 7: Full Duplex Chat Using TCP/IP

Aim

To create Full Duplex Chat Using TCP/IP.

Code

Server

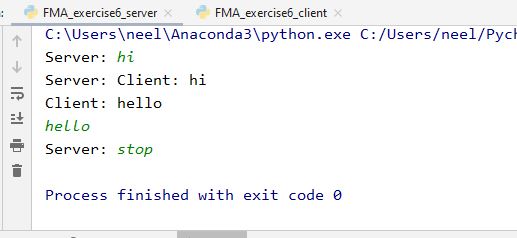
#7. Full Duplex chat  
import socket  
from threading import \*  
from \_thread import \*  
  
s = socket.socket()  
s.bind((**'localhost'**,9999))  
s.listen(1)  
d=0  
class fullDuplex(Thread):  
 def \_\_init\_\_(self,conn):  
 global d  
 Thread.\_\_init\_\_(self)  
 self.conn = conn  
 self.d=0  
 def run(self):  
 name = current\_thread().getName()  
 while True:  
 try:  
 if name == **"Sender"**:  
 data = input(**"Server: "**)  
  
 if data==**"stop"**:  
 self.conn.close()  
 self.conn.send(data.encode())  
 elif name == **"Reciever"**:  
 data = self.conn.recv(1024)  
  
 if data.decode()==**"stop"**:  
 self.conn.close()  
 print(**"Client:"**,data.decode())  
 except:  
 break  
  
  
c,addr = s.accept()  
sender = fullDuplex(c)  
sender.setName(**"Sender"**)  
  
reciever = fullDuplex(c)  
reciever.setName(**"Reciever"**)  
  
sender.start()  
reciever.start()

Client

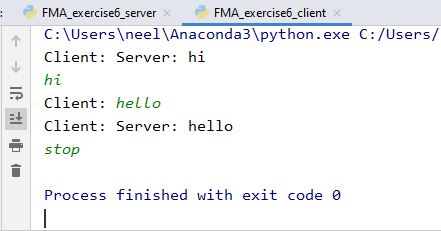
import socket  
from threading import \*  
from \_thread import \*  
  
c = socket.socket()  
c.connect((**"localhost"**,9999))  
  
class fullDuplex(Thread):  
 def \_\_init\_\_(self,conn):  
 Thread.\_\_init\_\_(self)  
 self.conn = conn  
 self.d=0  
 def run(self):  
 name = current\_thread().getName()  
 while True:  
 try:  
 if name == **"Sender"**:  
 data = input(**"Client: "**)  
  
 if data==**"stop"**:  
 self.conn.close()  
 self.conn.send(data.encode())  
 elif name == **"Reciever"**:  
 data = self.conn.recv(1024)  
  
 if data.decode()==**"stop"**:  
 self.conn.close()  
 print(**"Server:"**, data.decode())  
 except:  
 break  
  
sender = fullDuplex(c)  
sender.setName(**"Sender"**)  
  
reciever = fullDuplex(c)  
reciever.setName(**"Reciever"**)  
  
sender.start()  
reciever.start()

Output

Server



Client



Result

Full Duplex Chat was created Using TCP/IP.

Exercise 8: Implementation of File Transfer Protocol

Aim

To create a File transfer server.

Code

Server

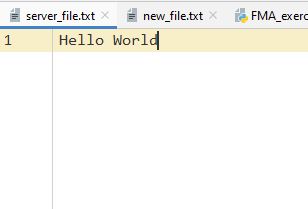
#8. File Transfer  
import socket  
import os  
  
s = socket.socket()  
s.bind((**'localhost'**, 9999))  
  
s.listen(1)  
print(**'Server is listening'**)  
  
c, addr = s.accept()  
print(**'Connected with '**, addr)  
  
with open(**"open.txt"**) as f:  
   
 while True:  
 data = f.read(1024)  
 if not data:  
 print(**"Stopping Connection"**)  
 break  
 c.sendall(data.encode())  
  
f.close()  
s.close()

Client

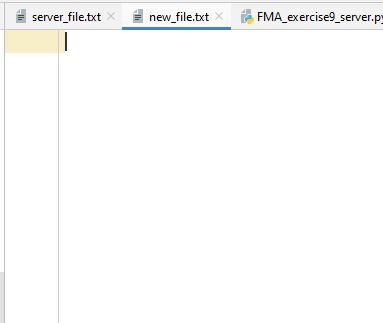
import socket  
c = socket.socket()  
c.connect((**'localhost'**,9999))  
print(**'Client waiting for connection'**)  
f = open(**"new\_file.txt"**,**"w"**)  
while True:  
 msg = c.recv(1024).decode()  
  
 f.write(msg)  
 if msg==**''**:  
 break  
  
f.close()  
f = open(**"new\_file.txt"**,**"r"**)  
print(f.read())  
f.close()  
c.close()

Output

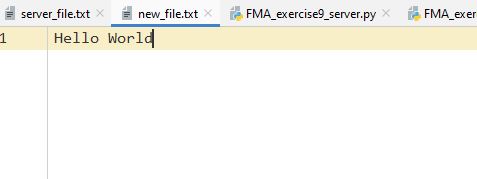
Server



New file before running program



New file after running program



Result

File transfer server was created.

Exercise 9: Remote Command Execution Using UDP

Aim

To perform Remote Command Execution Using UDP.

Code

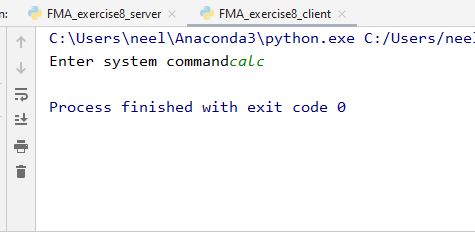
Server

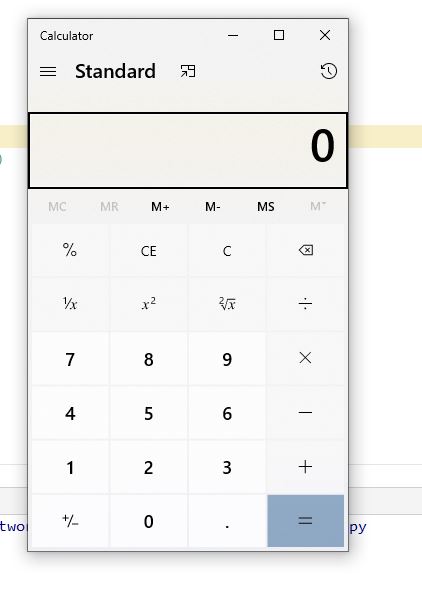
#9. UDP Remote Command Execution  
import socket  
import os  
s = socket.socket(type=socket.SOCK\_DGRAM)  
s.bind((**'localhost'**, 9999))  
  
d = s.recvfrom(1024)  
os.system(**'cmd /c "{}"'**.format(d[0].decode()))  
  
s.close()

Client

import socket  
  
c = socket.socket(type=socket.SOCK\_DGRAM)  
  
c.connect((**'localhost'**,9999))  
  
command = input(**"Enter system command"**)  
c.sendto(command.encode(),(**'localhost'**,9999))  
  
c.close()

Output





Result

Remote Command Execution Using UDP was done.

Exercise 10: ARP implementation Using UDP

Aim

To implement ARP using UDP.

Code

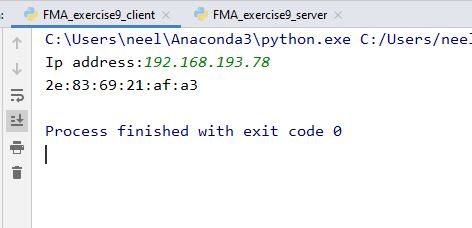
Server

#10. ARP implementation using UDP  
  
import socket  
import getmac  
s = socket.socket(type=socket.SOCK\_DGRAM)  
s.bind((**'localhost'**,9999))  
  
  
data = s.recvfrom(1024)  
ip\_address=data[0].decode()  
destination=data[1]  
  
mac = getmac.get\_mac\_address(ip=ip\_address)  
  
if mac is not None:  
 s.sendto(mac.encode(),destination)  
else:  
 s.sendto(**"Not found"**.encode(),destination)  
  
s.close()

Client

import socket  
  
c = socket.socket(type=socket.SOCK\_DGRAM)  
  
c.connect((**'localhost'**,9999))  
  
ip\_address = input(**"Ip address:"**)  
c.sendto(ip\_address.encode(),(**'localhost'**,9999))  
mac\_address=c.recvfrom(1024)  
print(mac\_address[0].decode())  
c.close()

Output



Result

ARP was implemented using UDP.

Exercise 11: Study of ipv6 addressing and sub-netting

Aim

To study ipv6 addressing and sub-netting.

Ipv6 addressing and sub-netting

Internet Protocol version 6 (IPv6) is the most recent version of the [Internet Protocol](https://en.wikipedia.org/wiki/Internet_Protocol" \o "Internet Protocol) (IP), the [communications protocol](https://en.wikipedia.org/wiki/Communication_protocol" \o "Communication protocol) that provides an identification and location system for computers on networks and routes traffic across the [Internet](https://en.wikipedia.org/wiki/Internet" \o "Internet).

An IPv6 packet has two parts: a [header](https://en.wikipedia.org/wiki/Header_(computing)" \o "Header (computing)) and [payload](https://en.wikipedia.org/wiki/Payload_(computing)" \o "Payload (computing)).

The header consists of a fixed portion with minimal functionality required for all packets and may be followed by optional extensions to implement special features.

The fixed header occupies the first 40 [octets](https://en.wikipedia.org/wiki/Octet_(computing)" \o "Octet (computing)) (320 bits) of the IPv6 packet. It contains the source and destination addresses, traffic classification options, a hop counter, and the type of the optional extension or payload which follows the header. This *Next Header* field tells the receiver how to interpret the data which follows the header. If the packet contains options, this field contains the option type of the next option. The "Next Header" field of the last option points to the upper-layer protocol that is carried in the packet's [payload](https://en.wikipedia.org/wiki/Payload_(computing)" \o "Payload (computing)).

Extension headers carry options that are used for special treatment of a packet in the network, e.g., for routing, fragmentation, and for security using the [IPsec](https://en.wikipedia.org/wiki/IPsec" \o "IPsec) framework.

Without special options, a payload must be less than 64kB. With a Jumbo Payload option (in a *Hop-By-Hop Options* extension header), the payload must be less than 4 GB.

Unlike with IPv4, routers never fragment a packet. Hosts are expected to use [Path MTU Discovery](https://en.wikipedia.org/wiki/Path_MTU_Discovery" \o "Path MTU Discovery) to make their packets small enough to reach the destination without needing to be fragmented. See [IPv6 packet fragmentation](https://en.wikipedia.org/wiki/IPv6_packet" \l "Fragmentation" \o "IPv6 packet).

The main advantage of IPv6 over IPv4 is its larger address space. The size of an IPv6 address is 128 bits, compared to 32 bits in IPv4.[[2]](https://en.wikipedia.org/wiki/IPv6" \l "cite_note-rfc2460-2) The address space therefore has 2128 = 340,282,366,920,938,463,463,374,607,431,768,211,456 addresses (approximately 3.4×1038). Some blocks of this space and some specific addresses are [reserved for special uses](https://en.wikipedia.org/wiki/Reserved_IP_addresses" \o "Reserved IP addresses).

All IPv6 addresses are 128 bits long, written as 8 sections of 16 bits each. They are expressed in hexadecimal representation, so the sections range from 0 to FFFF. Sections are delimited by colons, and leading zeroes in each section may be omitted. If two or more consecutive sections have all zeroes, they can be collapsed to a double colon.

An IPv6 address prefix is a combination of an IPv6 prefix (address) and a prefix length. The prefix takes the form *ipv6-prefix*/*prefix-length* and represents a block of address space (or a network). The *ipv6-prefix* variable follows general IPv6 addressing rules. The *prefix-length* variable is a decimal value that indicates the number of contiguous, higher-order bits of the address that make up the network portion of the address. For example, 10FA:6604:8136:6502::/64 is a possible IPv6 prefix with zeros compressed. The site prefix of the IPv6 address 10FA:6604:8136:6502::/64 is contained in the left most 64 bits, 10FA:6604:8136:6502.

Result

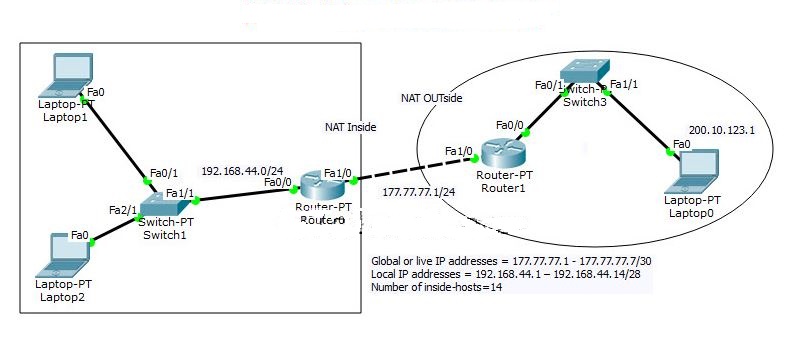
Ipv6 addressing and sub-netting was studied.

Exercise: 12 Implementation of NAT

Aim

To configure NAT in cisco packet tracer.

Network Diagram



Task 1: Configure IP addresses and setting a static default route.

Access the CLI and specify the following IP address configuration on both routers

R1 Config:

Router1(config)#int fa1/0

Router1(config-if)#ip address 177.77.77.10 255.255.255.0

Router1(config-if)#no shu

Router1(config)#ip route 192.168.44.0 255.255.255.0 177.77.77.1

R0 config:

Router0(config)#int fa1/0

Router0(config-if)#ip address 177.77.77.1 255.255.255.0

Router0(config-if)#no shut

Router0(config)#int fa0/0

Router0(config-if)#ip address 192.168.44.1 255.255.255.0

Router0(config-if)#no shut

Router0(config)#ip route 200.10.123.0 255.255.255.0 177.77.77.10

Router0(config)#exit

Task 2: Configure NAT on Router0

Step 1: Access the CLI on the Router0

Step 2: Configure a standard access list using an access list id of 1 and permit any device on subnet 192.168.44.0 /24

Router0(config)#access-list 1 permit 192.168.44.0 0.0.0.255

Here access list is used to identify which source IPs are going to be translated using NAT, and this example allows any device from the 192.168.44.0 network.

Step 3: Create a dynamic NAT IP-address pool, this will hold a list of inside global addresses.

Name of Pool: NAT-POOL

Starting IP address 192.168.44.1

Ending IP address 192.168.44.14

Network mask 255.255.255.240

Complete this configuration with following commands:

Router0(config)#ip nat pool NAT-POOL 177.77.77.1 177.77.77.7 netmask 255.255.255.240

Router0(config)#ip nat inside source list 1 pool NAT-POOL

For network address translations, we are requires to identify at least two interfaces to be our inside and outside.

Interface fa0/0 will be our inside interface

Interface fa1/0 our outside interface

You can use the following commands for configuring inside and outside interfaces.

R0(config)#int fa0/0

R0(config-if)#ip nat inside

R0(config-if)#int fa1/0

R0(config-if)#ip nat outside

Result

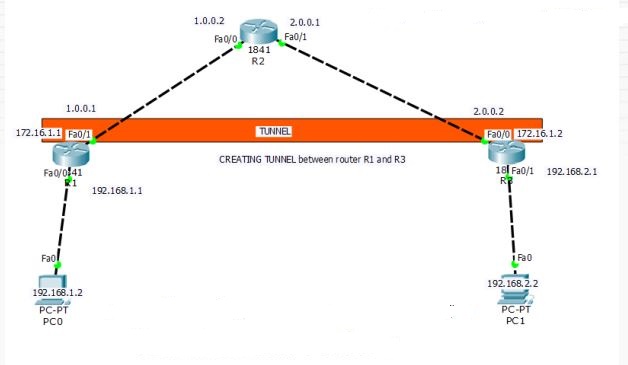
NAT was configured.

Exercise 13: Implementation of VPN

Aim

To configure VPN in routers in cisco packet tracer.

Network Diagram



Router 1

Router>enable

Router#config t

Router(config)#host r1

r1(config)#int fa0/0

r1(config-if)#ip add 192.168.1.1 255.255.255.0

r1(config-if)#no shut

r1(config-if)#exit

r1(config)#int fa0/1

r1(config-if)#ip address 1.0.0.1 255.0.0.0

r1(config-if)#no shut

r1>enable

r1#config t

Enter configuration commands, one per line. End with CNTL/Z.

r1(config)#ip route 0.0.0.0 0.0.0.0 1.0.0.2

r1(config)#

r1#config t

r1(config)#interface tunnel 10

r1(config-if)#ip address 172.16.1.1 255.255.0.0

r1(config-if)#tunnel source fa0/1

r1(config-if)#tunnel destination 2.0.0.2

r1(config-if)#no shut

r1(config)#ip route 192.168.2.0 255.255.255.0 172.16.1.2

Router 2

Router>enable

Router#config t

Router(config)#host r2

r2(config)#int fa0/0

r2(config-if)#ip add 1.0.0.2 255.0.0.0

r2(config-if)#no shut

r2(config-if)#exit

r2(config)#int fa0/1

r2(config-if)#ip add 2.0.0.1 255.0.0.0

r2(config-if)#no shut

Router 3

Router>enable

Router#config t

Router(config)#host r3

r3(config)#int fa0/0

r3(config-if)#ip add 2.0.0.2 255.0.0.0

r3(config-if)#no shut

r3(config-if)#exit

r3(config)#int fa0/1

r3(config-if)#ip add 192.168.2.1 255.255.255.0

r3(config-if)#no shut

r3>enable

r3#config t

Enter configuration commands, one per line. End with CNTL/Z.

r3(config)#ip route 0.0.0.0 0.0.0.0 2.0.0.1

r3(config)#

r3#config t

r3(config)#interface tunnel 100

r3(config-if)#ip address 172.16.1.2 255.255.0.0

r3(config-if)#tunnel source fa0/0

r3(config-if)#tunnel destination 1.0.0.1

r3(config-if)#no shut

r3(config)#ip route 192.168.1.0 255.255.255.0 172.16.1.1

Result

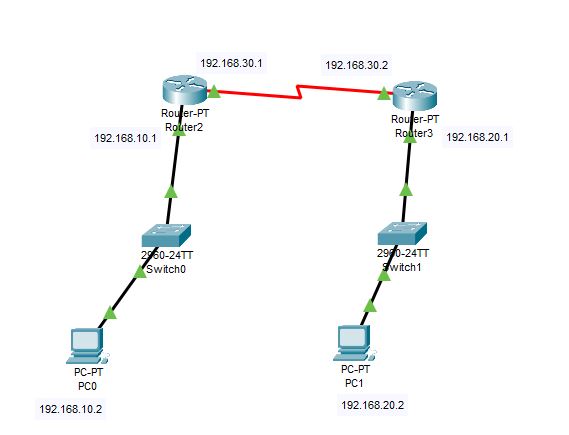
VPN in router was configured.

Exercise 14: Communication Using HDLC

Aim

Enabling router to router communication using HDLC.

Network Diagram



Code

Router 2:

Router2>en

Router2#conf t

Router2(config)#int fa0/0

Router2(config-if)#ip address 192.168.10.1 255.255.255.0

Router2(config-if)#no shut

Router2(config-if)#exit

Router2(config)#int se2/0

Router2(config-if)#encapsulation hdlc

Router2(config-if)#ip address 192.168.30.1 255.255.255.0

Router2(config-if)#no shut

Router2(config-if)#exit

Router2(config)#ip route 0.0.0.0 0.0.0.0 se2/0

Router2(config)#end

Router 3:

Router 3:

Router3>en

Router3#conf t

Router3(config)#int fa0/0

Router3(config-if)#ip address 192.168.20.1 255.255.255.0

Router3(config-if)#no shut

Router3(config-if)#exit

Router3(config)#int se2/0

Router3(config-if)#encapsulation hdlc

Router3(config-if)#ip address 192.168.30.2 255.255.255.0

Router3(config-if)#no shut

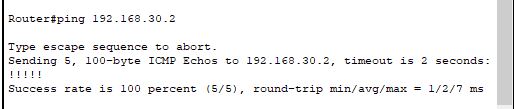
Router3(config-if)#exit

Router3(config)#ip route 0.0.0.0 0.0.0.0 se2/0

Router3(config)#end

Result

Exp 14 hdlc



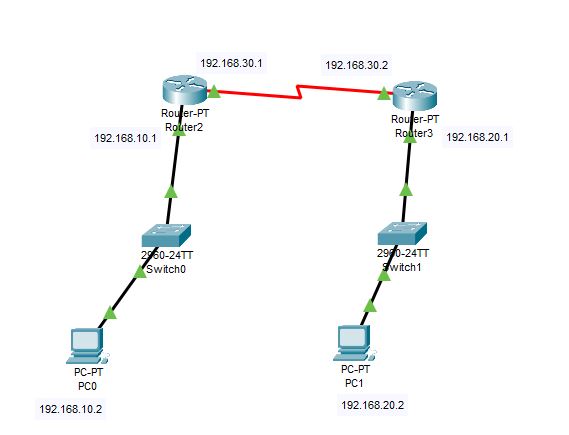
Router to router communication was successfully done using HDLC and tested using ping command.

Exercise 15: Communication Using PPP

Aim

Enabling router to router communication using PPP.

Network Diagram



Code

Router 2:

Router2>en

Router2#conf t

Router2(config)#int fa0/0

Router2(config-if)#ip address 192.168.10.1 255.255.255.0

Router2(config-if)#no shut

Router2(config-if)#exit

Router2(config)#int se2/0

Router2(config-if)#encapsulation ppp

Router2(config-if)#ip address 192.168.30.1 255.255.255.0

Router2(config-if)#no shut

Router2(config-if)#exit

Router2(config)#ip route 0.0.0.0 0.0.0.0 se2/0

Router2(config)#end

Router 3:

Router 3:

Router3>en

Router3#conf t

Router3(config)#int fa0/0

Router3(config-if)#ip address 192.168.20.1 255.255.255.0

Router3(config-if)#no shut

Router3(config-if)#exit

Router3(config)#int se2/0

Router3(config-if)#encapsulation ppp

Router3(config-if)#ip address 192.168.30.2 255.255.255.0

Router3(config-if)#no shut

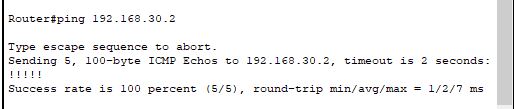
Router3(config-if)#exit

Router3(config)#ip route 0.0.0.0 0.0.0.0 se2/0

Router3(config)#end

Result

Exp 15 PPP



Router to router communication was successfully done using PPP and tested using ping command.