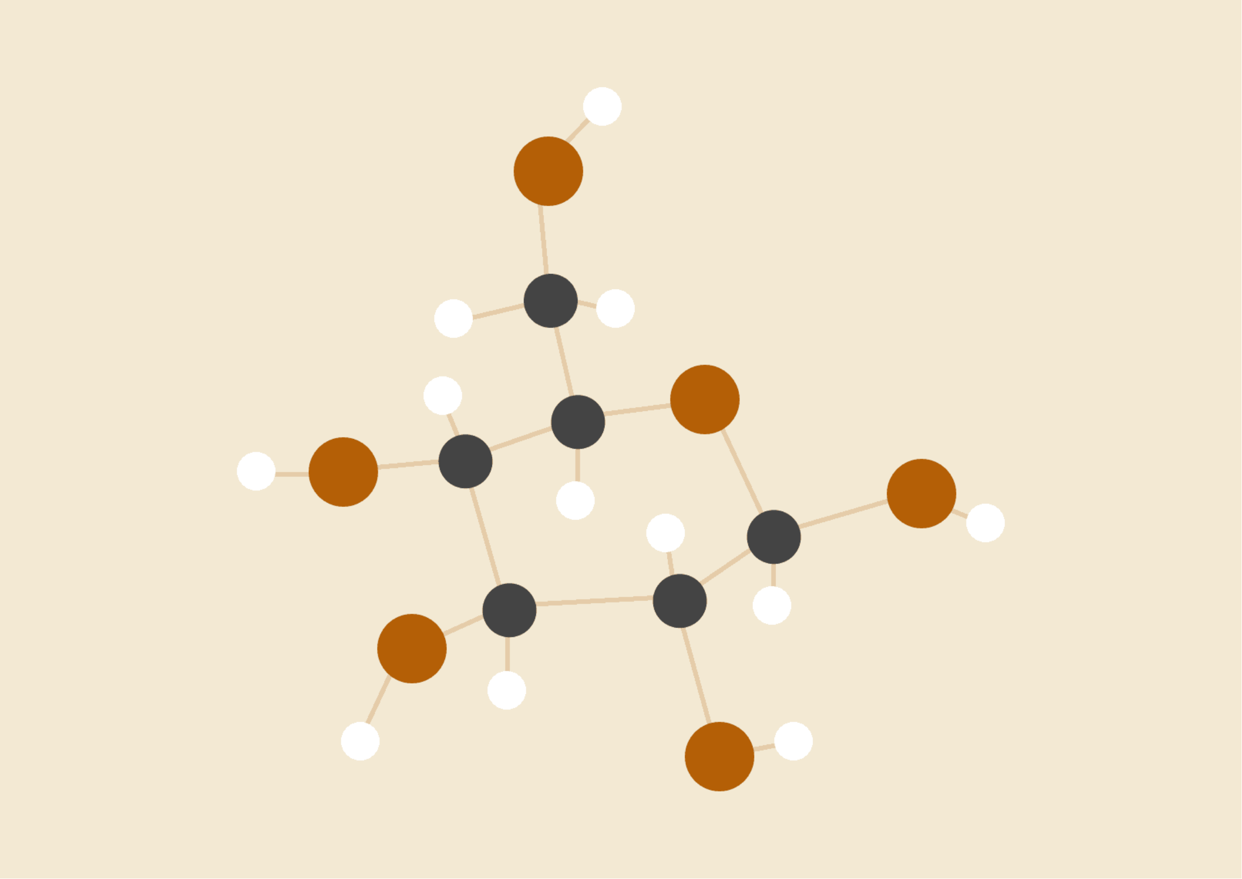
NETWORKING LAB REPORT

CLASS *BCSE III* ***SEM*** *FIFTH* ***YEAR*** *2021*



NAME Neeladri Pal

ROLL 001910501015

GROUP A1

ASSIGNMENT - 7

PROBLEM STATEMENT

Implement any two protocols using TCP/UDP socket as available.

1. BOOTP
2. FTP
3. DHCP
4. BGP
5. RIP

DHCP

The Dynamic Host Configuration Protocol (DHCP) has been devised to provide static and dynamic address allocation that can be manual or automatic.

A DHCP server has a database that statically binds physical addresses to IP addresses.

DHCP has a second database with a pool of available IP addresses. This second database makes DHCP dynamic. When a DHCP client requests a temporary IP address, the DHCP server goes to the pool of available (unused) IP addresses and assigns an IP address for a negotiable period of time.

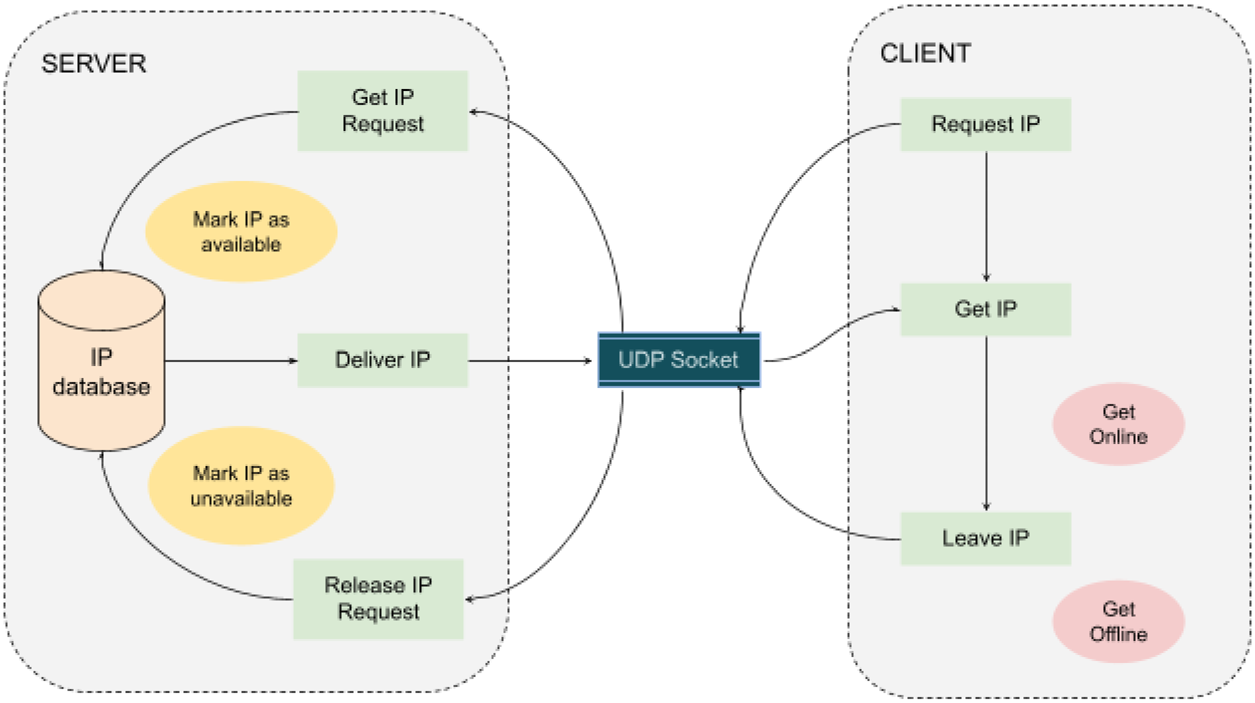
When a DHCP client sends a request to a DHCP server, the server first checks its static database. If an entry with the requested physical address exists in the static data- base, the permanent IP address of the client is returned. On the other hand, if the entry does not exist in the static database, the server selects an IP address from the available pool, assigns the address to the client, and adds the entry to the dynamic database.

The dynamic aspect of DHCP is needed when a host moves from network to net- work or is connected and disconnected from a network (as is a subscriber to a service provider). DHCP provides temporary IP addresses for a limited time.

The addresses assigned from the pool are temporary addresses. The DHCP server issues a lease for a specific time. When the lease expires, the client must either stop using the IP address or renew the lease. The server has the option to agree or disagree with the renewal. If the server disagrees, the client stops using the address.

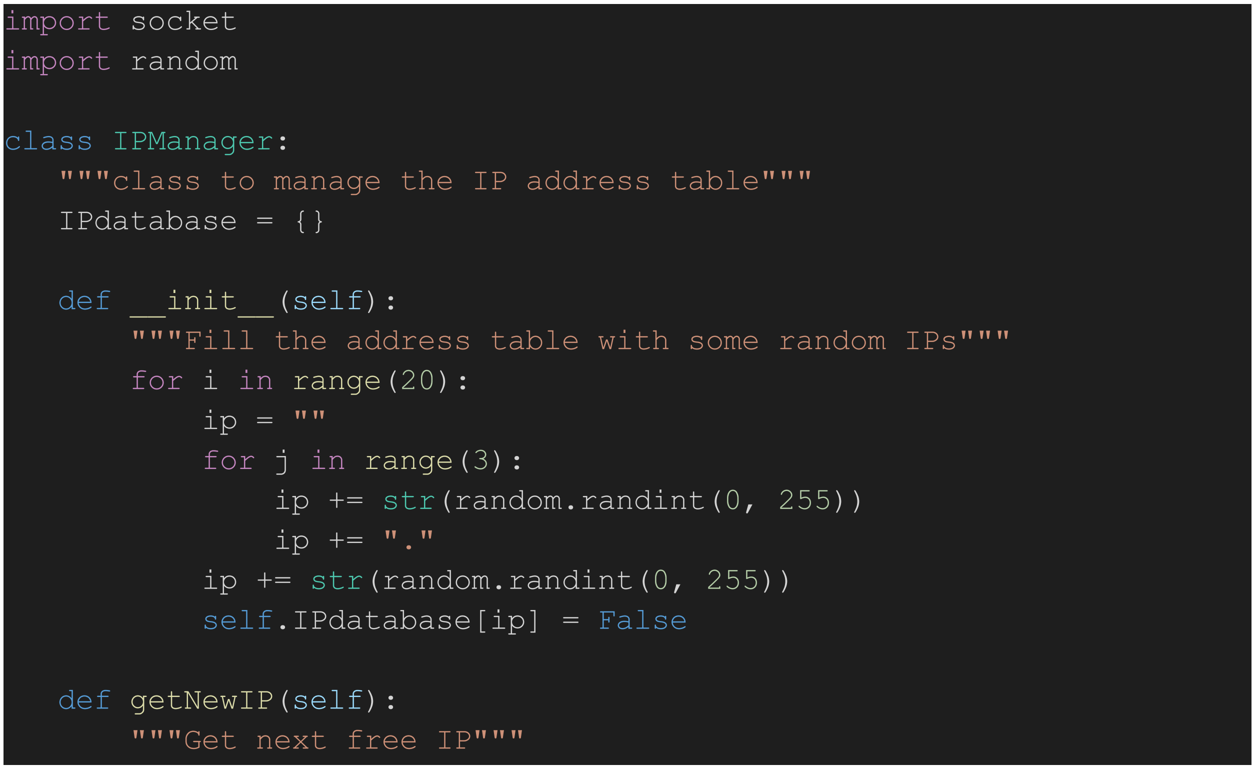
DHCP allows both manual and automatic configurations. Static addresses are created manually~ dynamic addresses are created automatically.

DESIGN

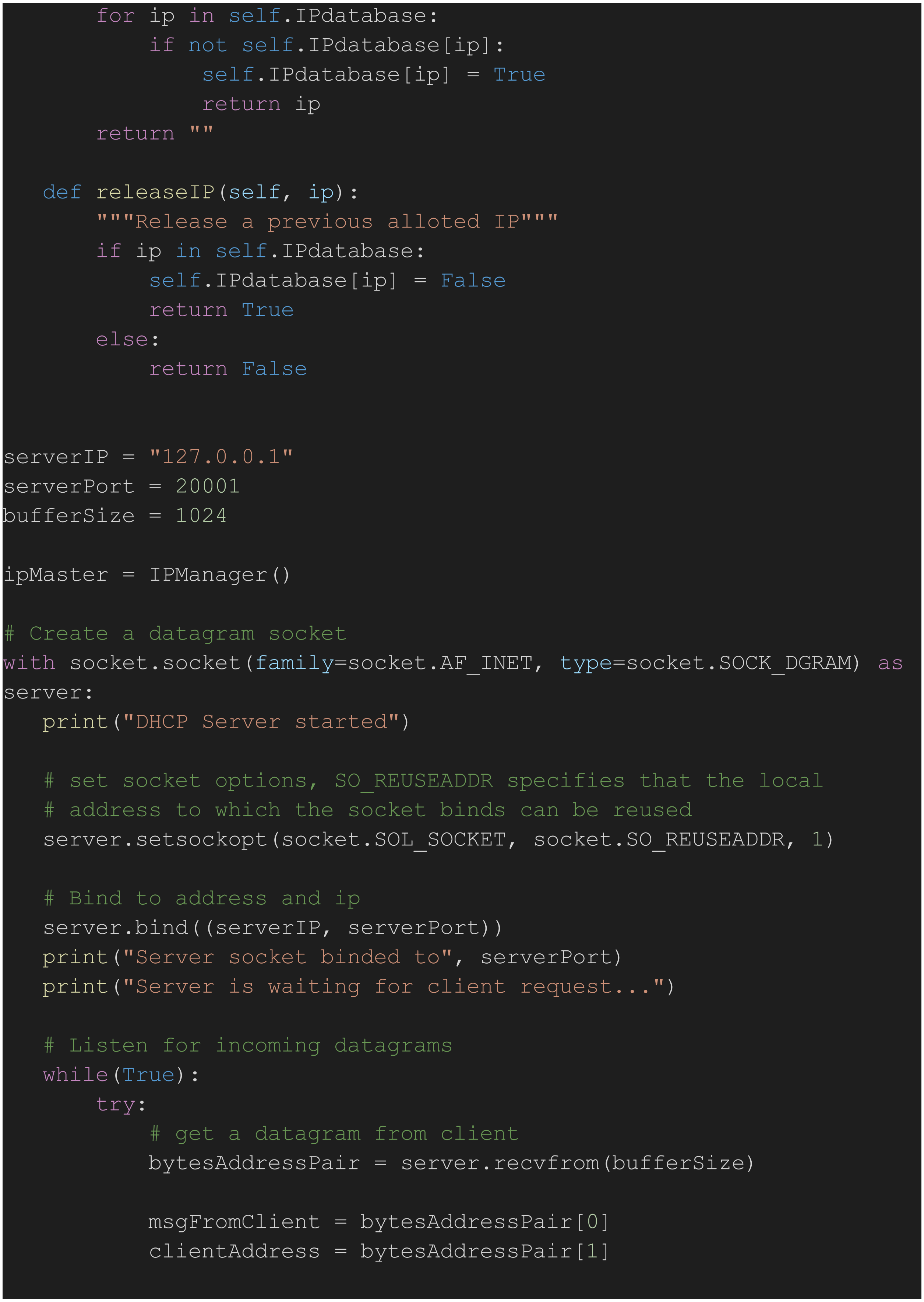


IMPLEMENTATION

DHCP Server →



```
import socket
import random

class IPManager:
   """class to manage the IP address table"""
   IPdatabase = {}

   def __init__(self):
       """Fill the address table with some random IPs"""
       for i in range(20):
           ip = ""
           for j in range(3):
               ip += str(random.randint(0, 255))
               ip += "."
           ip += str(random.randint(0, 255))
           self.IPdatabase[ip] = False

   def getNewIP(self):
       """Get next free IP"""
```

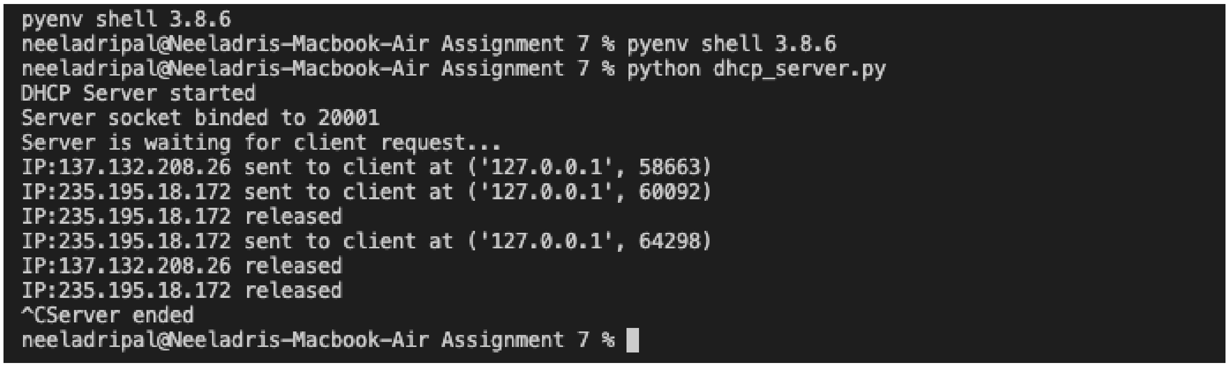


```
        for ip in self.IPdatabase:
            if not self.IPdatabase[ip]:
                self.IPdatabase[ip] = True
                return ip
        return ""

    def releaseIP(self, ip):
        """Release a previous alloted IP"""
        if ip in self.IPdatabase:
            self.IPdatabase[ip] = False
            return True
        else:
            return False


serverIP = "127.0.0.1"
serverPort = 20001
bufferSize = 1024

ipMaster = IPManager()

# Create a datagram socket
with socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM) as
server:
    print("DHCP Server started")

    # set socket options, SO_REUSEADDR specifies that the local
    # address to which the socket binds can be reused
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # Bind to address and ip
    server.bind((serverIP, serverPort))
    print("Server socket binded to", serverPort)
    print("Server is waiting for client request...")

    # Listen for incoming datagrams
    while(True):
        try:
            # get a datagram from client
            bytesAddressPair = server.recvfrom(bufferSize)

            msgFromClient = bytesAddressPair[0]
            clientAddress = bytesAddressPair[1]
            clientIP, clientPort = clientAddress

            if msgFromClient:
                msgFromClient = msgFromClient.decode()

                # if the client requests a new IP
                if msgFromClient == "Give me IP":
                    # get the next available ip
                    newIP = ipMaster.getNewIP()
                    if newIP == "":
                        msgFromServer = "No IPs left"
                    else:
                        msgFromServer = "IP:" + newIP
                    # send it to client
                    server.sendto(str.encode(msgFromServer),
clientAddress)
                    print(msgFromServer, "sent to client at",
clientAddress)

                # if the client wants to release the ip
                else:
                    # receive the ip to be released
                    ipToRelease = msgFromClient
                    if (ipMaster.releaseIP(ipToRelease)):
                        print("IP:"+ipToRelease+" released")
                    else:
                        print("IP:"+ipToRelease+" not found")

        except:
            break

print("Server ended")
```

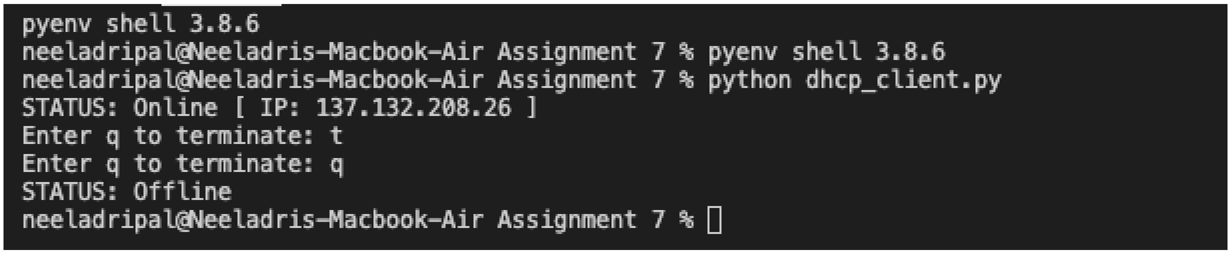
DHCP Client →

```
import socket
import time

serverAddress = ("127.0.0.1", 20001)
bufferSize = 1024

# create a UDP socket at client side
with socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM) as
client:
   # request IP from DHCP server
   msgFromClient = "Give me IP"
   client.sendto(str.encode(msgFromClient), serverAddress)

   # get the IP back from server
   bytesAddressPair = client.recvfrom(bufferSize)

   if bytesAddressPair:
       msgFromServer = bytesAddressPair[0].decode()

       # if an IP is available
       if msgFromServer != "No IPs found":
           ip = msgFromServer.split(":")[1]
           print("STATUS: Online [ IP:", ip, "]")
           while True:
               time.sleep(2)
               chr = input("Enter q to terminate: ")
               if chr == 'q' or chr == 'Q':
                   # while terminating, send back the IP to the
server
                   client.sendto(str.encode(ip), serverAddress)
                   break

print("STATUS: Offline")
```

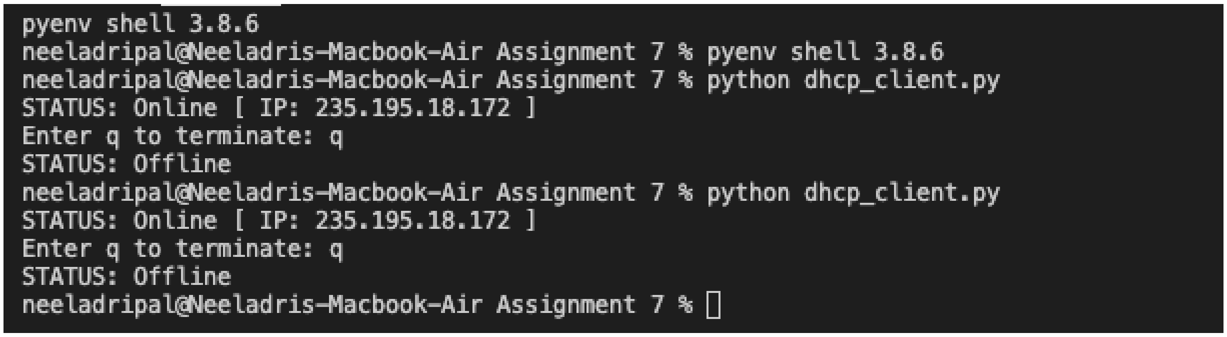
OUTPUT



```
pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % python dhcp_server.py
DHCP Server started
Server socket binded to 20001
Server is waiting for client request...
IP:137.132.208.26 sent to client at ('127.0.0.1', 58663)
IP:235.195.18.172 sent to client at ('127.0.0.1', 60092)
IP:235.195.18.172 released
IP:235.195.18.172 sent to client at ('127.0.0.1', 64298)
IP:137.132.208.26 released
IP:235.195.18.172 released
^CServer ended
neeladripal@Neeladris-Macbook-Air Assignment 7 % 
```



```
pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % python dhcp_client.py
STATUS: Online [ IP: 137.132.208.26 ]
Enter q to terminate: t
Enter q to terminate: q
STATUS: Offline
neeladripal@Neeladris-Macbook-Air Assignment 7 %
```



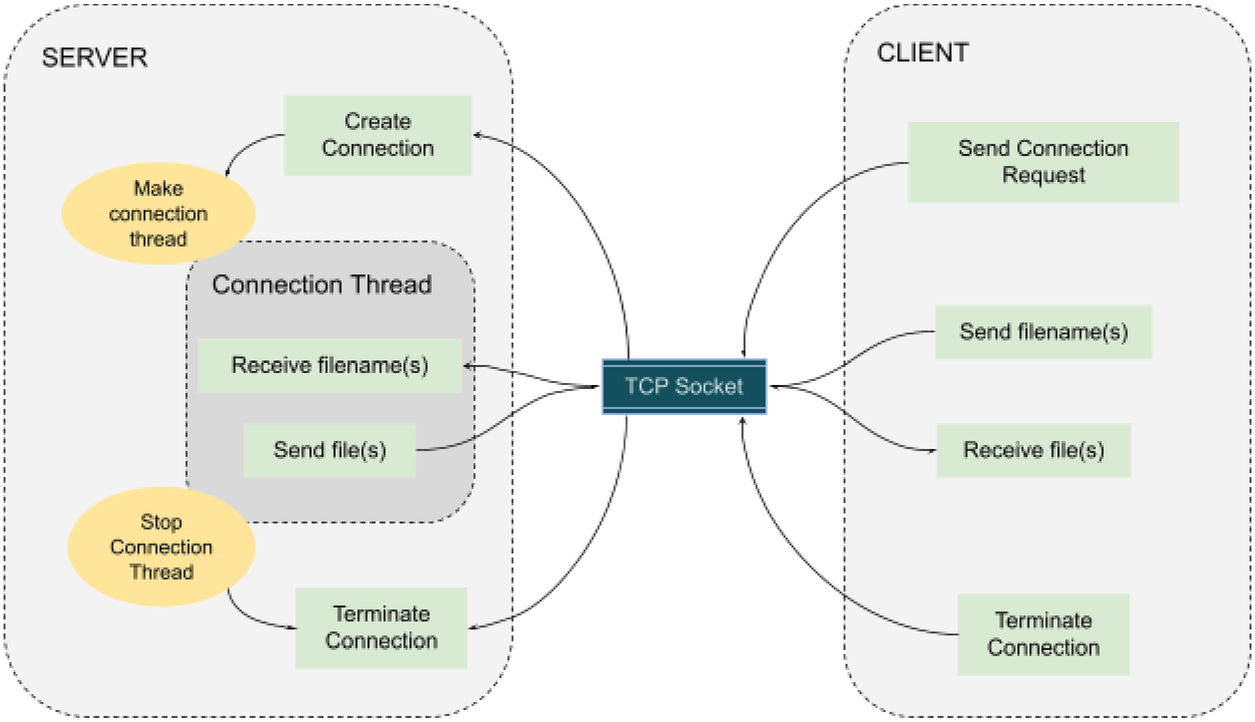
```
pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % python dhcp_client.py
STATUS: Online [ IP: 235.195.18.172 ]
Enter q to terminate: q
STATUS: Offline
neeladripal@Neeladris-Macbook-Air Assignment 7 % python dhcp_client.py
STATUS: Online [ IP: 235.195.18.172 ]
Enter q to terminate: q
STATUS: Offline
neeladripal@Neeladris-Macbook-Air Assignment 7 %
```

FTP

File Transfer Protocol (FTP) is the standard mechanism provided by TCP/IP for copying a file from one host to another. Although transferring files from one system to another seems simple and straightforward, some problems must be dealt with first. For example, two systems may use different file name conventions. Two systems may have different ways to represent text and data. Two systems may have different directory structures. All these problems have been solved by FTP in a very simple and elegant approach.

FTP establishes two connections between the hosts. One connection is used for data transfer, the other for control information (commands and responses). Separation of commands and data transfer makes FTP more efficient. The control connection uses very simple rules of communication. We need to transfer only a line of command or a line of response at a time. The data connection, on the other hand, needs more complex rules due to the variety of data types transferred. However, the difference in complexity is at the FTP level, not TCP. For TCP, both connections are treated the same.

DESIGN

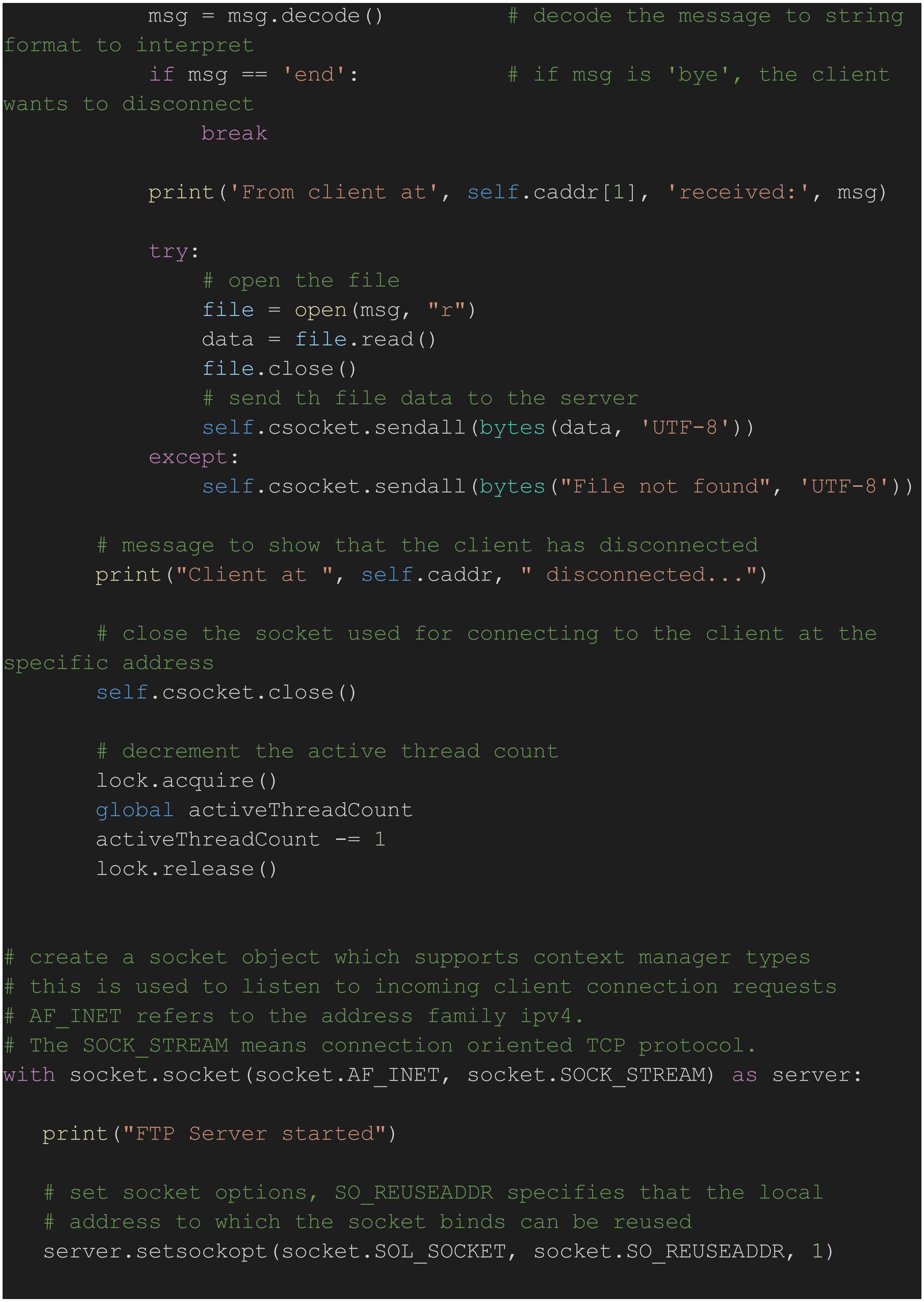


IMPLEMENTATION

FTP Server →

```
# import the necessary modules
import socket
from threading import Thread, Lock

lock = Lock()            # to regulate the limit on number of active
threads at a time

activeThreadCount = 0            # number of threads currently active
# maximum number of clients that can be served simultaneously
MAXIMUM_PROCESSING = 3
MAXIMUM_WAITING = 2              # maximum number of clients waiting
to connect to server

# specify a host network interface, here we use loopback interface
whose IPv4 address is 127.0.0.1
# If a hostname is used in the host portion of IPv4/v6 socket
address, the program may show a
# non-deterministic behavior, as Python uses the first address
returned from the DNS resolution
HOST = '127.0.0.1'
```

```
# reserve a port on local machine for listening to incoming client
requests
PORT = 12345


# new thread for a new connection
class ConnectionThread (Thread):
   def __init__(self, clientAddress, clientSocket) -> None:
       Thread.__init__(self)

       # increment the active thread count
       # lock must be acquired to prevent possible race conditions
       lock.acquire()
       global activeThreadCount
       activeThreadCount += 1
       lock.release()

       # save the properties of the incoming client communication
       self.csocket = clientSocket
       self.caddr = clientAddress
       print('Got new connection from', clientAddress)

   # override the __init__ method to specify the code that the thread
would run on
   def run(self) -> None:

       # send a message to indicate that the server is ready to
respond
       self.csocket.send(bytes(
           "You are now connected to server.\n\tSend the name of the
file you want.\n\tSay end to terminate the session.", 'utf-8'))
       while True:
           try:
               # a blocking call to receive message from client in
bytes form
               msg = self.csocket.recv(1024)
           except:
               # in case the client is abruptly terminated
               print('Cannot receive data')

           if not msg:            # if the message contains no data,
it must be due to some error
               break
```



```
           msg = msg.decode()          # decode the message to string
format to interpret
           if msg == 'end':            # if msg is 'bye', the client
wants to disconnect
               break

           print('From client at', self.caddr[1], 'received:', msg)

           try:
               # open the file
               file = open(msg, "r")
               data = file.read()
               file.close()
               # send th file data to the server
               self.csocket.sendall(bytes(data, 'UTF-8'))
           except:
               self.csocket.sendall(bytes("File not found", 'UTF-8'))

       # message to show that the client has disconnected
       print("Client at ", self.caddr, " disconnected...")

       # close the socket used for connecting to the client at the
specific address
       self.csocket.close()

       # decrement the active thread count
       lock.acquire()
       global activeThreadCount
       activeThreadCount -= 1
       lock.release()


# create a socket object which supports context manager types
# this is used to listen to incoming client connection requests
# AF_INET refers to the address family ipv4.
# The SOCK_STREAM means connection oriented TCP protocol.
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:

   print("FTP Server started")

   # set socket options, SO_REUSEADDR specifies that the local
   # address to which the socket binds can be reused
   server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

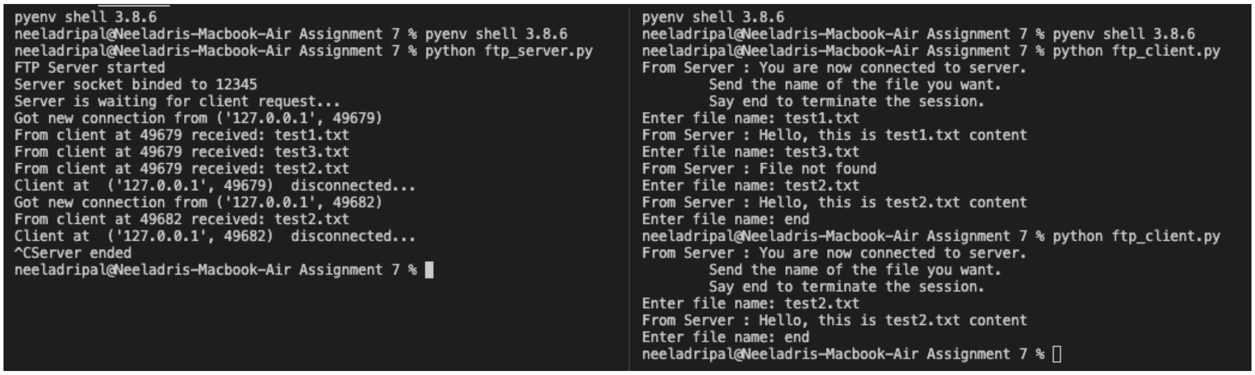
```
    # Next bind to the port
    server.bind((HOST, PORT))
    print("Server socket binded to %s" % (PORT))

    # put the socket into listening mode, Its backlog parameter
    # specifies the number of unaccepted connections that the
    # system will allow before refusing new connections.
    server.listen(MAXIMUM_WAITING)
    print("Server is waiting for client request...")

    # a forever loop until we interrupt it or an error occurs
    while True:
        try:
            # fetch the active thread count
            # lock ensures fetch does not take place while
            # the value is being updated
            lock.acquire()
            n = activeThreadCount
            lock.release()

            if n < MAXIMUM_PROCESSING:
                # Establish connection with client
                # a new socket is returned for connecting to the
client
                # which is different from the listening socket
                conn, addr = server.accept()

                # interaction with client continues in a new thread
                newthread = ConnectionThread(addr, conn)
                newthread.start()
        except:
            break

print("Server ended")
```

FTP Client →

```
# import socket module
import socket

SERVER = "127.0.0.1"        # host interface of the server
PORT = 12345                # port on which the server listens

# create a new socket object
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client:
```

```python
client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# connect to server
client.connect((SERVER, PORT))

# a forever loop until we interrupt it or an error occurs
while True:
    # recieve data from server
    in_data = client.recv(1024)
    print("From Server :", in_data.decode())

    # take message from user
    msg = input('Enter file name: ')

    # send it to server
    client.sendall(bytes(msg, 'UTF-8'))

    # if message is 'end', terminate the connection
    if not msg or msg == 'end':
        break
```

OUTPUT



```
pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % python ftp_server.py
FTP Server started
Server socket binded to 12345
Server is waiting for client request...
Got new connection from ('127.0.0.1', 49679)
From client at 49679 received: test1.txt
From client at 49679 received: test3.txt
From client at 49679 received: test2.txt
Client at  ('127.0.0.1', 49679)  disconnected...
Got new connection from ('127.0.0.1', 49682)
From client at 49682 received: test2.txt
Client at  ('127.0.0.1', 49682)  disconnected...
^CServer ended
neeladripal@Neeladris-Macbook-Air Assignment 7 %
```

```
pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % pyenv shell 3.8.6
neeladripal@Neeladris-Macbook-Air Assignment 7 % python ftp_client.py
From Server : You are now connected to server.
        Send the name of the file you want.
        Say end to terminate the session.
Enter file name: test1.txt
From Server : Hello, this is test1.txt content
Enter file name: test3.txt
From Server : File not found
Enter file name: test2.txt
From Server : Hello, this is test2.txt content
Enter file name: end
neeladripal@Neeladris-Macbook-Air Assignment 7 % python ftp_client.py
From Server : You are now connected to server.
        Send the name of the file you want.
        Say end to terminate the session.
Enter file name: test2.txt
From Server : Hello, this is test2.txt content
Enter file name: end
neeladripal@Neeladris-Macbook-Air Assignment 7 %
```

COMMENTS

The assignment gives a glimpse of how servers work, how IP addresses are allocated, how data flows over the network. It would be better if the server and client machines were located in different subnets so as to get a better idea of IP address management.