

# Computer Networks Assignment 5

Harsh Prajapati - 12241300

November 20, 2024

## Part 1 : Implement a Collaborative Text Editor using (TCP) socket programming

In this part, our aim was to create a collaborative text-editor using client-server socket programming. For this, i made 2 separate files named **server.py** and **client.py**. let's take a look at how the program works.

```
import socket
import threading

HOST = '0.0.0.0'
PORT = 8080
```

Figure 1: Code for defining PORT and HOST

Figure 1 shows that we specified **wildcard IP address 0.0.0.0** as **HOST** and **PORT 8080**. This Host address allows connections from any IP address on port 8080.

```
def start_server():
    global clients
    try:
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.bind((HOST, PORT))
        server_socket.listen(10)
        print("Server listening on port 8080")
```

Figure 2: Code for starting the server

Above code snippet of the Figure 2 is to start the server. First, we are creating a socket where **socket.AF\_INET** specifies the use of IPv4 addresses only and **socket.SOCK\_STREAM** specifies the use of TCP socket. Then we associated HOST with PORT number and specified the maximum number of queued connection requests (which is set to 10 here) waiting to be accepted.

```

import socket
import threading

# HOST = '192.168.11.39'
HOST = '127.0.0.1'
PORT = 8080

def receive_updates(client_socket):
    while True:
        try:
            message = client_socket.recv(1024).decode()
            if not message:
                break
            print(message)

        except Exception as e:
            print("Error receiving updates:", e)
            break

def start_client():
    try:
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((HOST, PORT))
        print("Connected to the server. Start typing to edit the document:")

        threading.Thread(target=receive_updates, args=(client_socket,)).start()

        while True:
            update = input()
            if update.lower() == "exit":
                break
            client_socket.sendall(update.encode())

    except Exception as e:
        print("Connection failed. Please check the server and try again.")
    finally:
        client_socket.close()

if __name__ == "__main__":
    start_client()

```

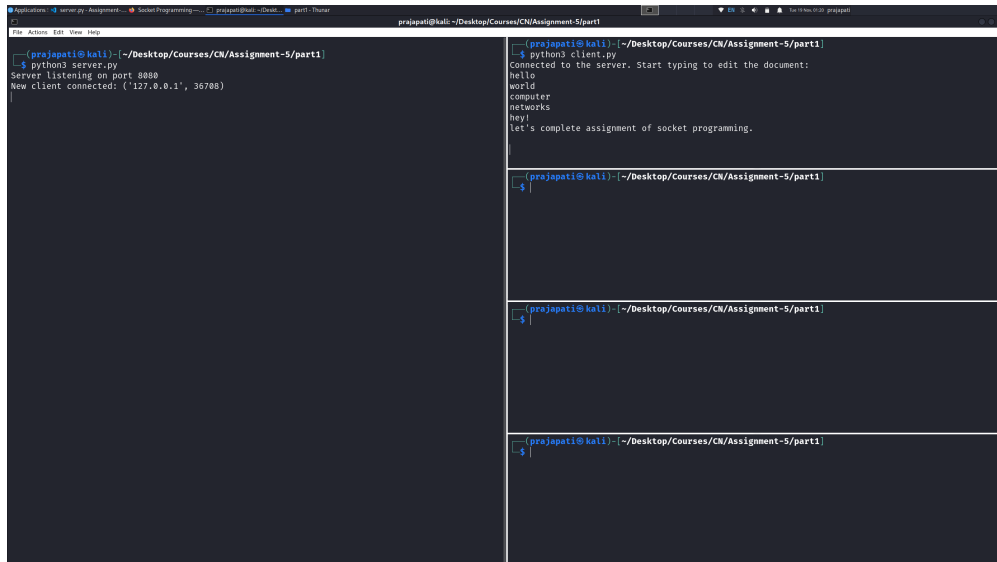
Figure 3: Client side code

As shown in the figure 3, **client.py** file contains the client side code where **HOST = 127.0.0.1** since we are connecting to the server which is running on the localhost machine.

**receive\_updates** function is used for receiving updates from the server side.

**start\_client** function will start the client on IPv4 address using tcp socket and print the message **"Connected to the server. Start typing to edit the document:"** if connection successful.

After that, the present state of the document will be shown to the client.



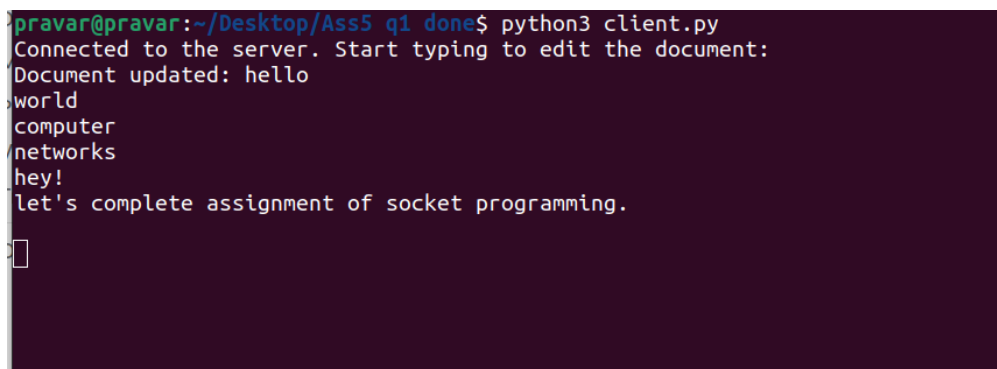
```
prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8080
New client connected: ('127.0.0.1', 35708)

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 client.py
Connected to the server. Start typing to edit the document:
hello
world
computer
networks
hey!
let's complete assignment of socket programming.

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$
```

Figure 4: 1st client connected to the server

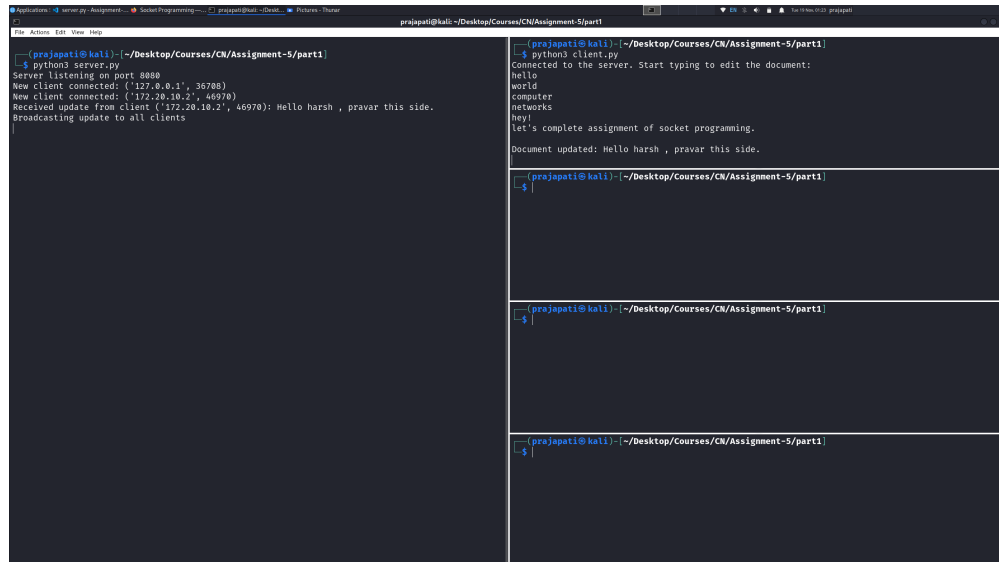
In the figure4, we can see that server was listening on the port 8080 (left half of the picture) and first client connected to the server on that port (client.py is shown on the right half). As client connected to the server, it was served by the current state of the editor which was stored in the **temp.txt** file by the server.



```
pravar@pravar:~/Desktop/Ass5 q1 done$ python3 client.py
Connected to the server. Start typing to edit the document:
Document updated: hello
world
computer
networks
hey!
let's complete assignment of socket programming.


```

Figure 5: A new client from outside connected (2nd client)



```
prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8888
New client connected: ('127.0.0.1', 35788)
New client connected: ('172.28.18.2', 46970)
Received update from client ('172.28.18.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$ python3 client.py
Connected to the server. Start typing to edit the document:
hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Document updated: Hello harsh , pravar this side.

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$
```

Figure 6: A new client from outside connected (2nd client)

In figure 5, we can see that a new client from outside was able to connect the server since we used wildcard address for the HOST to allow these connections.

The message from the remote client is also visible on the terminal of the server and it is **broadcasted** to the other connected client.

At this point in time, first client from the localhost and the remote client s(outside client) both were active at the same time and were connected to the server on different threads for parallel work.

Connections from the clients were being handled by the **function client\_handle** of **server.py** file which is shown in the figure 5.

```

def client_handle(client_socket, client_address):
    global document

    with document_lock:
        client_socket.sendall(document.encode())

    print(f"New client connected: {client_address}")

    while True:
        try:
            update = client_socket.recv(1024).decode()
            if not update:
                break # client disconnected

            print(f"Received update from client {client_address}: {update}")

            with document_lock:
                document += update + "\n"
                with open("temp.txt", "a") as file:
                    file.write(update + "\n")

            broadcast_update(update, client_socket)
            print("Broadcasting update to all clients")

        except Exception as e:
            print(f"Error handling client {client_address}: {e}")
            break

    with document_lock:
        clients.remove(client_socket)
    print(f"Client {client_address} disconnected")
    client_socket.close()

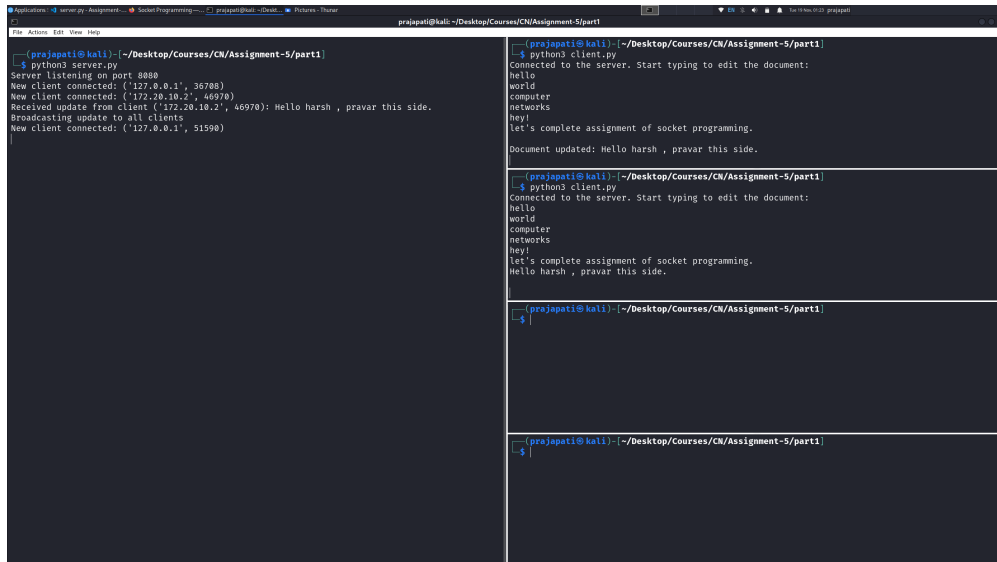
```

Figure 7: Code for handling client

The function handles the client connections which prints **"New client connected: {client\_address}"** whenever a new client connects to the server.

Also this function prints the updates on the terminal received by the clients and broadcast it to all the clients which can be seen in the figures below.

It also prints the disconnected client on the terminal.



```
prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8888
New client connected: ('127.0.0.1', 36788)
New client connected: ('127.20.10.2', 46970)
Received update from client ('127.20.10.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51590)

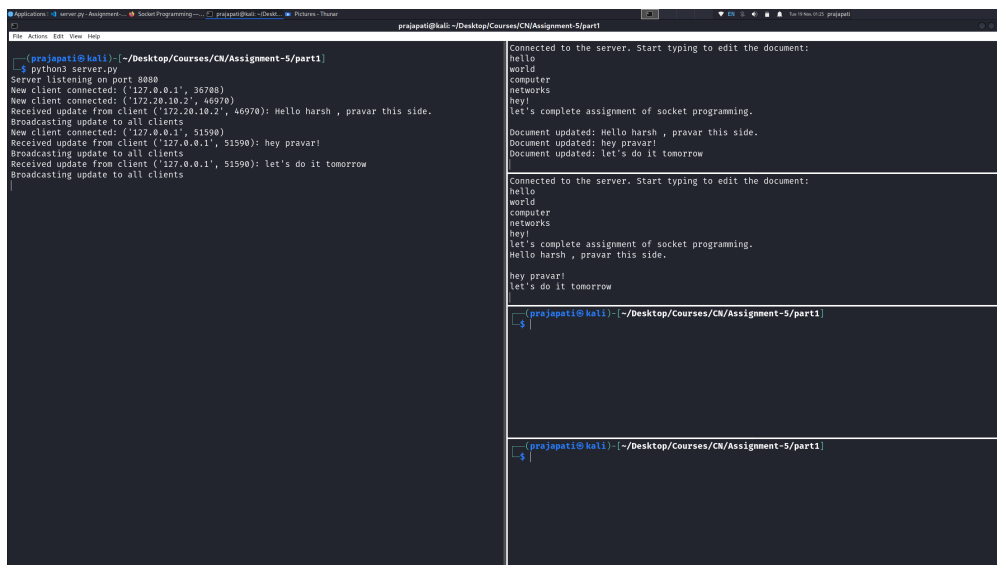
prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$ python3 client.py
Connected to the server. Start typing to edit the document:
hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Document updated: Hello harsh , pravar this side.

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$ python3 client.py
Connected to the server. Start typing to edit the document:
hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Hello harsh , pravar this side.

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$
```

Figure 8: 3rd client connected from the localhost (2nd localhost connection)



```
prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8888
New client connected: ('127.0.0.1', 36788)
New client connected: ('127.20.10.2', 46970)
Received update from client ('127.20.10.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51590)
Received update from client ('127.0.0.1', 51590): hey pravar!
Broadcasting update to all clients
Received update from client ('127.0.0.1', 51590): let's do it tomorrow
Broadcasting update to all clients

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$ python3 client.py
Connected to the server. Start typing to edit the document:
hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Document updated: Hello harsh , pravar this side.
Document updated: hey pravar!
Document updated: let's do it tomorrow

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$ python3 client.py
Connected to the server. Start typing to edit the document:
hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Hello harsh , pravar this side.
hey pravar!
let's do it tomorrow

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$

prajapati@kali:~/Desktop/Courses/CN/Assignment-5/part1
$
```

Figure 9: updation of the editor by 3rd client (2nd localhost connection)

In the figure 8, we can see that client 3 (2nd localhost client) updated the document.

```

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8888
New client connected: ('127.0.0.1', 36788)
New client connected: ('172.20.10.2', 46970)
Received update from client ('172.20.10.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51590)
Received update from client ('127.0.0.1', 51590): hey pravar!
Broadcasting update to all clients
Received update from client ('172.20.10.2', 51590): let's do it tomorrow
Broadcasting update to all clients
Received update from client ('172.20.10.2', 46970): bye brother goodnight
Broadcasting update to all clients
Client ('172.20.10.2', 46970) disconnected

Hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Document updated: Hello harsh , pravar this side.
Document updated: hey pravar!
Document updated: let's do it tomorrow
Document updated: bye brother goodnight

Hello
world
computer
networks
hey!
let's complete assignment of socket programming.
Hello harsh , pravar this side.
hey pravar!
let's do it tomorrow
Document updated: bye brother goodnight

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$

```

Figure 10: Code for starting the server

In the figure 9, we can see that remote client updated the document and then it disconnected from the server.

```

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8888
New client connected: ('127.0.0.1', 36788)
New client connected: ('172.20.10.2', 46970)
Received update from client ('172.20.10.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51590)
Received update from client ('127.0.0.1', 51590): hey pravar!
Broadcasting update to all clients
Received update from client ('127.0.0.1', 51590): let's do it tomorrow
Broadcasting update to all clients
Received update from client ('172.20.10.2', 46970): bye brother goodnight
Broadcasting update to all clients
Client ('172.20.10.2', 46970) disconnected
New client connected: ('127.0.0.1', 57968)
Received update from client ('127.0.0.1', 57968): anyone's there who can work with me?
Broadcasting update to all clients

world
computer
networks
hey!
let's complete assignment of socket programming.
Document updated: Hello harsh , pravar this side.
Document updated: hey pravar!
Document updated: let's do it tomorrow
Document updated: bye brother goodnight
Document updated: anyone's there who can work with me?

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$

```

Figure 11: Client 3 (2nd localhost client) disconnected and 3rd localhost client connected

In the figure 10, we can see that client 3 has been disconnected from the server and a new client (3rd localhost client) connected to the server.



```

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8888
New client connected: ('127.0.0.1', 36788)
New client connected: ('127.20.10.2', 46978)
Received update from client ('127.20.10.2', 46978): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51598)
Received update from client ('127.0.0.1', 51598): hey pravar!
Broadcasting update to all clients
Received update from client ('127.0.0.1', 51598): let's do it tomorrow
Broadcasting update to all clients
Received update from client ('127.20.10.2', 46978): bye brother goodnight
Broadcasting update to all clients
Client ('127.20.10.2', 46978) disconnected
Client ('127.0.0.1', 51598) disconnected
New client connected: ('127.0.0.1', 57968)
Received update from client ('127.0.0.1', 57968): anyone's there who can work with me?
Broadcasting update to all clients
New client connected: ('127.0.0.1', 39178)
Received update from client ('127.0.0.1', 39178): yeah! But we can continue later since not all the clients are up now.
Broadcasting update to all clients

```

computer networks  
hey!  
let's complete assignment of socket programming.  
Document updated: Hello harsh , pravar this side.  
Document updated: hey pravar!  
Document updated: let's do it tomorrow  
Document updated: bye brother goodnight  
Document updated: anyone's there who can work with me?  
yeah! But we can continue later since not all the clients are up now.

```

prajapati@kali: ~/Desktop/Courses/CN/Assignment-5/part1
$

```

computer networks  
hey!  
let's complete assignment of socket programming.  
Hello harsh , pravar this side.  
hey pravar!  
let's do it tomorrow  
bye brother goodnight  
anyone's there who can work with me?  
Document updated: yeah! But we can continue later since not all the clients are up now.

computer networks  
hey!  
let's complete assignment of socket programming.  
Hello harsh , pravar this side.  
hey pravar!  
let's do it tomorrow  
bye brother goodnight  
anyone's there who can work with me?  
Document updated: yeah! But we can continue later since not all the clients are up now.

Figure 12: localhost client 4 connected to the server and client 1 updated the document

Now, a new client (localhost client 4) has been also connected to the server. It received the file before the localhost client 1 updated it. After updates, it also broadcasts the message **Document Updated: ...**

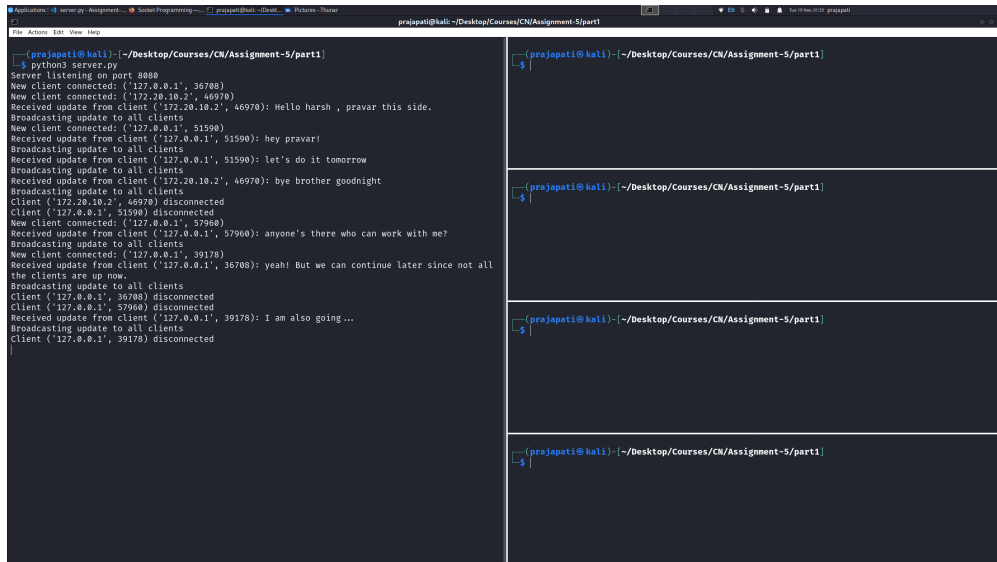
```

def broadcast_update(update, sender_socket):
    with document_lock:
        for client in clients:
            if client != sender_socket:
                try:
                    client.sendall(f"Document updated: {update}".encode())
                except:
                    clients.remove(client)

```

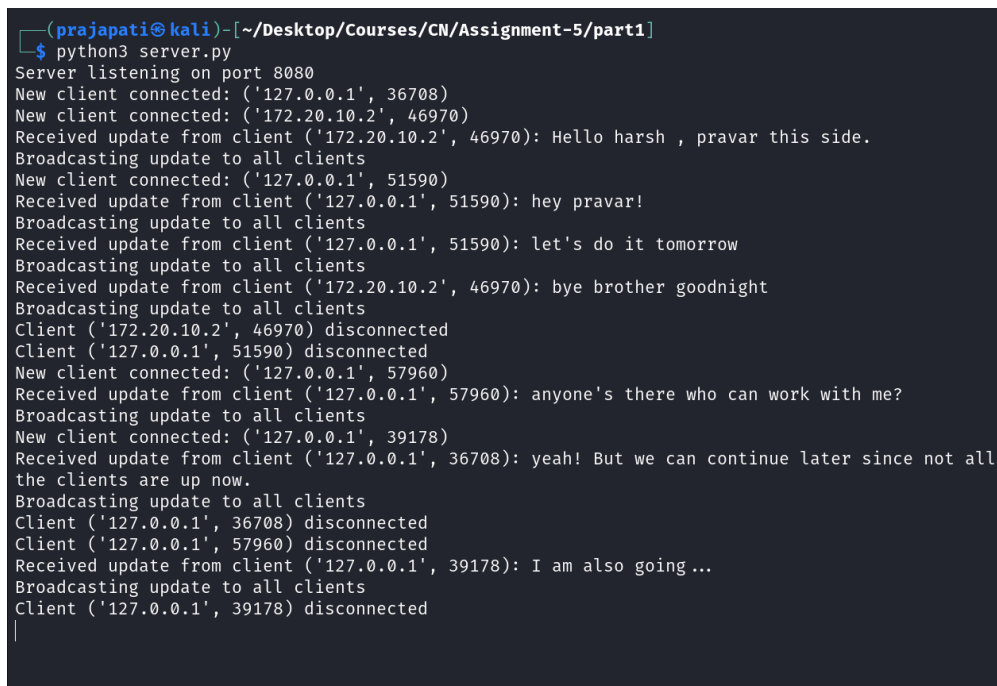
Figure 13: Code for broadcasting updates

The function shown in the above picture is used to broadcast the updates to all the connected clients.



```
(prajapati@kali) ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8080
New client connected: ('127.0.0.1', 36708)
New client connected: ('172.20.10.2', 46970)
Received update from client ('172.20.10.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51590)
Received update from client ('127.0.0.1', 51590): hey pravar!
Broadcasting update to all clients
Received update from client ('127.0.0.1', 51590): let's do it tomorrow
Broadcasting update to all clients
Received update from client ('172.20.10.2', 46970): bye brother goodnight
Broadcasting update to all clients
Client ('172.20.10.2', 46970) disconnected
Client ('127.0.0.1', 51590) disconnected
New client connected: ('127.0.0.1', 57960)
Received update from client ('127.0.0.1', 57960): anyone's there who can work with me?
Broadcasting update to all clients
New client connected: ('127.0.0.1', 39178)
Received update from client ('127.0.0.1', 36708): yeah! But we can continue later since not all
the clients are up now.
Broadcasting update to all clients
Client ('127.0.0.1', 36708) disconnected
Client ('127.0.0.1', 57960) disconnected
Received update from client ('127.0.0.1', 39178): I am also going...
Broadcasting update to all clients
Client ('127.0.0.1', 39178) disconnected
```

Figure 14: All the clients are disconnected



```
(prajapati@kali) ~/Desktop/Courses/CN/Assignment-5/part1
$ python3 server.py
Server listening on port 8080
New client connected: ('127.0.0.1', 36708)
New client connected: ('172.20.10.2', 46970)
Received update from client ('172.20.10.2', 46970): Hello harsh , pravar this side.
Broadcasting update to all clients
New client connected: ('127.0.0.1', 51590)
Received update from client ('127.0.0.1', 51590): hey pravar!
Broadcasting update to all clients
Received update from client ('127.0.0.1', 51590): let's do it tomorrow
Broadcasting update to all clients
Received update from client ('172.20.10.2', 46970): bye brother goodnight
Broadcasting update to all clients
Client ('172.20.10.2', 46970) disconnected
Client ('127.0.0.1', 51590) disconnected
New client connected: ('127.0.0.1', 57960)
Received update from client ('127.0.0.1', 57960): anyone's there who can work with me?
Broadcasting update to all clients
New client connected: ('127.0.0.1', 39178)
Received update from client ('127.0.0.1', 36708): yeah! But we can continue later since not all
the clients are up now.
Broadcasting update to all clients
Client ('127.0.0.1', 36708) disconnected
Client ('127.0.0.1', 57960) disconnected
Received update from client ('127.0.0.1', 39178): I am also going...
Broadcasting update to all clients
Client ('127.0.0.1', 39178) disconnected
```

Figure 15: All the clients are disconnected

Now the figure 13 and 14 shows that all the clients have been disconnected from the server and activities of all the clients are visible on the server side.

## Part 2 : Implement Leader Election and torrent-like File sharing in peer- to-peer Networks using (TCP) socket programming.

### Q1

In this part of the question we have to implement a peer-to-peer network with 3 nodes where all the nodes will show client and server both functionalities.

First all the nodes will start the server on them and they will try to connect to other nodes. After connecting to other nodes, nodes will share their IDs to other nodes. Once the IDs are shared, the node with lowest ID will be elected as a leader and other nodes will keep the connection with leader node only. Other nodes will be disconnected from each other.

File q1.py in the submitted archive's part2 directory, contains the code to achieve this network and other functionalities.

Initial File structure :

Directory q1 :

- **Node1** — 1\_1.txt , 1\_2.txt
- **Node2** — 2\_1.txt
- **Node3** — 3\_1.txt , 3\_2.txt , 3\_3.txt
- Peer\_files.txt
- q1.py
- README.md

Now, let's take a look the execution of the file q1.py

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ python3 q1.py 1
Node 1 is listening on ('127.0.0.1', 8080) ...
Node 1 connected to Node 2.
|

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ python3 q1.py 2
Node 2 is listening on ('127.0.0.1', 8081) ...
Node 2 connected with Node 1.
|
```

Figure 16: Network when only 2 nodes were up

In the figure above, we can see that only two nodes were up at the time and they exchanged their IDs with each other. But still leader was not elected since ID of 3rd node was not known to everyone.

Hence, both the nodes were waiting for the node3 to be connected.

Now, let's see what happened when all the nodes were up. We will analyze each node's logs on terminal.

```

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ python3 q1.py 1
Node 1 is listening on ('127.0.0.1', 8080) ...
Node 1 connected to Node 2.
Node 1 connected to Node 3.
Leader elected: Node 1
Leader Node 1 remains connected to all peers.
Files received from Node 2.
Files received from Node 3.

```

Figure 17: Terminal log for Node1

In the above figure, we can see that after activating all the nodes, nodes have chosen a leader among them who was having lowest ID. In our case it was Node1.

```

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ python3 q1.py 2
Node 2 is listening on ('127.0.0.1', 8081) ...
Node 2 connected with Node 1.

Node 2 connected to Node 3.
Leader elected: Node 1
Node 2 disconnected from Node 3.

```

Figure 18: Terminal log for Node2

In the above image we can see the logs of Node2. After connecting to all the nodes, there was a leader election result shown on the terminal. Since Node1 was elected as a leader, Node2 terminated connection from the Node3.

```

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ python3 q1.py 3
Node 3 is listening on ('127.0.0.1', 8082) ...
Node 3 connected with Node 1.
Node 3 connected with Node 2.
Leader elected: Node 1
Node 3 disconnected from Node 2.

```

Figure 19: Terminal log for Node3

In the above image, we can see that Node3's terminal logs are same as Node2's logs. It also disconnected from the Node2 after Node1 elected as a leader.

Now, let's take a look at the Peer\_files.log file if it logged all the files or not???

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ python3 q1.py 1
Node 1 is listening on ('127.0.0.1', 8080) ...
Node 1 connected to Node 2.
Node 1 connected to Node 3.
Leader elected: Node 1
Leader Node 1 remains connected to all peers.
Files received from Node 2.
Files received from Node 3.

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part2/q1]
$ cat Peer_files.log
Leader Node 1 files:
1_1.txt
1_2.txt

Node 2 files:
2_1.txt

Node 3 files:
3_2.txt
3_3.txt
3_1.txt
```

Figure 20: Data logged into file Peer\_files.txt

In the above image, we can see that file transfer was successful since leader has logged all the files from the nodes and now it has a record of each node that which node has which files.

Now, let's take a look at the code for this question.

```

def run(self):
    self.server_ready.wait()
    time.sleep(3)
    self.connect_to_peers()
    time.sleep(3)
    self elect_leader()
    time.sleep(3)
    self.request_file_logs()
    time.sleep(20)

if __name__ == "__main__":
    import sys

    if len(sys.argv) != 2:
        print("Usage: python3 q1.py <node_id>")
        sys.exit(1)

    node_id = int(sys.argv[1])
    if node_id not in nodes:
        print("Invalid node_id. Must be 1, 2, or 3.")
        sys.exit(1)

    node = PeerNode(node_id)
    node.run()

```

Figure 21: Flow of the function calls

Here, we can see the flow of the function calls for this setup.

`self.server_ready` will wait for server to be fully started and then it calls the `connect_to_peers` function.

As all the peers will connect to each other, the one with the least ID will be elected as a leader.

Then at the end, `request_file_logs` will be called to log files from all the nodes into `Peer_files.log` file.

All the functions are explained below.

```

def __init__(self, node_id):
    self.node_id = node_id
    self.server_address = nodes[node_id]
    self.peer_addresses = {k: v for k, v in nodes.items() if k != node_id}
    self.peer_connections = {}
    self.leader_id = None
    self.server_ready = threading.Event()

    threading.Thread(target=self.start_server, daemon=True).start()

def start_server(self):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #starting server fro IPv4 only
    server_socket.bind(self.server_address)
    server_socket.listen(5)
    print(f"Node {self.node_id} is listening on {self.server_address}...")
    self.server_ready.set()

    while True:
        connection, addr = server_socket.accept()
        threading.Thread(target=self.handle_connection, args=(connection,), daemon=True).start()

```

Figure 22: Start<sub>server</sub> function

start\_server function starts the server to accept Ipv4 connections using TCP.

```

def handle_connection(self, connection):
    try:
        data = connection.recv(1024).decode()
        # print("data" , data)

        if data.startswith("ID:"):
            peer_id = int(data.split(":")[1])
            self.peer_connections[peer_id] = connection
            print(f"Node {self.node_id} connected with Node {peer_id}.")
        elif data == "SEND":
            files = get_file_list(f"Node{self.node_id}")
            # print(files)
            connection.send("\n".join(files).encode())
        else:
            print(f"Node {self.node_id} received unknown command: {data}")
    except Exception as e:
        print(f"Error in connection handling: {e}")

```

Figure 23: handle\_connection

**handle\_connection** function prints the connection between the nodes. If leader has sent **SEND** command that means that leader has asked for the file logs, hence nodes will send that to the leader.



```

def connect_to_peers(self):
    for peer_id, address in self.peer_addresses.items():
        if peer_id not in self.peer_connections:
            while True:
                try:
                    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    sock.connect(address)
                    sock.send(f"ID:{self.node_id}".encode())
                    self.peer_connections[peer_id] = sock
                    print(f"Node {self.node_id} connected to Node {peer_id}.")
                    break
                except ConnectionRefusedError:
                    time.sleep(2)

```

Figure 24: Connect to peers function

This function will be called for the ID exchange between peers for the leader election .

```

def elect_leader(self):
    all_nodes = list(self.peer_connections.keys()) + [self.node_id]
    self.leader_id = min(all_nodes)
    print(f"Leader elected: Node {self.leader_id}")

    # code to disconnect non-leader nodes with each-other
    if self.node_id == self.leader_id:
        print(f"Leader Node {self.node_id} remains connected to all peers.")
    else:
        for peer_id, conn in self.peer_connections.items():
            if peer_id != self.leader_id:
                conn.close()
                print(f"Node {self.node_id} disconnected from Node {peer_id}.")

```

Figure 25: leader election function

**elect\_leader** function was electing leader from all the connected nodes. the node with the lowest ID will be chosen as a leader. after electing leader, it will disconnect non-leader nodes with each other.

```

def request_file_logs(self):
    if self.node_id == self.leader_id:
        with open(log, "w") as log_file:
            log_file.write(f"Leader Node {self.leader_id} files:\n")
            log_file.write("\n".join(get_file_list(f"Node{self.leader_id}")) + "\n\n")

            for peer_id, address in self.peer_addresses.items():
                for _ in range(3):
                    try:
                        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                            sock.connect(address)
                            sock.send("SEND".encode())
                            files = sock.recv(1024).decode()
                            log_file.write(f"Node {peer_id} files:\n{files}\n\n")
                            print(f"Files received from Node {peer_id}.")
                            break
                    except Exception as e:
                        print(f"Error receiving files from Node {peer_id}: {e}")
                        time.sleep(2)

```

Figure 26: request file logs function

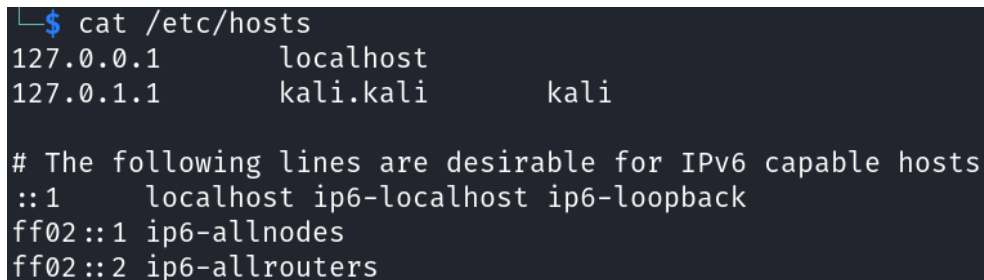
Above function **request\_file\_logs** will only be called by the leader node to log all the files from each node and store the log in the Peer\_files.txt.

## Part 3 : Making Echo Client/Server “Protocol Independent”

The **echo server** is the program which listens for incoming client connections on a specific IP and port number. It accepts connections from the client and receives messages from the connected client. After receiving message, it sends back the same message again to the client.

The **echo client** connects to the echo server using the specified IP and port. It sends the message to the server then receives the same (echoed) message from the server showing that the communication was successful.

Here, we created the echo client-server application which supports both IPv4 and IPv6 protocols for connection protocol-independently. We achieved this by using **getaddrinfo** for dynamically resolving addresses and accommodating address structures with **sockaddr\_storage**.

A terminal window with a dark background and light blue text. The prompt is a green cursor followed by a dollar sign. The command 'cat /etc/hosts' has been executed. The output shows two IPv4 entries for 'localhost' and 'kali.kali', followed by a comment line and three IPv6 entries for 'localhost', 'ip6-allnodes', and 'ip6-allrouters'.

```
$ cat /etc/hosts
127.0.0.1      localhost
127.0.1.1      kali.kali      kali

# The following lines are desirable for IPv6 capable hosts
::1           localhost ip6-localhost ip6-loopback
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters
```

Figure 27: hosts file to verify results of the program

In the above figure, we can see the mappings of hostnames to the corresponding loopback address. There are 2 IPv4 addresses and some IPv6 addresses. But the interesting thing is that **localhost** is found for both network protocols. **So at a time, server may get a request from IPv6 mapping of localhost or from IPv4 mapping.** For this, client will choose which protocol to choose based on the system configurations.

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

|

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : localhost
Successfully connected to the server at ('::1', 8080, 0, 0)

Enter message ('q' to quit): hello, server!
Echoed message from the server: hello, server!

Enter message ('q' to quit): |
```

Figure 28: First client connected to the echo server

In the Figure 16, we can see that server was open for the IPv6 and Ipv4 connections on server port 8080. Then a client connected to the server on port 8080 from **client localhost IP ::1** and **client port 55100**. As client entered a message "hello, server!", It reflected by the server and echoed to the client. This is what the echo-server does with messages received from the clients.

Here localhost was resolved as IPv6 protocol since **most of the client systems prioritizes IPv6 connection over IPv4**.

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

New Client connected : Client IP: ::1 Client Port: 36236

|
```

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : localhost
Successfully connected to the server at ('::1', 8080, 0, 0)

Enter message ('q' to quit): hello, server!
Echoed message from the server: hello, server!

Enter message ('q' to quit): |
```

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : ip6-localhost
Successfully connected to the server at ('::1', 8080, 0, 0)

Enter message ('q' to quit): Client2 with IP6
Echoed message from the server: Client2 with IP6

Enter message ('q' to quit): |
```

Figure 29: 2nd client connected to echo-server

Here, 2nd client also connected to echo-server but it was connected via **ip6-localhost** hence the connection was set up on IPv6 from **client port 36236**.

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

New Client connected : Client IP: ::1 Client Port: 36236

Client with Client IP: ::1 and Client Port: 55100 disconnected.

|

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : localhost
Successfully connected to the server at ('::1', 8080, 0, 0)

Enter message ('q' to quit): hello, server!
Echoed message from the server: hello, server!

Enter message ('q' to quit): q

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$
```

Figure 30: Client 1 disconnected from the server

In the figure 18, we can see that client 1 was disconnected from the server.

```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

New Client connected : Client IP: ::1 Client Port: 36236

Client with Client IP: ::1 and Client Port: 55100 disconnected.

New Client connected : Client IP: ::ffff:127.0.0.1 Client Port: 60508
|

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : 127.0.0.1
Successfully connected to the server at ('127.0.0.1', 8080)

Enter message ('q' to quit): client3 with IPv4
Echoed message from the server: client3 with IPv4

Enter message ('q' to quit): |
```

Figure 31: Another client connected to server via IPv4

In the above image (figure 19) , we can see that the client is connected to the server via **localhost IPv4 address 127.0.0.1 from client port 60508.**

This shows that the echo-server accepts connections for both the Network address family IPv4 and IPv6.

Here, it stored ipv4 address as ipv6 address. Hence, our purpose to create a socket which accepts teh connections from both the address protocols is fulfilled.

```

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

New Client connected : Client IP: ::1 Client Port: 36236

Client with Client IP: ::1 and Client Port: 55100 disconnected.

New Client connected : Client IP: ::ffff:127.0.0.1 Client Port: 60508

Client with Client IP: ::1 and Client Port: 36236 disconnected.

|

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : 127.0.0.1
Successfully connected to the server at ('127.0.0.1', 8080)

Enter message ('q' to quit): client3 with IPv4
Echoed message from the server: client3 with IPv4

Enter message ('q' to quit): |

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : ip6-localhost
Successfully connected to the server at ('::1', 8080, 0, 0)

Enter message ('q' to quit): Client2 with IP6
Echoed message from the server: Client2 with IP6

Enter message ('q' to quit): q

```

Figure 32: disconnected client 2

In the figure 20, we can see that client 2 was disconnected from the server.



```
(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

New Client connected : Client IP: ::1 Client Port: 36236

Client with Client IP: ::1 and Client Port: 55100 disconnected.

New Client connected : Client IP: ::ffff:127.0.0.1 Client Port: 60508

Client with Client IP: ::1 and Client Port: 36236 disconnected.

New Client connected : Client IP: ::ffff:127.0.0.1 Client Port: 57704
|

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : 127.0.0.1
Successfully connected to the server at ('127.0.0.1', 8080)

Enter message ('q' to quit): client3 with IPv4
Echoed message from the server: client3 with IPv4

Enter message ('q' to quit): |

(prajapati@kali)-[~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-client.py
Enter the server address : kali
Successfully connected to the server at ('127.0.1.1', 8080)

Enter message ('q' to quit): client4
Echoed message from the server: client4

Enter message ('q' to quit): |
```

Figure 33: Another client connected to the server by resolving hostname kali

In the above figure client 4 connects to the server using **hostname kali (IP : 127.0.1.1)** from **client port 57704**. Here the message client4 was echoed from the server.

```
(prajapati@kali) - [~/Desktop/Courses/CN/Assignment-5/part3]
$ python3 echo-server.py
Server listening for connections on port 8080
New Client connected : Client IP: ::1 Client Port: 55100

New Client connected : Client IP: ::1 Client Port: 36236

Client with Client IP: ::1 and Client Port: 55100 disconnected.

New Client connected : Client IP: ::ffff:127.0.0.1 Client Port: 60508

Client with Client IP: ::1 and Client Port: 36236 disconnected.

New Client connected : Client IP: ::ffff:127.0.0.1 Client Port: 57704

Client with Client IP: ::ffff:127.0.0.1 and Client Port: 60508 disconnected.

Client with Client IP: ::ffff:127.0.0.1 and Client Port: 57704 disconnected.
```

Figure 34: final log of the server side

At last we can see that last two connections were also terminated by the clients and server was listening for the new connections.

The above snapshots of running programs and explanation shows that our echo client-server application is well-designed and functions properly to echo (reflect) messages received from the clients.

**This is used to test if our network applications are working properly or not.** If we are receiving the same message which we sent to the server from the client side application, then we can get surity of our application that messages were recieved at the server properly. Also, it can be used to measure RTT (round trip time), network connectivity and some other applications.

Now, let's take a look at the code that how it accepted

```
def server_socket(port):
    server_socket = socket.socket(socket.AF_INET6, socket.SOCK_STREAM) #line1
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #line2
    server_socket.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_V6ONLY, 0) #line3
    server_socket.bind("", port) #line4
    server_socket.listen(5) #line5
    return server_socket #line6
```

Figure 35: Socket creation of server

- Line1 creates a TCP socket using IPv6 address family (**AF\_INET6**) which supports only IPv6 addresses by default.
- Line2 allows the reuse of local address and port.

- Line3 enables the dual-stack mode of our socket. setting the value of **socket.IPV6\_V6ONLY** to **0 (false)** allows this socket to accept both IPv6 and Ipv4 connections.
- Line4 binds all the available network interfaces to a specific port. In this case, we fixed port to 8080.
- Line5 specifies the max. number of queued connections before refusing new connections.
- Line6 returns the configured server socket.