CS301 Computer Networks Assignment 5

Ojus Goel 12241190

November 20, 2024

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

Setup and File Creation

Setup and File Generation The server has to generate and manage a file called temp.txt to retain the content of the document persistently.

1. **Creating temp.txt File**

   File Creation Check: At the time of starting, the server does a file-existence check for temp.txt. It creates the file if the file does not exist so that, even when no old data exists, there will always be a file where to write in updated version of documents.

   File Persistence: Each update received from a client is appended to temp.txt. That means the state of the document is persisted, and the loaded updates may be retrieved if the server restarts. Creation and Updating of File File creation and updating is done as shown below. If temp.txt does not exist, the server creates an empty file by using the open(”temp.txt”, ”w”) method. If the file exists, the server reads in its contents to load the previous state of the document into memory.

```python
# Function to start server
def start_server():
    global document

    # Create temp.txt if it doesn't exist
    try:
        if not os.path.exists("temp.txt"):
            open("temp.txt", "w").close()
            print("[INFO] Created temp.txt file")

    # Printing error and stopping the server
    except Exception as e:
        print("[ERROR] Error opening file temp.txt")
        return

    # Loadign existing content from temp.txt
    try:
        with open("temp.txt", "r") as file:
            document = file.read()
        print("[INFO] Loaded existing content from temp.txt")

    except Exception as e:
        print("[ERROR] Error reading file temp.txt")
        return
```

2. **Server Listening Message** Once the server is ready to accept connections, it binds to the local network interface and listens on port 8080. That's the communication channel for the clients to connect to the server. Once the server starts listening, it prints: "Server listening on port 8080". This message means that the server is now actually waiting to receive a connection from an incoming client.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 server.py
[INFO] Loaded existing content from temp.txt
[INFO] Server listening on port 8080
```

```python
# Starting the server
try:
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(("0.0.0.0", 8080))
    server_socket.listen(10)
    print("[INFO] Server listening on port 8080")
```

3. **Handling Exception for temp.txt** Error Handling When Creating a File: If the server is unable to open or create the temp.txt file then an error occurs such as a permission error or a file-system problem, the server prints an error message to the administrator's console. The error message is: Error opening file temp.txt This informs the server operator of any issues related to files that prevent the server from working properly.

Server

The server is intended to handle multiple clients simultaneously while allowing them to communicate in real-time over a shared document.

1. **Accepting Connections**

   Concurrency: The server can run with maximum clients of 10 simultaneously. This is achieved by running threads. For each client connection, there's a different thread, which ensures that the server does not block other connections while serving various clients.

   Client Connection Logging: When a new client connects to the server, a message is displayed that shows the address of the client. For every incoming connection, the server logs the following message: "New client connected: client_address "

```python
# Function to handle the clients
def handle_client(client_socket, client_addr):
    global document
    print(f"[INFO] New client connected: {client_addr}")
```

```python
with client_lock:

    # Checking number of connected clients. Reject if more than 10 clients
    num_clients = 10 # Change number of allowable clients

    if len(list_clients) >= num_clients:
        print(f"[WARN] Server full. Rejected connection from {client_addr}")
        client_socket.sendall("Server is full. Please try again later.".encode())
        client_socket.close()

    else:
        list_clients.append(client_socket)
        print(f"[INFO] Accepted connection from {client_addr}")
        threading.Thread(target=handle_client, args=(client_socket, client_addr)).start()
```

We allowed only 10 clients as asked in the question but this can be modified.

2. **Sending Initial Document State**

   Sending Document on Connect On connecting to the server, every client receives the current state of the document. In this way, each client is able to start with the latest version of the document that contains all preexisting content. The document resides in memory but remains in the temp.txt file throughout which it is accessed by reading from and writing to through the server.

   Message Flow: The server transmits the document's content to the client using socket's sendall() method. This way, if a client is just joining a session, it will notice the most recent version of the document.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 client.py
Connected to the server. Start typing to edit the document:
Document updated: Hello there! This is client 1
hello I am filling the document just to demonstrate that the data existing in file can be seen by newly connected client

This is the newly connected client and can observe previous data
```

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 server.py
[INFO] Loaded existing content from temp.txt
[INFO] Server listening on port 8080
[INFO] Accepted connection from ('127.0.0.1', 55832)
[INFO] New client connected: ('127.0.0.1', 55832)
[INFO] Sent current document to client ('127.0.0.1', 55832)
[UPDATE] Document updated by ('127.0.0.1', 55832): Hello there! This is client 1
[UPDATE] Document updated by ('127.0.0.1', 55832): hello I am filling the document just to demonstrate that the data existing in
 file can be seen by newly connected client
[INFO] Accepted connection from ('127.0.0.1', 40442)
[INFO] New client connected: ('127.0.0.1', 40442)
[INFO] Sent current document to client ('127.0.0.1', 40442)
```

```python
# Sending the current doc state to the newly connected client
with doc_lock:
    try:
        client_socket.sendall(document.encode())
        print(f"[INFO] Sent current document to client {client_addr}")
    except Exception as e:
        print(f"[ERROR] Failed to send document to client {client_addr}: {e}")
```

3. **Receiving and Broadcasting Updates** Receiving Client Updates: When ever the client transmits an update of the document, the server accepts the update by: Appended Update: The server writes the obtained update to the document variable held in the server's memory. Thus, the real-time state of a document is kept. Persistence: To ensure that the content of a

document is saved, the server also writes the update to the temp.txt file, and thus, the history of a document is preserved even if the server goes down. Broadcasting Updates: The server broadcasts updates received and processed to all the connected clients except the sender. This is done so that all the clients remain in sync with each other and can see the changes performed by others. Broadcasting is accomplished by sending an update to all the clients in the list of clients excluding the client which initiated the update.

```python
# Broadcasting the update to all clients (except the sender)
clients_snapshot = []
with client_lock:
    clients_snapshot = list_clients[:]
for client in clients_snapshot:
    if client != client_socket:
        try:
            client.sendall(update.encode())
            print(f"[INFO] Broadcasted update to client {client.getpeername()}")
        except Exception as e:
            print(f"[ERROR] Failed to send update to client {client.getpeername()}: {e}")
```

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 client.py
Connected to the server. Start typing to edit the document:
Document updated: Hello there! This is client 1
hello I am filling the document just to demonstrate that the data existing in file can be seen by newly connected client

This is the newly connected client and can observe previous dataDocument updated: hi from client1 should be broadcasted to clien
t2

Document updated: Hi from client-1 broadcasting to client2
```

Logging on Update: The server logs: "Received update from client client_address: update_content" Once the server broadcasts the update to other clients it then logs: "Broadcasting update to all clients"

```
[INFO] Accepted connection from ('127.0.0.1', 37286)
[INFO] New client connected: ('127.0.0.1', 37286)
[INFO] Sent current document to client ('127.0.0.1', 37286)
[UPDATE] Document updated by ('127.0.0.1', 37286): hey all. this is client3
[INFO] Broadcasted update to client ('127.0.0.1', 55832)
[INFO] Broadcasted update to client ('127.0.0.1', 40442)
```

4. **Client Disconnection Handling**

Dropping Disconnected Clients: Once the client has disconnected either on their own account or due to a connection reset, the server drops the client from the list of active clients (list_clients). This way, the server won't try endlessly to send updates to disconnected clients.

Disconnection Logging: Whenever a client disconnect is detected, the server logs: " Client client_address disconnected"

```python
# Handling client disconnection
finally:
    with client_lock:
        if client_socket in list_clients:
            list_clients.remove(client_socket)

    client_socket.close()
    print(f"[INFO] Client {client_addr} disconnected")
```

```
[UPDATE] Document updated by ('127.0.0.1', 37286): Disconnecting client3. see ya around!
[INFO] Broadcasted update to client ('127.0.0.1', 55832)
[INFO] Broadcasted update to client ('127.0.0.1', 40442)
[INFO] Client ('127.0.0.1', 37286) disconnected
```

Client

Implementation of client-side application. This code established TCP connection with server with real time communication. It operates in two concurrent threads, one listens to update from server and display them to user while other sends update from users to server. We use multi threading for synchronization.

Connection to Server: The client prints the following message when successfully connected to the server: "Connected to the server. Start editing to see non-realtime results" This is an indication that a client has successfully established communication with the server, and it is ready to receive and transmit updates to the document.

```python
def main():

    try:
        # creating socket for client with IPv4 and TCP as transport protocol
        socket_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        socket_client.connect(("127.0.0.1", 8080))
        print("Connected to the server. Start typing to edit the document:")

        # Start a thread to receive updates from the server
        threading.Thread(target=get_updates, args=(socket_client,), daemon=True).start()

        # Sending user edits to the server
        while True:
            inp = input()
            socket_client.sendall(inp.encode())

    except ConnectionRefusedError:
        print("Connection failed. Please check the server and try again")

    finally:
        socket_client.close()
```

Document State and Updates Receiving:

The client is always scanning for updates from the server. Upon receiving an update, the client displays the following message to signify that the document has been updated: "Document updated: ¡received_content¿" In this case, ¡received_content¿ is the newest content or update coming from another client or the server and presents the changes to the document to the user in a real-time manner.

```python
def get_updates(socket_client):

    while True:

        try:
            # takes updates upto 1024 bytes from the server and decodes using standard decoding (UTF-8)
            update = socket_client.recv(1024).decode()

            # break if no update is received
            if not update:
                break

            print(f"Document updated: {update}")

        except:
            break
```

Sending User Edits:

The client captures the user input via the console and sends this as an update to the server. Now, immediately after the update has been sent, no further message is printed on the client side. This means there is no feedback following every input from the user. Only the content the user entered is being sent to the server, which then propagates it to the other clients that are connected.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 client.py
Connected to the server. Start typing to edit the document:
Hello there! This is client 1
```

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 server.py
[INFO] Loaded existing content from temp.txt
[INFO] Server listening on port 8080
[INFO] Accepted connection from ('127.0.0.1', 55832)
[INFO] New client connected: ('127.0.0.1', 55832)
[INFO] Sent current document to client ('127.0.0.1', 55832)
[UPDATE] Document updated by ('127.0.0.1', 55832): Hello there! This is client 1
```

Concurrency and Synchronization

To ensure smooth and conflict-free simultaneous edits by multiple clients, the server implements concurrency and synchronization as follows:
Thread-Based Client Handling:
Each client connection is managed in a separate thread, enabling multiple clients to connect and edit the document simultaneously.
Mutex for Synchronization:
Document Access: A mutex (doc_lock) ensures synchronized access to the shared document in memory, preventing race conditions during read and write operations.
File Operations. The very same mutex is used to lock file access (temp.txt) when appending updates in order to ensure consistent, conflict-free file writing.
The above design preserves the integrity of both document states-in-memory and persistent-while supporting concurrency, as clients can interact with each other.

File Saving and persistence

Saving And Persistence of a File The server maintains the state of the document across sessions. It saves each client update into a file named temp.txt to ensure that any incoming updates do not lose the content of the document if the server stops or restarts. Here is a detailed explanation of how saving and persistence of a file works:

1. **Append the Updates to temp.txt**

   Continuous Saving of Updates: For every update received from the clients by the server, it adds this update to the temp.txt file immediately. Hence, the state of the document is always saved, which reduces the chances of any data being lost in case the server stops unexpectedly.
   File Appending Process: If a server receives a client update, the server writes it to the temp.txt through the following code snippet:

```python
# Appending data to document and temp.txt
with doc_lock:
    document += update + "\n"
    with open("temp.txt", "a") as file:
        file.write(update + "\n")
    print(f"[UPDATE] Document updated by {client_addr}: {update.strip()}")
```

The a mode in the open() function makes sure that this update is appended at the end of the file instead of overwriting the present content. This ensures that the document continues to evolve while saving every change.

2. **Copy of Document's State** Persistent Backup: The temp.txt is always a kind of backup for the content of the document. Every time a new update is received and appended, it ensures that a persistent record of the document is available. In case of server failure or restart, this type of backup file contains the recent version of the document to allow the server to reload the last saved state .

   Reload the Document:

   The server reads the contents of the file temp.txt upon startup to load the last saved state of the document into memory. This mechanism enables a server to continue serving clients from where it left off, with no updates lost in the process.

```
# Loadign existing content from temp.txt
try:
    with open("temp.txt", "r") as file:
        document = file.read()
    print("[INFO] Loaded existing content from temp.txt")

except Exception as e:
    print("[ERROR] Error reading file temp.txt")
    return
```

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 client.py
Connected to the server. Start typing to edit the document:
Document updated: Hello there! This is client 1
hello I am filling the document just to demonstrate that the data existing in file can be seen by newly connected client
hi from client1 should be broadcasted to client2
This is the newly connected client and can observe previous dat
Hi from client-1 broadcasting to client2
hey all. this is client3
Disconnecting client3. see ya around!
```

Error Handling

Error Handling Server Start Errors:

If the server fails to create or bind the socket, it prints: "Server error: Unable to start server on port 8080".

If it fails to open or create temp.txt, then it prints: "Error opening file temp.txt".

Client Connect Errors:

If a client is unable to connect to the server, it prints: "Connection failed. Please check the server and try again".

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5$ python3 server.py
[INFO] Loaded existing content from temp.txt
[ERROR] Server error: [Errno 98] Address already in use
[INFO] Server shut down
```

```
except ConnectionRefusedError:
    print("Connection failed. Please check the server and try again")
```

```
# Server Shutdown
except Exception as e:
    print(f"[ERROR] Server error: {e}")

finally:
    server_socket.close()
    print("[INFO] Server shut down")
```

```
except Exception as e:
    print("[ERROR] Error reading file temp.txt")
    return
```

```
# Printing error and stopping the server
except Exception as e:
    print("[ERROR] Error opening file temp.txt")
    return
```

```
with doc_lock:
    try:
        client_socket.sendall(document.encode())
        print(f"[INFO] Sent current document to client {client_addr}")
    except Exception as e:
        print(f"[ERROR] Failed to send document to client {client_addr}: {e}")
```

Part 3: Making Echo Client/Server Protocol Independent

The aim was to implement an echo client-server application that supports both IPv4 and IPv6. To achieve this, the following modifications and strategies were applied:

- **Protocol Independence:**

  The application makes the use of getaddrinfo() to convert hostnames and ports into protocol-independent addresses. Servers can bind to any IP address (IPv4 or IPv6) using dual-stack socket.

- **Compatibility:**

  The application follows the API of socket programming using socket(), bind(), connect(), send()/recv() for message exchange.

- **Use of sockaddr_storage:**

  The server side used sockaddr_storage to accommodate both IPv4 and IPv6 address structures. This ensures the application can support both address types without size restrictions.

Client

The client program is intended to:
Resolve the server's hostname and port into a protocol-agnostic address dynamically using getaddrinfo() and connect to the server without knowing whether its address is an IPv4 or IPv6 Code Fragments.

- **Address Resolution:**

  The getaddrinfo() function performs hostname-to-address mappings that return both IPv4 and IPv6 addresses depending on system configuration. The first valid address is used in creating a socket, which will be connected to the server.

- **Dynamic Socket Creation:**

  The socket family and address are resolved at runtime to ensure protocol independence.

- **Data Transfer:**

  The client uses sendall() and recv() to send and receive messages, which echo back the server's response to the user.

getaddrinfo() returns a list of tuples, with each tuple providing information about a resolved address. The format of each tuple is:
(family, socketType, protocol, canonicalName, socketAddress)

1. `family|`:

   Specifies the address family for the resolved address. Possible values include:

   - `socket.AF_INET` (IPv4)
   - `socket.AF_INET6` (IPv6)

2. `socketType|`:

   Specifies the type of socket for the address. Common values are:

   - `socket.SOCK_STREAM` (TCP for reliable connections)
   - `socket.SOCK_DGRAM` (UDP for connectionless communication)

3. `protocol|`:

   Indicates the protocol to use with the socket. Typical values are:

   - `socket.IPPROTO_TCP` (for TCP)
   - `socket.IPPROTO_UDP` (for UDP)

4. `canonicalName|`:

   A string with the canonical (official) name of the host if `AI_CANONNAME` is specified in the `flags` parameter. Otherwise, it's an empty string.

5. `socketAddress|`:

   A tuple containing the resolved address and port:

   - For IPv4: `(address, port)` (e.g., `("192.168.1.1", 8080)`)
   - For IPv6: `(address, port, flowInfo, scopeId)` (e.g., `("::1", 8080, 0, 0)`)
     - `flowInfo`: Information for Quality of Service (QoS).
     - `scopeId`: Scope for the address (used for link-local IPv6 addresses).

```python
import socket

def echo_client(hostname, port):
    # Resolving address for hostname and port
    addrInfo = socket.getaddrinfo(
        hostname, port, socket.AF_UNSPEC, socket.SOCK_STREAM
    )

    # Choosing the first valid address
    family, socketType, protocol, canonical, socketaddr = addrInfo[0]

    # Creating the socket and connecting to the sevrer
    with socket.socket(family, socketType, protocol) as client_socket:
        client_socket.connect(socketaddr)
        print(f"Connected to server at {socketaddr}")

        while True:
            message = input("Enter message to send (or 'exit' to quit): ")

            if message.lower() == 'exit':
                break

            # sending data to server
            client_socket.sendall(message.encode())

            # receiving data from the server
            data = client_socket.recv(1024)
            print(f"Echo from server: {data.decode()}")

if __name__ == "__main__":
    hostname = input("Enter the server address (e.g., 'localhost', '::1'): ")
    # port = int(input("Enter the server port (e.g., 12345): "))
    port = 8080
    echo_client(hostname, port)
```

Server

The server program is written to:
Use a dual-stack socket to enable communications using both IPv4 and IPv6 and bind to the wildcard address (::) for IPv6, so it can accept IPv4 traffic if the system supports dual-stack sockets and dynamically handle incoming connections using sockaddr_storage to accommodate the possibility of IPv4 and IPv6 clients.

- **Dual-Stack Support:**

  The server creates an IPv6 socket (AF_INET6) and binds to ::. This allows for compatibility with IPv4 addresses through automatic address mapping.

- **Connection Handling:**

  The server accepts incoming connections and determines the client's address family dynamically.

- **Threaded Client Management:**

  Each client connection is treated in a separate thread to enable concurrent communication.

- **Use of sockaddr_storage**

  This ensures sufficient memory allocation for both sockaddr_in (IPv4) and sockaddr_in6 (IPv6).

```python
# Managing the client
def manage_client(connection, client_addr):

    # client_addr is tuple containing client's IP addr and port number
    print(f"Connection from [{client_addr[0]}]:: port :{client_addr[1]} (IPv6)")

    with connection:
        while True:
            # receive inp from client (upto 1024 bytes)
            inp = connection.recv(1024)
            if not inp:
                break
            print(f"Received: {inp.decode()} from [{client_addr[0]}]:{client_addr[1]}")

            # echo the data
            connection.sendall(inp)

    print(f"Connection closed for [{client_addr[0]}]:{client_addr[1]}")
```

```python
def echo_server(hostname, port):

    # Creating an IPv6 socket
    with socket.socket(socket.AF_INET6, socket.SOCK_STREAM) as server_socket:

        # Allowing reuse of address and port if the server restarts quickly after shutting down.
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        # Binding to the specified hostname and port for IPv6
        server_socket.bind(('', port))
        server_socket.listen(5)
        print(f"Server is listening on [{hostname}]:{port} (IPv6)")

        while True:
            # Accepting a new connection
            connection, client_addr = server_socket.accept()

            # Creating a new thread to handle the client
            client_thread = threading.Thread(target=manage_client, args=(connection, client_addr))
            #Daemonize the thread to terminate with the main program
            client_thread.daemon = True
            client_thread.start()
```

Code

Finding out all the mappings in my computer
We can either use localhost or 127.0.0.1 for running our code now (For IPv4).
Similarily we have mapping for IPv6.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       ojusg-Inspiron-14-5420

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Now trying to run client code without running server code.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): localhost
Traceback (most recent call last):
  File "/home/ojusg/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3/client.py", line 29, in <module>
    echo_client(hostname, port)
  File "/home/ojusg/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3/client.py", line 14, in echo_client
    client_socket.connect(sockaddr)
ConnectionRefusedError: [Errno 111] Connection refused
```

This throws an error as server is not yet started. Let's start the server now.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 server.py
Server is listening on [::]:8080 (IPv6)
```

The server starts listening on port 8080. We can change this in the code.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): localhost
Connected to server at ('127.0.0.1', 8080)
Enter message to send (or 'exit' to quit):
```

Connected to localhost. This is an IPv4 address.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): ::1
Connected to server at ('::1', 8080, 0, 0)
Enter message to send (or 'exit' to quit):
```

Connected to ::1. This is an IPv6 address.

```
(base) ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): ojusg-Inspiron-14-5420
Connected to server at ('127.0.1.1', 8080)
Enter message to send (or 'exit' to quit):
```

Connected to ojusg-Inspiron-14-5420. This is again an IPv4 address.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 server.py
Server is listening on [::]:8080 (IPv6)
Connection from [::ffff:127.0.0.1]:40412 (IPv6)
Connection from [::1]:40752 (IPv6)
Connection from [::ffff:127.0.0.1]:47224 (IPv6)
```

All the IPv4 and IPv6 addresses are connected to server successfully.

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): localhost
Connected to server at ('127.0.0.1', 8080)
Enter message to send (or 'exit' to quit): Hey. I am IPv6 address.
Echo from server: Hey. I am IPv6 address.
Enter message to send (or 'exit' to quit): Sorry, I am an IPv4 address. My bad!
Echo from server: Sorry, I am an IPv4 address. My bad!
Enter message to send (or 'exit' to quit):
```

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): ::1
Connected to server at ('::1', 8080, 0, 0)
Enter message to send (or 'exit' to quit): Hey. Pls echo. I am IPv6 address
Echo from server: Hey. Pls echo. I am IPv6 address
Enter message to send (or 'exit' to quit):
```

```
(base) ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 client.py
Enter the server address (e.g., 'localhost', '::1'): ojusg-Inspiron-14-5420
Connected to server at ('127.0.1.1', 8080)
Enter message to send (or 'exit' to quit): I am again an mapping to an IPv4 address. Hey there!
Echo from server: I am again an mapping to an IPv4 address. Hey there!
Enter message to send (or 'exit' to quit):
```

```
ojusg@ojusg-Inspiron-14-5420:~/Desktop/College_Sem_V/Computer_Networks/Assignment5/Part-3$ python3 server.py
Server is listening on [::]:8080 (IPv6)
Connection from [::ffff:127.0.0.1]:40412 (IPv6)
Connection from [::1]:40752 (IPv6)
Connection from [::ffff:127.0.0.1]:47224 (IPv6)
Received: Hey. I am IPv6 address. from [::ffff:127.0.0.1]:40412
Received: Sorry, I am an IPv4 address. My bad! from [::ffff:127.0.0.1]:40412
Received: Hey. Pls echo. I am IPv6 address from [::1]:40752
Received: I am again an mapping to an IPv4 address. Hey there! from [::ffff:127.0.0.1]:47224
```

Successfully sent message from all the clients individually to the server and got the corresponding echo from server for each host as shown in above screenshots.
We hence resolved echo client and server to be protocol independent.