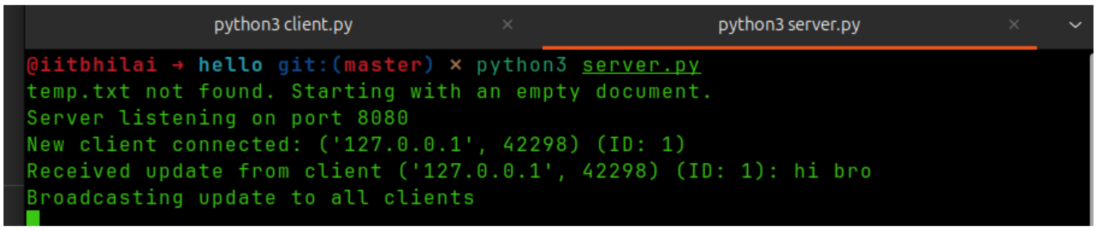
CS300: Computer Networks(2023–24–M)
Assignment

Full name : Munjam Navadeep **ID: 12241060**

Question 1



```
@iitbhilai → hello git:(master) ✗ python3 server.py
temp.txt not found. Starting with an empty document.
Server listening on port 8080
New client connected: ('127.0.0.1', 42298) (ID: 1)
Received update from client ('127.0.0.1', 42298) (ID: 1): hi bro
Broadcasting update to all clients
```

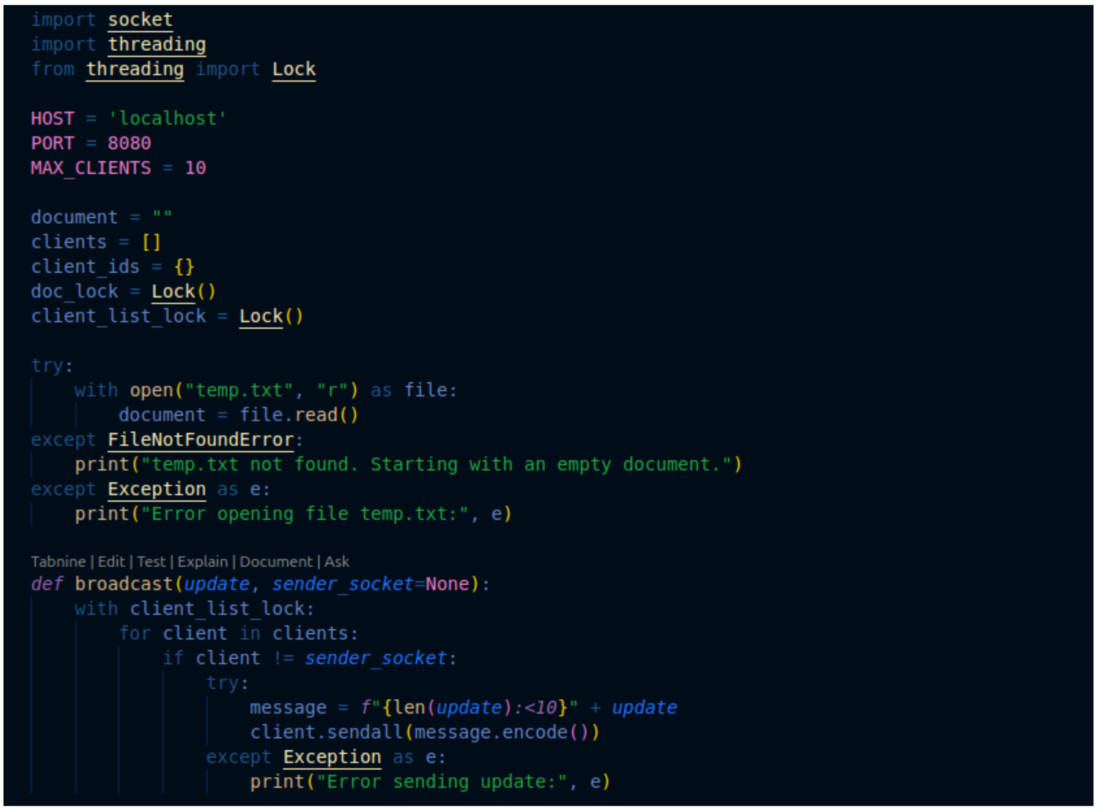
Figure 1: output when no temp.txt file is present

1. **Socket Setup and Connection**:
   - `HOST` and `PORT` define the server address (`'localhost'`) and port (8080) for the connection.
   - `client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)` This actually creates a TCP connection
   - `client_socket.connect((HOST, PORT))` It will try to establish a connection with server on specified host and specified port
2. **Receiving Updates**:
   - The function `for_receiving_The_updates(sock)` will run in a separate thread so that it can continuously receive updates from the server.
   - First it reads a header (10 bytes) which it is length of incoming message from client
   - The function retrieves the actual message content based on the length indicated by the header using `sock.recv(updated_the_length)`.
   - If the data is not received or if any error occurs the loop automatically terminates and exits.
3. **Thread for Asynchronous Reception**:
   - A new thread is created using Python's `threading.Thread` class to run `for_receiving_The_updates`. this helps the client to send messages and receive messages simultaneously without blocking anything.
4. **Sending Updates**:
   - In the main loop, the client waits for user input using `input()`.

- The message is formatted by prepending the length of the message to enable the server to correctly read and process the data.
- The combined message (`message = f"{len(update):<10}" + update`) is sent over the network using `client_socket.sendall(message.encode())`.

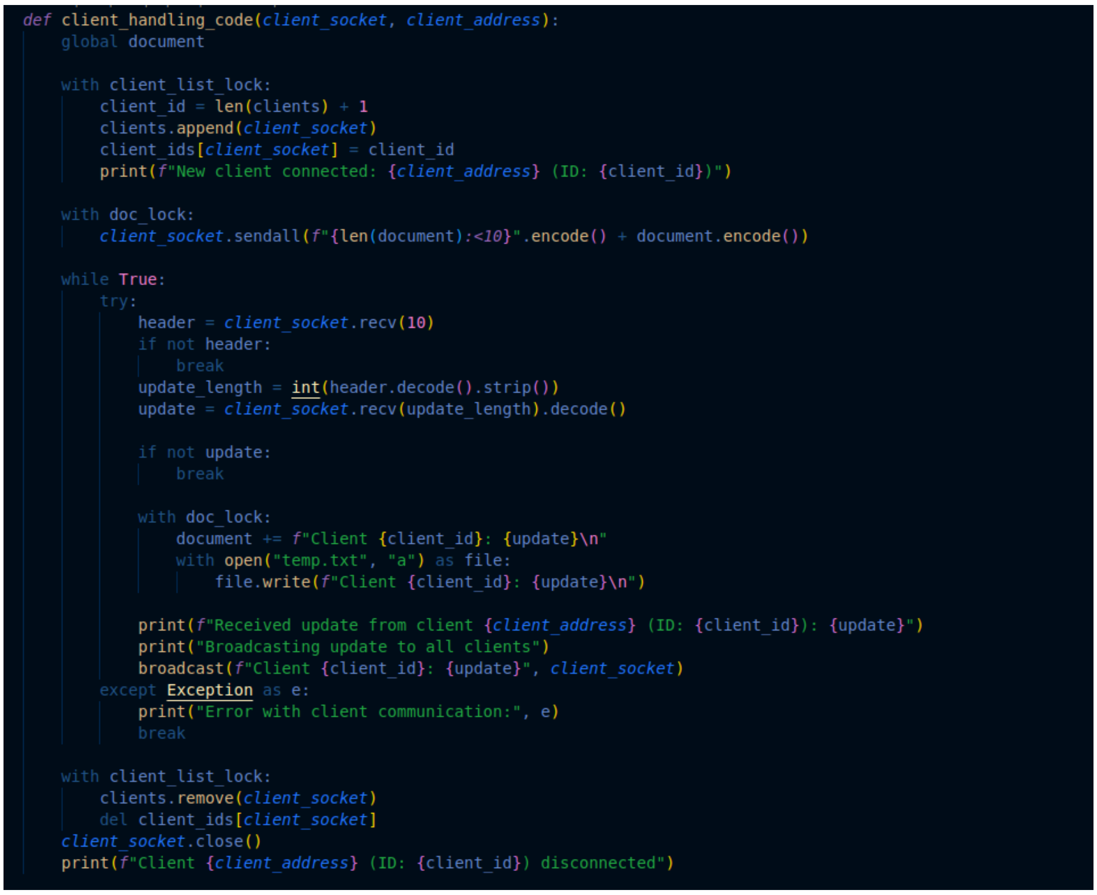
5. **Error Handling**:

- Basic error-handling mechanisms are in place to manage issues with connection and data transmission, providing error messages in case of failures.



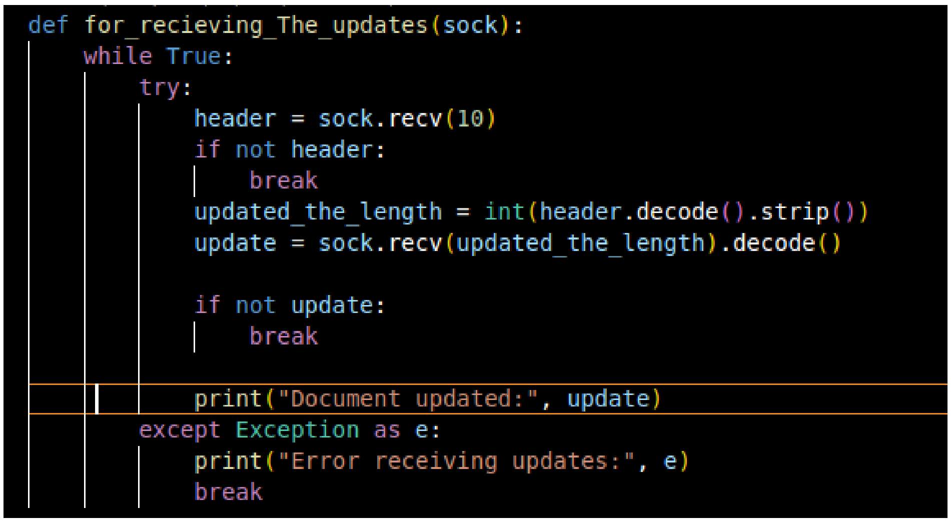
```
import socket
import threading
from threading import Lock

HOST = 'localhost'
PORT = 8080
MAX_CLIENTS = 10

document = ""
clients = []
client_ids = {}
doc_lock = Lock()
client_list_lock = Lock()

try:
    with open("temp.txt", "r") as file:
        document = file.read()
except FileNotFoundError:
    print("temp.txt not found. Starting with an empty document.")
except Exception as e:
    print("Error opening file temp.txt:", e)

Tabnine | Edit | Test | Explain | Document | Ask
def broadcast(update, sender_socket=None):
    with client_list_lock:
        for client in clients:
            if client != sender_socket:
                try:
                    message = f"{len(update):<10}" + update
                    client.sendall(message.encode())
                except Exception as e:
                    print("Error sending update:", e)
```

Figure 2: server code



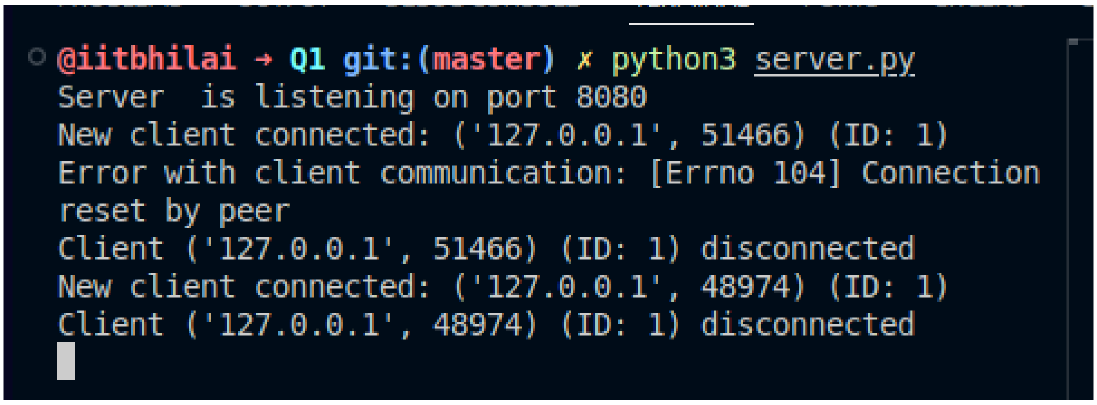
```python
def client_handling_code(client_socket, client_address):
    global document

    with client_list_lock:
        client_id = len(clients) + 1
        clients.append(client_socket)
        client_ids[client_socket] = client_id
        print(f"New client connected: {client_address} (ID: {client_id})")

    with doc_lock:
        client_socket.sendall(f"{len(document):<10}".encode() + document.encode())

    while True:
        try:
            header = client_socket.recv(10)
            if not header:
                break
            update_length = int(header.decode().strip())
            update = client_socket.recv(update_length).decode()

            if not update:
                break

            with doc_lock:
                document += f"Client {client_id}: {update}\n"
                with open("temp.txt", "a") as file:
                    file.write(f"Client {client_id}: {update}\n")

            print(f"Received update from client {client_address} (ID: {client_id}): {update}")
            print("Broadcasting update to all clients")
            broadcast(f"Client {client_id}: {update}", client_socket)
        except Exception as e:
            print("Error with client communication:", e)
            break

    with client_list_lock:
        clients.remove(client_socket)
        del client_ids[client_socket]
    client_socket.close()
    print(f"Client {client_address} (ID: {client_id}) disconnected")
```

Figure 3: client handling function code



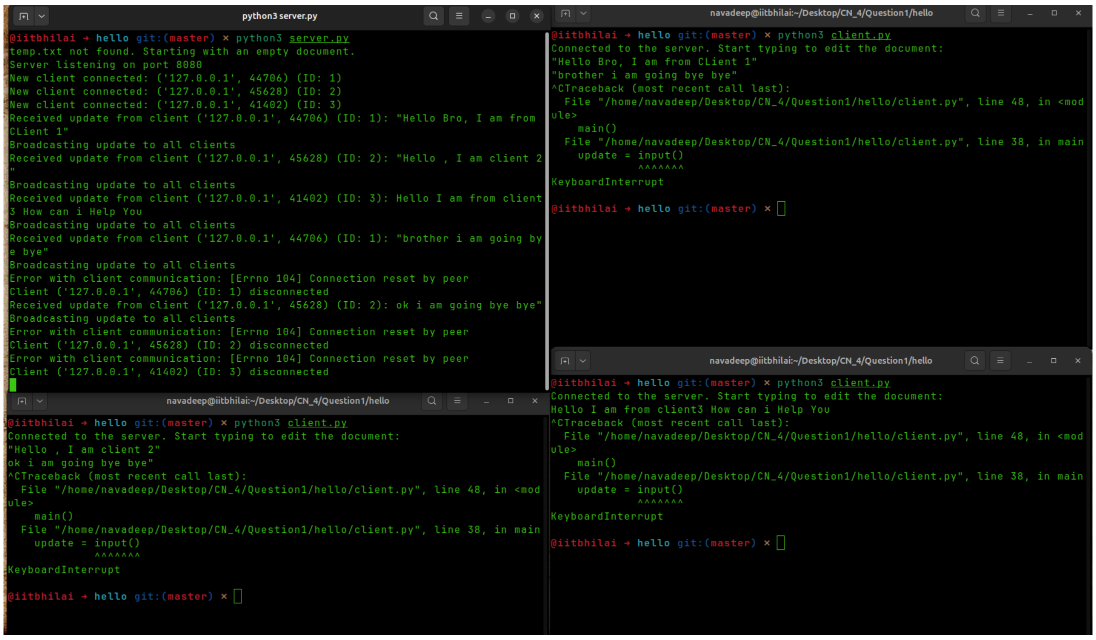
```python
def for_recieving_The_updates(sock):
    while True:
        try:
            header = sock.recv(10)
            if not header:
                break
            updated_the_length = int(header.decode().strip())
            update = sock.recv(updated_the_length).decode()

            if not update:
                break

            print("Document updated:", update)
        except Exception as e:
            print("Error receiving updates:", e)
            break
```

Figure 4: client code



```
@iitbhilai → Q1 git:(master) ✗ python3 server.py
Server  is listening on port 8080
New client connected: ('127.0.0.1', 51466) (ID: 1)
Error with client communication: [Errno 104] Connection
reset by peer
Client ('127.0.0.1', 51466) (ID: 1) disconnected
New client connected: ('127.0.0.1', 48974) (ID: 1)
Client ('127.0.0.1', 48974) (ID: 1) disconnected
```

Figure 5: connection-disconnection output



```
@iitbhilai → hello git:(master) ✗ python3 server.py
temp.txt not found. Starting with an empty document.
Server listening on port 8080
New client connected: ('127.0.0.1', 44706) (ID: 1)
New client connected: ('127.0.0.1', 45628) (ID: 2)
New client connected: ('127.0.0.1', 41402) (ID: 3)
Received update from client ('127.0.0.1', 44706) (ID: 1): "Hello Bro, I am from
CLient 1"
Broadcasting update to all clients
Received update from client ('127.0.0.1', 45628) (ID: 2): "Hello , I am client 2
"
Broadcasting update to all clients
Received update from client ('127.0.0.1', 41402) (ID: 3): Hello I am from client
3 How can i Help You
Broadcasting update to all clients
Received update from client ('127.0.0.1', 44706) (ID: 1): "brother i am going by
e bye"
Broadcasting update to all clients
Error with client communication: [Errno 104] Connection reset by peer
Client ('127.0.0.1', 44706) (ID: 1) disconnected
Received update from client ('127.0.0.1', 45628) (ID: 2): ok i am going bye bye"
Broadcasting update to all clients
Error with client communication: [Errno 104] Connection reset by peer
Client ('127.0.0.1', 45628) (ID: 2) disconnected
Error with client communication: [Errno 104] Connection reset by peer
Client ('127.0.0.1', 41402) (ID: 3) disconnected
```

```
@iitbhilai → hello git:(master) ✗ python3 client.py
Connected to the server. Start typing to edit the document:
"Hello Bro, I am from CLient 1"
"brother i am going bye bye"
^CTraceback (most recent call last):
  File "/home/navadeep/Desktop/CN_4/Question1/hello/client.py", line 48, in <mod
ule>
    main()
  File "/home/navadeep/Desktop/CN_4/Question1/hello/client.py", line 38, in main
    update = input()
             ^^^^^^^
KeyboardInterrupt

@iitbhilai → hello git:(master) ✗
```

```
@iitbhilai → hello git:(master) ✗ python3 client.py
Connected to the server. Start typing to edit the document:
"Hello , I am client 2"
ok i am going bye bye"
^CTraceback (most recent call last):
  File "/home/navadeep/Desktop/CN_4/Question1/hello/client.py", line 48, in <mod
ule>
    main()
  File "/home/navadeep/Desktop/CN_4/Question1/hello/client.py", line 38, in main
    update = input()
             ^^^^^^^
KeyboardInterrupt

@iitbhilai → hello git:(master) ✗
```

```
@iitbhilai → hello git:(master) ✗ python3 client.py
Connected to the server. Start typing to edit the document:
Hello I am from client3 How can i Help You
^CTraceback (most recent call last):
  File "/home/navadeep/Desktop/CN_4/Question1/hello/client.py", line 48, in <mod
ule>
    main()
  File "/home/navadeep/Desktop/CN_4/Question1/hello/client.py", line 38, in main
    update = input()
             ^^^^^^^
KeyboardInterrupt

@iitbhilai → hello git:(master) ✗
```

Figure 6: Output for question 1

Server Responsibilities

1. **Listen for Incoming Client Connections:**

   The server listens on a specific port (8080) for incoming client connections. It handles up to 10 clients concurrently by spawning a new thread for each client.

2. **Manage a Shared Document:** The server maintains the document in memory, which is shared among all connected clients. It holds the current state of the document and is responsible for broadcasting any updates to all connected clients.

3. **Handle Concurrent Edits:** The server uses synchronization mechanisms such as mutex locks to ensure that the document is edited safely, preventing race conditions when multiple clients attempt to update the document at the same time.

4. **Broadcast Updates:** When a client sends an update, the server appends the change to the document in memory and saves it to `temp.txt` to ensure persistence. It then broadcasts the update to all other connected clients.
5. **Persist Document Changes:** All document updates are appended to the `temp.txt` file to ensure that the document is saved and can be restored in case the server stops or restarts.
6. **Handle Client Disconnects:** When a client disconnects, the server removes the client from the active list and prints the client's disconnection message.

Server Error Handling

The server handles errors in the following scenarios:

- **Server Startup Errors:** If the server cannot create or bind the socket to port 8080, it prints an error message: `"Server error: Unable to start server on port 8080"`.
- **File Access Errors:** If there is an issue opening the `temp.txt` file for writing, the server prints: `"Error opening file temp.txt"`.

1 Client Responsibilities

The client performs the following tasks to interact with the server and contribute to the collaborative editing process:

1. **Connect to the Server:** The client attempts to connect to the server using the server's IP address and port (8080). Upon successful connection, the client prints: `"Connected to the server. Start typing to edit the document:"`.
2. **Receive Document Updates:** The client continuously listens for updates from the server. When a new update is received, the client prints: `"Document updated: <received_content>"` to display the latest version of the document.
3. **Send Edits to the Server:** The client allows the user to input text to edit the document. The client sends the edited text back to the server for processing and broadcasting to other clients.

Client Error Handling

The client handles errors such as connection failures. If the client cannot connect to the server, it displays: `"Connection failed. Please check the server and try again."`

Concurrency and Synchronization

- **Concurrency:** The server handles multiple client connections concurrently by spawning a new thread for each client. Each client can independently send updates and receive changes without blocking other clients. This enables multiple users to edit the document simultaneously.

- **Synchronization:** To prevent data inconsistencies, the server uses *mutex locks* to synchronize access to the shared document. This ensures that only one client can update the document at any given time, preventing race conditions and ensuring data integrity.

Document Persistence

To ensure the document persists even if the server crashes or restarts:

- Each update from the client is appended to a file named `temp.txt`.
- On server restart, the server can load the contents of `temp.txt` to restore the document to its last saved state.

Message Flow

1. **Server Starts:** The server listens on port 8080. If successful, it prints `"Server listening on port 8080"`.
2. **Client Connects:** A client connects to the server, and the server sends the current state of the document.
3. **Client Edits and Sends Updates:** The client makes edits and sends the updated text to the server. The server processes the update, saves it to `temp.txt`, and broadcasts it to all connected clients.
4. **Clients Receive Updates:** All clients receive the broadcasted update and display the new version of the document.

Question 2

Note : In this Question 2 folder there The server will on listen for incoming TCP connections on port `12345`. It is used to handle both `IPv4` and `IPv6` addresses.

The server runs asynchronously, means that it can accept and handle multiple client connections simultaneously without blocking.

Client Handling

When a client connects, the server assigns a coroutine to manage that connection. The `asyncio` library ensures that the server can handle multiple clients concurrently, so that itcan allow to read from and write to clients in a non-blocking manner. This avoids delays, as the server does not wait for one client to finish before serving the next.

Reading and Writing Data

Once the client is connected, the server reads data sent by the client, it will logs the received message, and then sends the same data back to the client (echoing it). The reading and writing of data is done asynchronously, which means the server doesn't block waiting for data. It can continue handling other tasks while waiting for input or output operations to complete.

Logging

Throughout the server's operation, various events are logged using Python's `logging` module. This includes:

- When a client connects.
- The data received from the client.
- The data sent back (echoed) to the client.
- Any errors or unexpected events that occur during communication.

Connection Management

The server will continue to handle communication with clients until the client disconnects or sends no data. If any error occurs, the server logs the error and gracefully closes the connection with the client. It ensures proper cleanup and avoids leaving open connections.

Asynchronous Execution

The core of the server is built around `asyncio`, which allows it to perform asynchronous operations. This means the server is capable of handling multiple clients simultaneously:

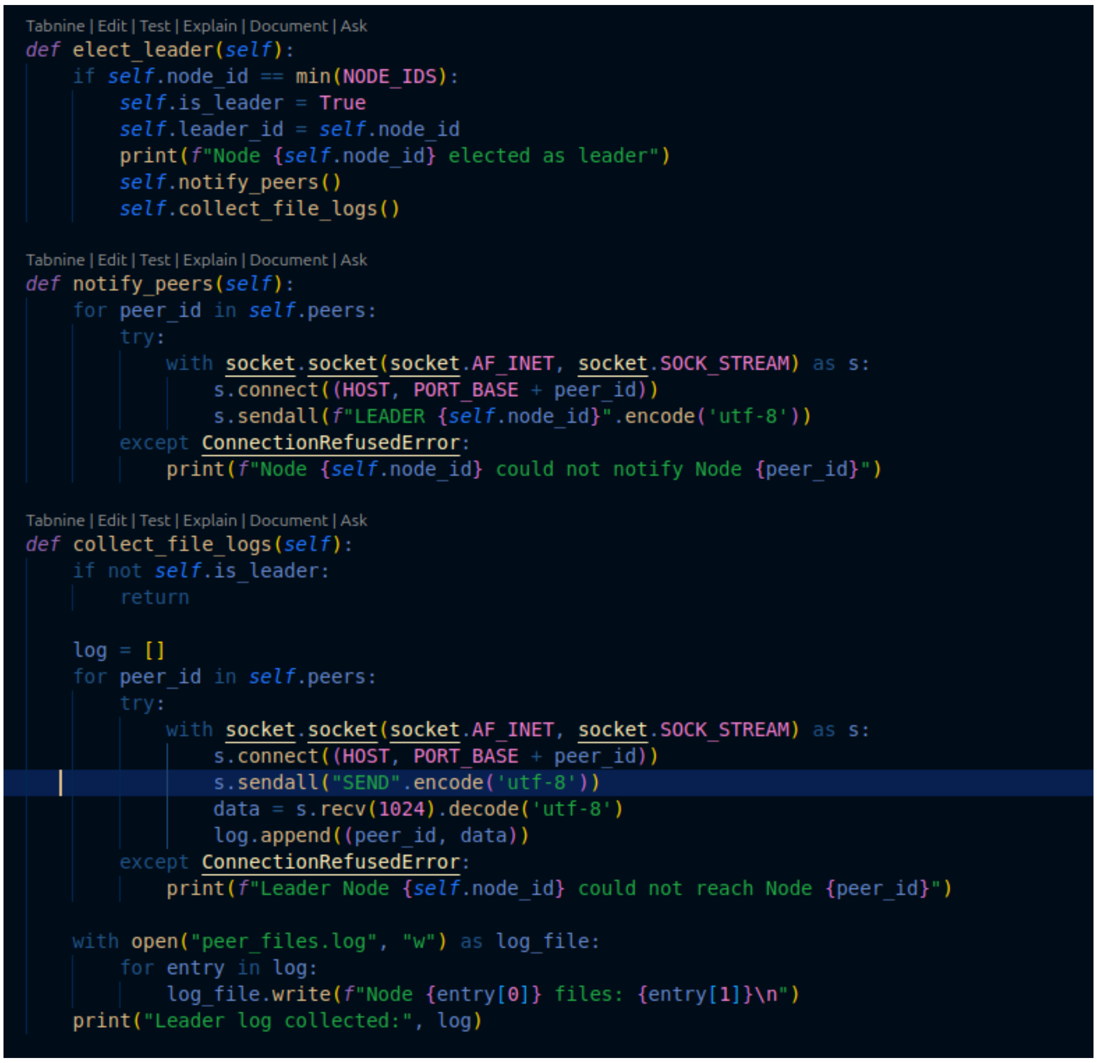
- It reads and writes data to clients asynchronously, meaning it does not block or wait for one client to finish before serving another.

- The `await` keyword is used to pause the execution of a task until it is completed, allowing the server to switch between tasks efficiently without blocking other tasks.

Key Features

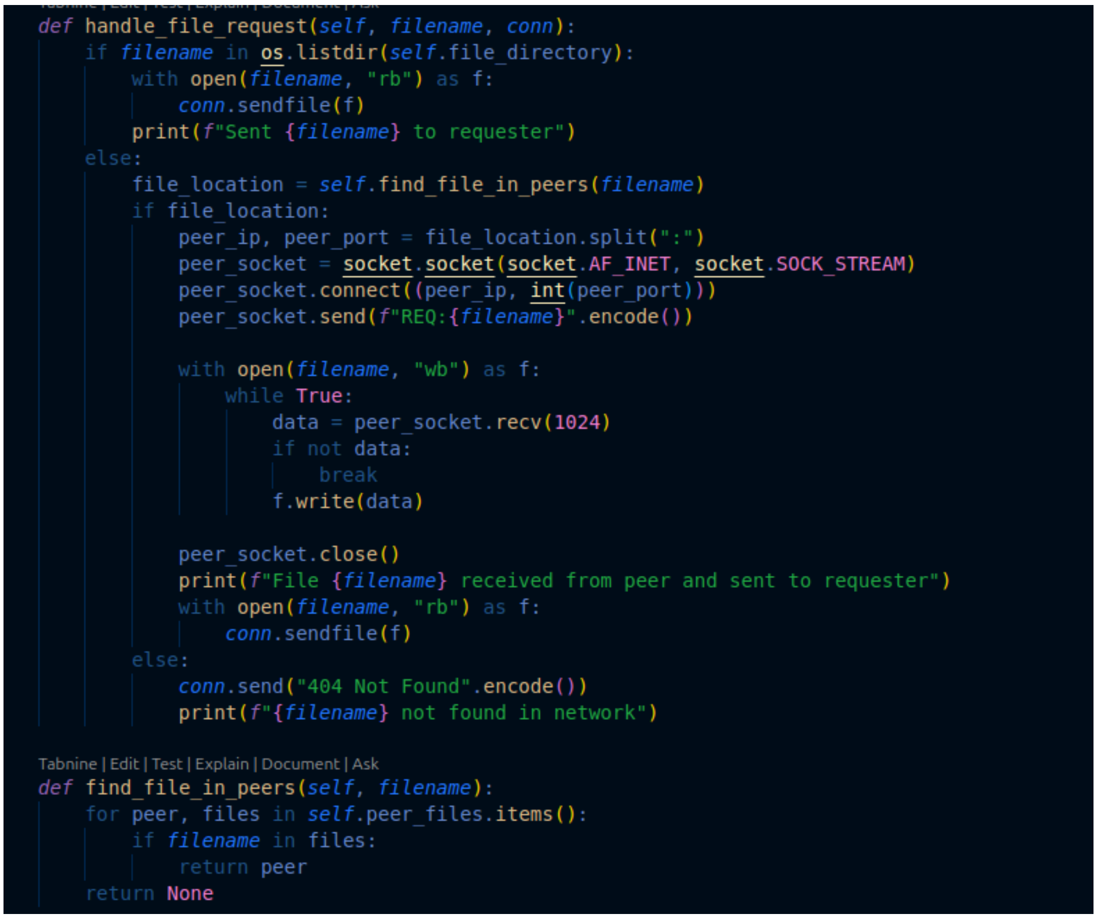
The server is designed with the following key features:

- **Non-blocking behavior**: It means that the server can handle multiple clients at the same time without blocking.
- **Echo server**: The server simply echoes back whatever message that client sends.
- **Logging**: Every action taken by the server is logged, like the incoming connections and data exchanges.
- **Graceful shutdown**: The server handles errors and ensures thst connections are properly closed.



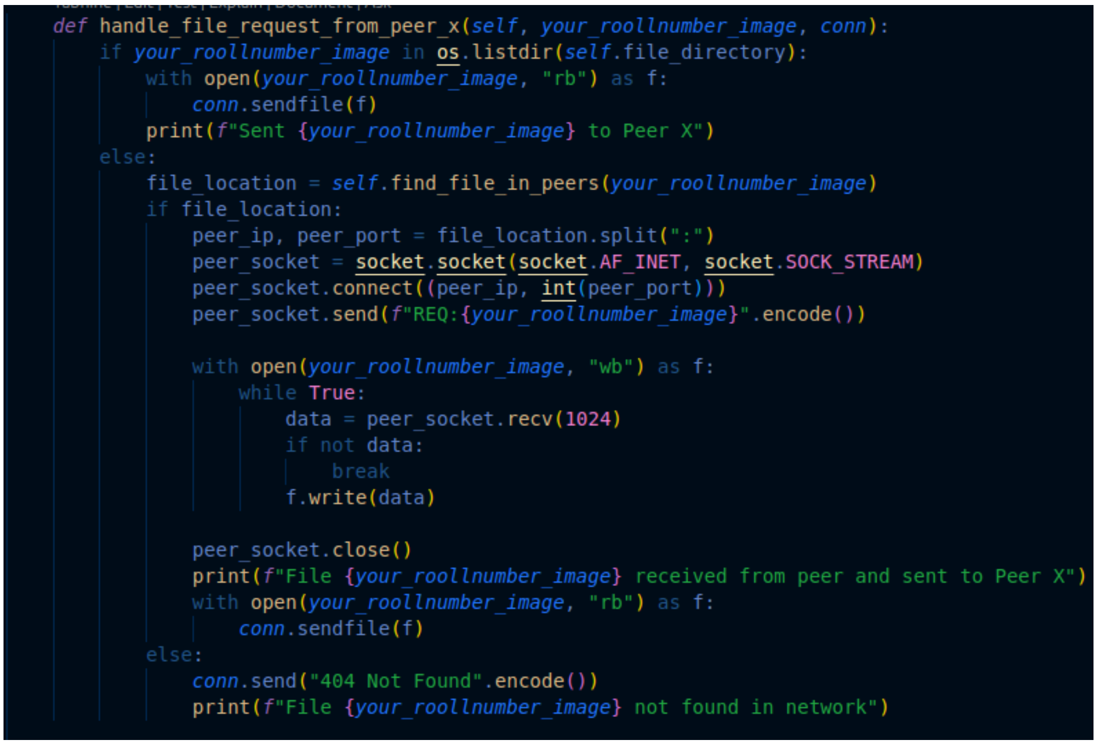
```python
def elect_leader(self):
    if self.node_id == min(NODE_IDS):
        self.is_leader = True
        self.leader_id = self.node_id
        print(f"Node {self.node_id} elected as leader")
        self.notify_peers()
        self.collect_file_logs()

def notify_peers(self):
    for peer_id in self.peers:
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((HOST, PORT_BASE + peer_id))
                s.sendall(f"LEADER {self.node_id}".encode('utf-8'))
        except ConnectionRefusedError:
            print(f"Node {self.node_id} could not notify Node {peer_id}")

def collect_file_logs(self):
    if not self.is_leader:
        return

    log = []
    for peer_id in self.peers:
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((HOST, PORT_BASE + peer_id))
                s.sendall("SEND".encode('utf-8'))
                data = s.recv(1024).decode('utf-8')
                log.append((peer_id, data))
        except ConnectionRefusedError:
            print(f"Leader Node {self.node_id} could not reach Node {peer_id}")

    with open("peer_files.log", "w") as log_file:
        for entry in log:
            log_file.write(f"Node {entry[0]} files: {entry[1]}\n")
    print("Leader log collected:", log)
```

Figure 7: Output for question 2



```
def handle_file_request(self, filename, conn):
    if filename in os.listdir(self.file_directory):
        with open(filename, "rb") as f:
            conn.sendfile(f)
        print(f"Sent {filename} to requester")
    else:
        file_location = self.find_file_in_peers(filename)
        if file_location:
            peer_ip, peer_port = file_location.split(":")
            peer_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            peer_socket.connect((peer_ip, int(peer_port)))
            peer_socket.send(f"REQ:{filename}".encode())

            with open(filename, "wb") as f:
                while True:
                    data = peer_socket.recv(1024)
                    if not data:
                        break
                    f.write(data)

            peer_socket.close()
            print(f"File {filename} received from peer and sent to requester")
            with open(filename, "rb") as f:
                conn.sendfile(f)
        else:
            conn.send("404 Not Found".encode())
            print(f"{filename} not found in network")

Tabnine | Edit | Test | Explain | Document | Ask
def find_file_in_peers(self, filename):
    for peer, files in self.peer_files.items():
        if filename in files:
            return peer
    return None
```

Figure 8: Output for question 2



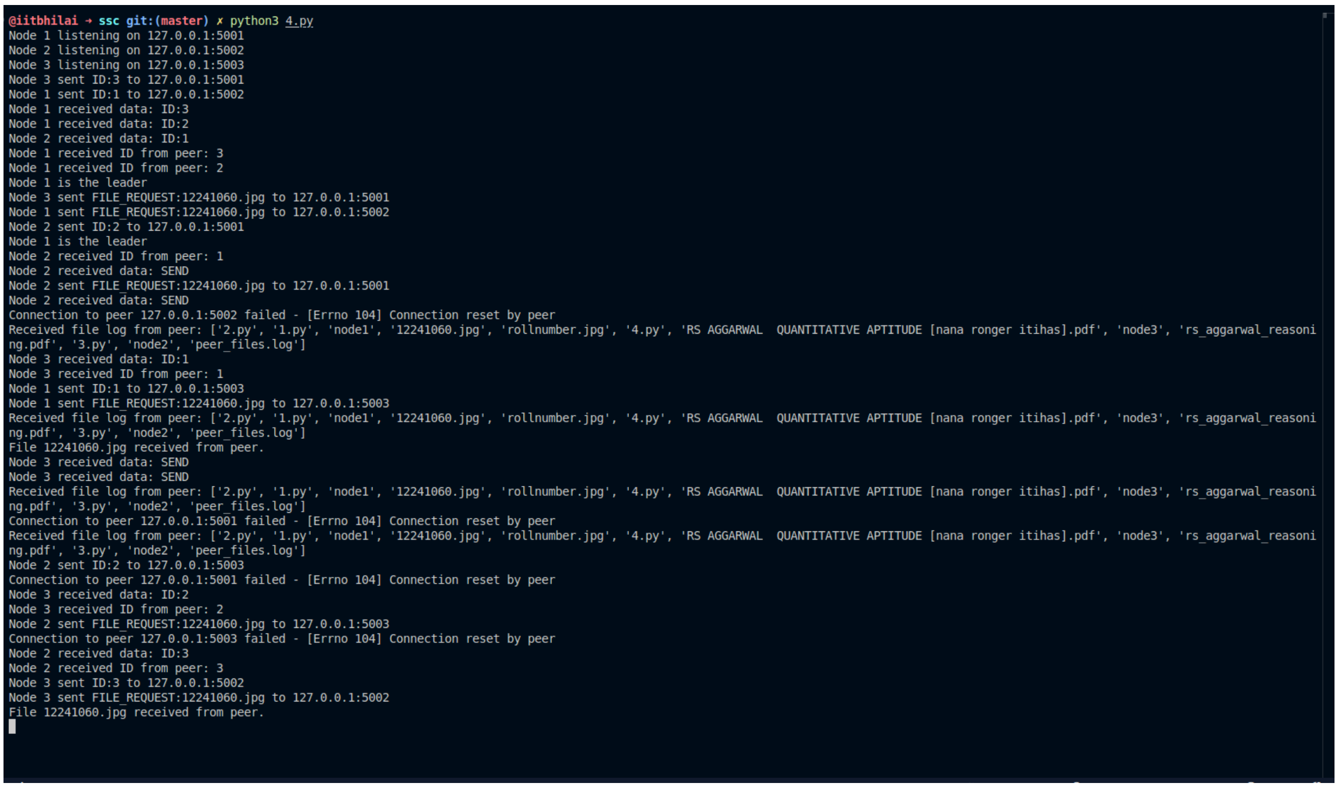
```
def handle_file_request_from_peer_x(self, your_roollnumber_image, conn):
    if your_roollnumber_image in os.listdir(self.file_directory):
        with open(your_roollnumber_image, "rb") as f:
            conn.sendfile(f)
        print(f"Sent {your_roollnumber_image} to Peer X")
    else:
        file_location = self.find_file_in_peers(your_roollnumber_image)
        if file_location:
            peer_ip, peer_port = file_location.split(":")
            peer_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            peer_socket.connect((peer_ip, int(peer_port)))
            peer_socket.send(f"REQ:{your_roollnumber_image}".encode())

            with open(your_roollnumber_image, "wb") as f:
                while True:
                    data = peer_socket.recv(1024)
                    if not data:
                        break
                    f.write(data)

            peer_socket.close()
            print(f"File {your_roollnumber_image} received from peer and sent to Peer X")
            with open(your_roollnumber_image, "rb") as f:
                conn.sendfile(f)
        else:
            conn.send("404 Not Found".encode())
            print(f"File {your_roollnumber_image} not found in network")
```

Figure 9: question 2 part 2 code

File Transfer:

- If Peer Y hosts the file, then the leader requests for file from Peer Y and it will forwards it to Peer X.
- when the leader hosts the file, then it directly sends it to Peer X.
- If the file is not found, the leader responds with `404 Not Found`.

Concurrency: Multi-threading used to handle files simentenuously



```
@iitbhilai → ssc git:(master) ✗ python3 4.py
Node 1 listening on 127.0.0.1:5001
Node 2 listening on 127.0.0.1:5002
Node 3 listening on 127.0.0.1:5003
Node 3 sent ID:3 to 127.0.0.1:5001
Node 1 sent ID:1 to 127.0.0.1:5002
Node 1 received data: ID:3
Node 1 received data: ID:2
Node 2 received data: ID:1
Node 1 received ID from peer: 3
Node 1 received ID from peer: 2
Node 1 is the leader
Node 3 sent FILE_REQUEST:12241060.jpg to 127.0.0.1:5001
Node 1 sent FILE_REQUEST:12241060.jpg to 127.0.0.1:5002
Node 2 sent ID:2 to 127.0.0.1:5001
Node 1 is the leader
Node 2 received ID from peer: 1
Node 2 received data: SEND
Node 2 sent FILE_REQUEST:12241060.jpg to 127.0.0.1:5001
Node 2 received data: SEND
Connection to peer 127.0.0.1:5002 failed - [Errno 104] Connection reset by peer
Received file log from peer: ['2.py', '1.py', 'node1', '12241060.jpg', 'rollnumber.jpg', '4.py', 'RS AGGARWAL  QUANTITATIVE APTITUDE [nana ronger itihas].pdf', 'node3', 'rs_aggarwal_reasoni
ng.pdf', '3.py', 'node2', 'peer_files.log']
Node 3 received data: ID:1
Node 3 received ID from peer: 1
Node 1 sent ID:1 to 127.0.0.1:5003
Node 1 sent FILE_REQUEST:12241060.jpg to 127.0.0.1:5003
Received file log from peer: ['2.py', '1.py', 'node1', '12241060.jpg', 'rollnumber.jpg', '4.py', 'RS AGGARWAL  QUANTITATIVE APTITUDE [nana ronger itihas].pdf', 'node3', 'rs_aggarwal_reasoni
ng.pdf', '3.py', 'node2', 'peer_files.log']
File 12241060.jpg received from peer.
Node 3 received data: SEND
Node 3 received data: SEND
Received file log from peer: ['2.py', '1.py', 'node1', '12241060.jpg', 'rollnumber.jpg', '4.py', 'RS AGGARWAL  QUANTITATIVE APTITUDE [nana ronger itihas].pdf', 'node3', 'rs_aggarwal_reasoni
ng.pdf', '3.py', 'node2', 'peer_files.log']
Connection to peer 127.0.0.1:5001 failed - [Errno 104] Connection reset by peer
Received file log from peer: ['2.py', '1.py', 'node1', '12241060.jpg', 'rollnumber.jpg', '4.py', 'RS AGGARWAL  QUANTITATIVE APTITUDE [nana ronger itihas].pdf', 'node3', 'rs_aggarwal_reasoni
ng.pdf', '3.py', 'node2', 'peer_files.log']
Node 2 sent ID:2 to 127.0.0.1:5003
Connection to peer 127.0.0.1:5001 failed - [Errno 104] Connection reset by peer
Node 3 received data: ID:2
Node 3 received ID from peer: 2
Node 2 sent FILE_REQUEST:12241060.jpg to 127.0.0.1:5003
Connection to peer 127.0.0.1:5003 failed - [Errno 104] Connection reset by peer
Node 2 received data: ID:3
Node 2 received ID from peer: 3
Node 3 sent ID:3 to 127.0.0.1:5002
Node 3 sent FILE_REQUEST:12241060.jpg to 127.0.0.1:5002
File 12241060.jpg received from peer.
```

Figure 10: Final output for whole question 2

Question 3

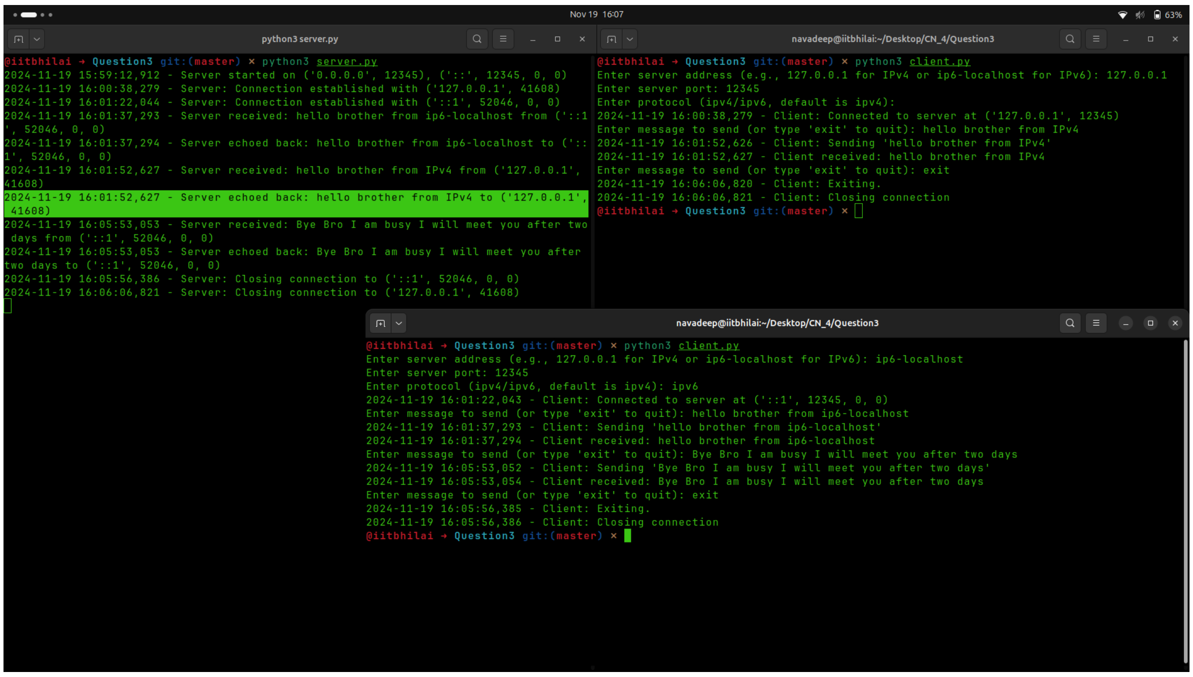
1. **Imports and Logging Configuration**:
   - In client code we will imports the `asyncio`, `socket`, and `logging` modules.
   - Logging is configured using `lg.basicConfig` with a log level of `INFO` and a simple format.
2. `client_echo_using_tcp` **Coroutine**:
   - This one is main synchronous function which is useful for handling the TCP connection
   - **Host, Port, and Protocol**:
     - The user specifies the server's `host` and `port`.
     - The `protocol` parameter defaults to `ipv4`, but if user want to choose `ipv6` then using ip6-localhost they can do.
   - **Address Information Resolution**:
     - Depending on the protocol, `socket.getaddrinfo` retrieves address information. `AF_INET` and `AF_INET6` specify IPv4 and IPv6 families respectively.
   - **Attempting Connection**:
     - This function iterates all the addresses returned by `getaddrinfo`.
     - first it tries to open a connection using `asyncio.open_connection`.
     - Upon success, it logs the connection establishment.
     - If a connection fails, then it tries to attempts to use other addresses until all options are exhausted.
3. **Message Sending and Receiving Loop**:
   - The client enters a loop, accepting the user input:
     - If the user want to exit from that server then if they types "exit", the loop terminates and the connection closes.
     - Otherwise, the input message is encoded and sent to the server using `writer.write()`.
     - The `await writer.drain()` ensures data is flushed through the socket.
   - The client then waits to receive a response using `await reader.read(1024)` and logs the received data.
   - The loop continues until the user decides to exit.
4. **Error Handling**:
   - The code handles `ConnectionRefusedError` and `OSError` to manage connection issues and retries other addresses as needed.
   - It will handle all the errors mentioned in the question.

User Input and Script Execution

In final part of code it prompts the user for the server's IP address, port, and protocol type. The function `aio.run(client_echo_using_tcp(host, port, protocol))` runs the asynchronous client function.

Asynchronous Operation

- The `asyncio` library enables non-blocking I/O, allowing the client to send and receive messages asynchronously.
- The use of `await` pauses the execution until runs after the data gets it will resume it progress.

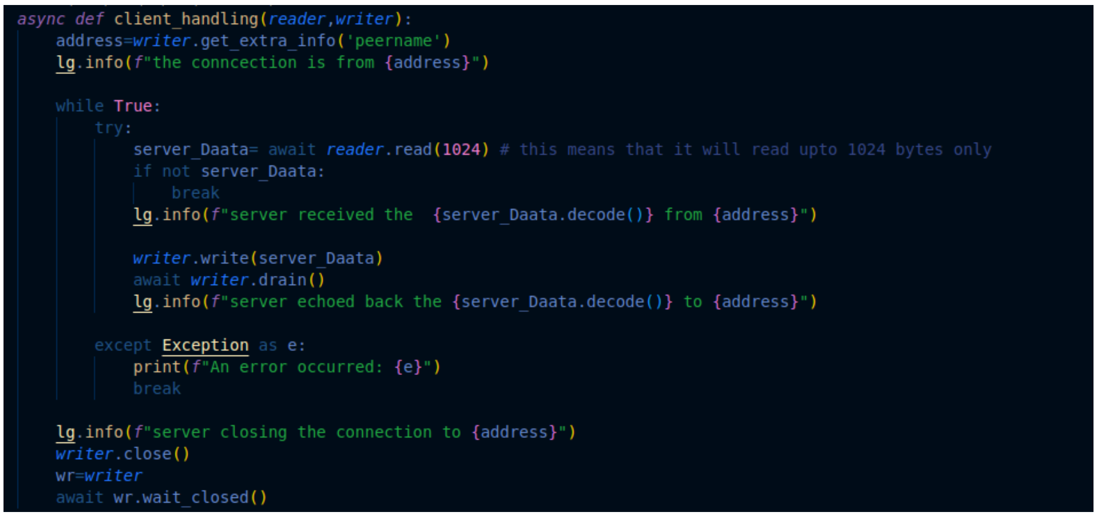

Nov 19 16:07
63%
python3 server.py
navadeep@iitbhilai:~/Desktop/CN_4/Question3
navadeep@iitbhilai:~/Desktop/CN_4/Question3

```
@iitbhilai → Question3 git:(master) ✗ python3 server.py
2024-11-19 15:59:12,912 - Server started on ('0.0.0.0', 12345), ('::', 12345, 0, 0)
2024-11-19 16:00:38,279 - Server: Connection established with ('127.0.0.1', 41608)
2024-11-19 16:01:22,044 - Server: Connection established with ('::1', 52046, 0, 0)
2024-11-19 16:01:37,293 - Server received: hello brother from ip6-localhost from ('::1
', 52046, 0, 0)
2024-11-19 16:01:37,294 - Server echoed back: hello brother from ip6-localhost to ('::
1', 52046, 0, 0)
2024-11-19 16:01:52,627 - Server received: hello brother from IPv4 from ('127.0.0.1',
41608)
2024-11-19 16:01:52,627 - Server echoed back: hello brother from IPv4 to ('127.0.0.1',
 41608)
2024-11-19 16:05:53,053 - Server received: Bye Bro I am busy I will meet you after two
 days from ('::1', 52046, 0, 0)
2024-11-19 16:05:53,053 - Server echoed back: Bye Bro I am busy I will meet you after
two days to ('::1', 52046, 0, 0)
2024-11-19 16:05:56,386 - Server: Closing connection to ('::1', 52046, 0, 0)
2024-11-19 16:06:06,821 - Server: Closing connection to ('127.0.0.1', 41608)
```

```
@iitbhilai → Question3 git:(master) ✗ python3 client.py
Enter server address (e.g., 127.0.0.1 for IPv4 or ip6-localhost for IPv6): 127.0.0.1
Enter server port: 12345
Enter protocol (ipv4/ipv6, default is ipv4):
2024-11-19 16:00:38,279 - Client: Connected to server at ('127.0.0.1', 12345)
Enter message to send (or type 'exit' to quit): hello brother from IPv4
2024-11-19 16:01:52,626 - Client: Sending 'hello brother from IPv4'
2024-11-19 16:01:52,627 - Client received: hello brother from IPv4
Enter message to send (or type 'exit' to quit): exit
2024-11-19 16:06:06,820 - Client: Exiting.
2024-11-19 16:06:06,821 - Client: Closing connection
@iitbhilai → Question3 git:(master) ✗
```

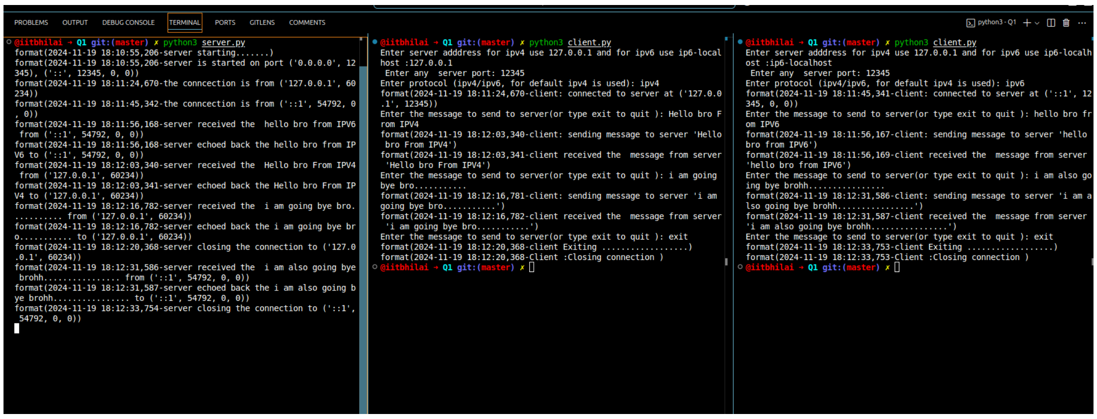
```
@iitbhilai → Question3 git:(master) ✗ python3 client.py
Enter server address (e.g., 127.0.0.1 for IPv4 or ip6-localhost for IPv6): ip6-localhost
Enter server port: 12345
Enter protocol (ipv4/ipv6, default is ipv4): ipv6
2024-11-19 16:01:22,043 - Client: Connected to server at ('::1', 12345, 0, 0)
Enter message to send (or type 'exit' to quit): hello brother from ip6-localhost
2024-11-19 16:01:37,293 - Client: Sending 'hello brother from ip6-localhost'
2024-11-19 16:01:37,294 - Client received: hello brother from ip6-localhost
Enter message to send (or type 'exit' to quit): Bye Bro I am busy I will meet you after two days
2024-11-19 16:05:53,052 - Client: Sending 'Bye Bro I am busy I will meet you after two days'
2024-11-19 16:05:53,054 - Client received: Bye Bro I am busy I will meet you after two days
Enter message to send (or type 'exit' to quit): exit
2024-11-19 16:05:56,385 - Client: Exiting.
2024-11-19 16:05:56,386 - Client: Closing connection
@iitbhilai → Question3 git:(master) ✗
```

Figure 11: Output for third scenario



```python
async def client_handling(reader,writer):
    address=writer.get_extra_info('peername')
    lg.info(f"the conncection is from {address}")

    while True:
        try:
            server_Daata= await reader.read(1024) # this means that it will read upto 1024 bytes only
            if not server_Daata:
                break
            lg.info(f"server received the  {server_Daata.decode()} from {address}")

            writer.write(server_Daata)
            await writer.drain()
            lg.info(f"server echoed back the {server_Daata.decode()} to {address}")

        except Exception as e:
            print(f"An error occurred: {e}")
            break

    lg.info(f"server closing the connection to {address}")
    writer.close()
    wr=writer
    await wr.wait_closed()
```

Figure 12: function code using client_handling



```
@iitbhilai → Q1 git:(master) ✗ python3 server.py
format(2024-11-19 18:10:55,206-server starting.......)
format(2024-11-19 18:10:55,206-server is started on port ('0.0.0.0', 12
345), ('::', 12345, 0, 0))
format(2024-11-19 18:11:24,670-the conncection is from ('127.0.0.1', 60
234))
format(2024-11-19 18:11:45,342-the conncection is from ('::1', 54792, 0
, 0))
format(2024-11-19 18:11:56,168-server received the  hello bro from IPV6
 from ('::1', 54792, 0, 0))
format(2024-11-19 18:11:56,168-server echoed back the hello bro from IP
V6 to ('::1', 54792, 0, 0))
format(2024-11-19 18:12:03,340-server received the  Hello bro From IPV4
 from ('127.0.0.1', 60234))
format(2024-11-19 18:12:03,341-server echoed back the Hello bro From IP
V4 to ('127.0.0.1', 60234))
format(2024-11-19 18:12:16,782-server received the  i am going bye bro.
.......... from ('127.0.0.1', 60234))
format(2024-11-19 18:12:16,782-server echoed back the i am going bye br
o........... to ('127.0.0.1', 60234))
format(2024-11-19 18:12:20,368-server closing the connection to ('127.0
.0.1', 60234))
format(2024-11-19 18:12:31,586-server received the  i am also going bye
 brohh................ from ('::1', 54792, 0, 0))
format(2024-11-19 18:12:31,587-server echoed back the i am also going b
ye brohh................ to ('::1', 54792, 0, 0))
format(2024-11-19 18:12:33,754-server closing the connection to ('::1',
 54792, 0, 0))
```

```
@iitbhilai → Q1 git:(master) ✗ python3 client.py
Enter server adddress for ipv4 use 127.0.0.1 and for ipv6 use ip6-local
host :127.0.0.1
 Enter any  server port: 12345
Enter protocol (ipv4/ipv6, for default ipv4 is used): ipv4
format(2024-11-19 18:11:24,670-client: connected to server at ('127.0.0
.1', 12345))
Enter the message to send to server(or type exit to quit ): Hello bro F
rom IPV4
format(2024-11-19 18:12:03,340-client: sending message to server 'Hello
 bro From IPV4')
format(2024-11-19 18:12:03,341-client received the  message from server
 'Hello bro From IPV4')
Enter the message to send to server(or type exit to quit ): i am going
bye bro...........
format(2024-11-19 18:12:16,781-client: sending message to server 'i am
going bye bro...........')
format(2024-11-19 18:12:16,782-client received the  message from server
 'i am going bye bro...........')
Enter the message to send to server(or type exit to quit ): exit
format(2024-11-19 18:12:20,368-client Exiting ..................)
format(2024-11-19 18:12:20,368-Client :Closing connection )
@iitbhilai → Q1 git:(master) ✗
```

```
@iitbhilai → Q1 git:(master) ✗ python3 client.py
Enter server adddress for ipv4 use 127.0.0.1 and for ipv6 use ip6-localh
ost :ip6-localhost
 Enter any  server port: 12345
Enter protocol (ipv4/ipv6, for default ipv4 is used): ipv6
format(2024-11-19 18:11:45,341-client: connected to server at ('::1', 12
345, 0, 0))
Enter the message to send to server(or type exit to quit ): hello bro fr
om IPV6
format(2024-11-19 18:11:56,167-client: sending message to server 'hello
bro from IPV6')
format(2024-11-19 18:11:56,169-client received the  message from server
'hello bro from IPV6')
Enter the message to send to server(or type exit to quit ): i am also go
ing bye brohh................
format(2024-11-19 18:12:31,586-client: sending message to server 'i am a
lso going bye brohh................')
format(2024-11-19 18:12:31,587-client received the  message from server
'i am also going bye brohh................')
Enter the message to send to server(or type exit to quit ): exit
format(2024-11-19 18:12:33,753-client Exiting ..................)
format(2024-11-19 18:12:33,753-Client :Closing connection )
@iitbhilai → Q1 git:(master) ✗
```

Figure 13: Final output for Question 3

This function `client_handling` is an asynchronous server-side handler.It is used to manage client connections.

1. **Accepting and Logging the Client Connection:**

   - The function retrieves the address (IP and port) of the client that has connected using `writer.get_extra_info('peername')`.
   - The server logs the client's address, allowing it to track the origin of the connection.

2. **Continuous Listening for Data:**

   - The function enters an infinite loop with `while True` to continuously listen for data from the client.
   - It reads up to 1024 bytes of incoming data from the client using `await reader.read(1024)`.

3. **Processing the Received Data:**

- When the server receives data successfully, then it will decodes the byte data into a string.
- The server logs the received data for debugging .

4. **Echoing the Data Back to the Client:**

    - After processing the data, the server immediately sends the exact same data back to the client (echoes it) using `writer.write(server_data)`.
    - The function waits for the data to be fully written to the client with `await writer.drain()`, ensuring no data is lost.

5. **Handling Errors:**

    - I used try ,expect method for if any error occurs during data writing or reading then this expect will handle the errors.
    - The error message will be printed , and the loop will break and connection is closed .

6. **Closing the Connection:**

    - Once the loop terminates (due to client disconnection or an error), then server logs that it is closing the connection.
    - The server then calls `writer.close()` to close the connection and waits for the closure to complete with `await writer.wait_closed()`.