TCP Server Script

The **TCP server script** establishes a multi-client chat server using Python's socket and threading modules. It listens for incoming TCP connections on a specified IP and port, accepts clients, and spawns a dedicated thread for each to handle message reception. Clients send their usernames upon connecting, and the server broadcasts all incoming messages to all connected users using a shared list of active clients. It gracefully handles disconnections and errors by removing clients and closing sockets. A shutdown function ensures that all sockets are closed properly when the server is stopped manually (e.g., via Ctrl+C), allowing for clean resource release.

Here is a detailed explanation for the TCP server script of the multi-client chat application.

Imports and Constants

These are modules we imported for the TCP server script

```
import signal
import sys
import threading
from socket import AF_INET, SOCK_STREAM, socket
```

- **signal**: Allows handling OS signals (e.g., SIGINT for Ctrl+C shutdown).
- **sys**: Used to exit the program cleanly.
- **threading**: Enables handling multiple clients simultaneously using threads.
- **socket**: The core library for implementing socket programming network connections.
    - **AF_INET**: Specifies the IPv4 address family.
    - **SOCK_STREAM**: Specifies a TCP (connection-oriented) socket.
    - **socket()**: Creates a new socket instance.

The following constants and global variables are used in the script.

```
SERVER_IP = "127.0.0.1"  # Standard loopback interface address (localhost)
SERVER_PORT = 65432  # Port to listen on, any non-privileged port > 1023 will do
LISTENER_LIMIT = 5  # Number of clients to litsen to concurrently
active_clients = []  # List of all connected users with the format (username, client)

# NOTE
# Any client wanting to connect to this server must use the above IP address and port number
```

- **SERVER_IP = "127.0.0.1"**: The server runs on localhost (loopback address).
- **SERVER_PORT = 65432**: Port number for the server to listen on.
- **LISTENER_LIMIT = 5**: Maximum number of queued client connections.
- **active_clients = []**: Keeps track of connected users.

There is also a server global variable used to hold the server socket object in the main function. We use this socket object to establish a connection with clients' sockets.

```
def main():
    """Server main function"""
    global server  # Declare server as global so we can close it in signal handler

    server = socket(AF_INET, SOCK_STREAM)
```

Removing Disconnected Clients

The `remove_client()` function is used to remove a client from the `active_clients` list when they disconnect.

```
def remove_client(client: socket, username: str):
    """Remove a disconnected client from active_clients"""
    global active_clients
    active_clients = [(user, conn) for user, conn in active_clients if conn != client]

    try:
        client.close()
    except Exception as e:
        print(f"server ~ Error closing client socket: {e}")

    send_message_to_all(f"server ~ {username} has left the chat.")
```

Arguments:

- `client`: The socket representing the connection.
- `username`: The name used to identify the client.

Returns:

- Nothing. Side effect: updates global list `active_clients` and notifies others.

Purpose:

- Cleans up a disconnected client.
- Uses list comprehension to remove the client's tuple from `active_clients`.
- Closes the socket.
- Notifies all users about the user leaving.

Listening for Client Messages

The `listen_for_messages()` function is used to continuously listen for incoming messages from the client.

```python
def listen_for_messages(client: socket, username: str):
    """Function to listen for incoming messages from a client"""
    while True:
        try:
            message = client.recv(2048).decode("utf-8")
            if message:
                prompt_message = f"{username} ~ {message}"
                send_message_to_all(prompt_message)
            else:
                print(f"server ~ {username} has disconnected.")
                remove_client(client, username)
                break
        except Exception:
            print(f"server ~ Connection lost with {username}.")
            remove_client(client, username)
            break
```

Arguments:

- `client`: The socket for a specific client.
- `username`: Name of the connected user.

Returns:

- Nothing. Runs in a thread.

Purpose:

- Runs a loop that constantly reads messages from the client using `recv()`.
- On valid message, formats and broadcasts it.
- On failure or disconnect, removes the client.

Key Concepts:

- `recv(2048)`: Reads up to 2048 bytes.
- `.decode("utf-8")`: Converts byte string to human-readable string.

Sending Messages

The `send_message_to_client()` function encodes and sends a message to a specific client using sendall(), ensuring full transmission.

```python
def send_message_to_client(client: socket, message: str):
    """Function to send a message to a single client"""
    client.sendall(message.encode("utf-8"))
```

Arguments:

- `client`: The target client's socket.
- `message`: String to be sent.

Returns:

- None.

Purpose:

- Sends a UTF-8 encoded message to a specific client using `sendall()`.

Key Concept:

- `sendall()` ensures the whole message is sent (unlike `send()` which may send partial data).

The `send_message_to_all()` function broadcasts messages to all connected clients

```
def send_message_to_all(message_sent: str):
    """Function to send any new messages to clients that are connected"""

    for user in active_clients:
        send_message_to_client(user[1], message_sent)
```

Arguments:

- `message_sent`: The message to broadcast.

Returns:

- None.

Purpose:

- Loops through all `active_clients` and sends the message to each using `send_message_to_client()`.

Handling New Client Connections

The `client_handler()` function is used to handle connection with the clients.

```python
def client_handler(client: socket):
    """Function to handle client connections"""

    while True:
        # Server will listen for client message that will contain the username
        username = client.recv(2048).decode("utf-8")
        if username != "":
            active_clients.append((username, client))
            send_message_to_all(f"server ~ {username} has joined")
            break
        else:
            print("server ~ client username is empty")

    threading.Thread(
        target=listen_for_messages,
        args=(
            client,
            username,
        ),
    ).start()
```

Arguments:

- `client`: The socket for the connected client.

Returns:

- None.

Purpose:

- Waits for the client to send its username.
- Adds the client to the active list.
- Notifies all users that a new client has joined
- Starts a thread to listen for messages from this client.

Shutting Down the Server

The shutdown_server() function is used to gracefully shutdown the server.

```
def shutdown_server():
    """Handles server shutdown and disconnects all clients"""
    global active_clients, server

    print("server ~ Closing all client connections...")
    for username, client in active_clients:
        try:
            client.sendall("server ~ Server is shutting down.".encode("utf-8"))
            client.close()
        except Exception as e:
            print(f"server ~ Error closing client {username}: {e}")

    active_clients.clear()  # Remove all clients from the list

    try:
        server.close()
        print("server ~ Server socket closed.")
    except Exception as e:
        print(f"server ~ Error closing server socket: {e}")

    sys.exit(0)  # Exit the program cleanly
```

Arguments:

- None.

Returns:

- None.

Purpose:

- Gracefully disconnects all clients and shuts down the server.
- Sends a shutdown message to all clients.
- Closes all sockets.
- Calls `sys.exit()` to terminate the script.

Main Function

The main function is where the execution of the script starts.

```
def main():
    """Server main function"""
    global server  # Declare server as global so we can close it in signal handler

    server = socket(AF_INET, SOCK_STREAM)

    try:
        server.bind((SERVER_IP, SERVER_PORT))
        print(f"server ~ Running the server @{SERVER_IP}:{SERVER_PORT}")
    except Exception as e:
        print(f"server ~ Unable to bind server @{SERVER_IP}:{SERVER_PORT}")
        print(f"server ~ {e}")
        sys.exit(1)  # Exit if binding fails

    server.listen(LISTENER_LIMIT)

    print("server ~ Waiting for connections... (Press Ctrl+C to stop)")

    try:
        while True:
            client, address = server.accept()
            print(f"server ~ Successfully connected to client @{address[0]}:{address[1]}")
            threading.Thread(target=client_handler, args=(client,)).start()
    except KeyboardInterrupt:
        print("\nserver ~ Shutting down gracefully...")
        shutdown_server()
```

Arguments:

- None.

Returns:

- None.

Purpose:

- Creates a TCP socket using `socket(AF_INET, SOCK_STREAM)`.
- Binds it to the host and port.
- Calls `listen()` to accept incoming connections.
- Loops infinitely to `accept()` clients and starts a new thread for each using `client_handler`.

Key Concepts:

- `bind()`: Associates the socket with an IP and port.
- `listen()`: Puts the socket into server mode.
  `accept()`: Blocks until a client connects; returns new socket and address.
- Each connection is handled in a separate thread → enables multiple clients (concurrent chat).

Signal Handling

The following line catches **`SIGINT`** signals (which is raised when the user presses Ctrl+C) and runs `shutdown_server()` instead of exiting abruptly.

```
# Handle Ctrl+C shutdown
signal.signal(signal.SIGINT, lambda sig, frame: shutdown_server())
```

Running the Server

Finally the last few lines ensure the script runs the `main()` function when the script is executed.

```
if __name__ == "__main__":
    main()
```