Computer Networks

Assignment - Socket Programming

Ranjot Sandhu

Section A: Server Side

Server Code (server_fixed.py)

The server is responsible for handling multiple client connections, managing client requests, and providing file-sharing functionality. It runs on a specified IP and port while handling a maximum of three clients using multi-threading. This ensures that multiple clients can connect and communicate with the server simultaneously.

Key Configurations:

```
SERVER_HOST = "192.168.2.53"
SERVER_PORT = 12345
MAX_CLIENTS = 3
FILE_REPOSITORY = "server_files"
```

This section of the code defines the key parameters for the server. SERVER_HOST specifies the IP address where the server is hosted, while SERVER_PORT defines the port number on which the server listens for incoming client connections. The MAX_CLIENTS variable limits the number of concurrent connections to three to prevent overload. The FILE_REPOSITORY directory is where shared files are stored for client access.

Handling Client Connections:

```
def handle_client(client_socket, client_address, client_na
    global active_clients
    with clients_lock:
        clients_cache[client_name] = {
            'address': client_address,
            'connected_at': datetime.now(),
            'disconnected_at': None
        }
```

The handle_client function is responsible for managing each individual client connection. When a new client connects, the server assigns it a unique name and records its IP address and connection time. This information is stored in a shared dictionary (clients_cache) that maintains active client connections. A threading lock (clients_lock) ensures that multiple clients do not modify the shared data at the same time, preventing data corruption.

Processing Client Requests:

```
try:
    while True:
        message = client_socket.recv(1024).decode()
        if not message:
            break  # Client disconnected

        if message.lower() == "exit":
            print(f"{client_name} disconnected.")
            break
        elif message.lower() == "status":
            status = "\n".join([f"{k}: {v['address']} - Connected at {v['connected_at']}" for k, v in cl
            client_socket.send(status.encode())
        elif message.lower() == "list":
            files = os.listdir(FILE_REPOSITORY)
            file_list = "\n".join(files) if files else "No files available."
            client_socket.send(file_list.encode())
        elif message.startswith("get "):
            filename = message.split(" ", 1)[1]
            file_path = os.path.join(FILE_REPOSITORY, filename)
            if os.path.exists(file_path):
                with open(file_path, "rb") as file:
                    client_socket.sendall(file.read())
            else:
                client_socket.send("File not found".encode())
        else:
            response = f"{message} ACK"
            client_socket.send(response.encode())
except Exception as e:
    print(f"Error with {client_name}: {e}")
```

This section of code is responsible for handling client commands. The server continuously listens for incoming messages from the client and processes them accordingly. If the client sends exit, the server disconnects the client. The status command retrieves details of all currently connected clients. The list command fetches a list of available files in the server_files directory. If a client requests a specific file using get <filename>, the server checks if the file exists and, if found, sends its contents to the client. Otherwise, it responds with a "File not found" message.

Staring the Server:

```
def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.S
    server_socket.bind((SERVER_HOST, SERVER_PORT))
    server_socket.listen(MAX_CLIENTS)
    print(f"Server listening on {SERVER_HOST}:{SERVER_PORT
```

The start_server function initializes the server by creating a TCP socket and binding it to the specified IP and port. The listen() function is used to allow up to MAX_CLIENTS connections at the same time. The server then waits for clients to connect and interact with it. **Section B:**

Section B: Client Side

Client Code (client_fixed.py)
The client program connects to the server and allows users to send commands and receive responses. This enables users to check server status, list files, and request file downloads.

Key Configuration:

```
SERVER_HOST = "10.49.189.118"
SERVER_PORT = 12345
```

This block of code sets up the client's configuration. SERVER_HOST specifies the IP address of the server the client is trying to connect to, while SERVER_PORT is the port number on which the server is listening. These values must match the server settings for a successful connection.

Connecting to the Server:

```
def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.S
    try:
        client_socket.connect((SERVER_HOST, SERVER_PORT))
        print("Connected to server. Type messages, 'status
    except Exception as e:
        print(f"Could not connect: {e}")
        return
```

The start_client function creates a TCP socket and attempts to connect to the server. If successful, it displays instructions for interacting with the server. If the connection fails, an error message is displayed, and the client exits.

Sending and Receiving Messages:

```
while True:
    message = input("You: ")
    client_socket.send(message.encode())
    if message.lower() == "exit":
        break
    response = client_socket.recv(1024).decode()
    print(f"Server: {response}")
```

This part of the client code runs a loop where the user can enter messages to send to the server. The message is encoded and sent through the socket connection. The client then waits for a response from the server and prints it. If the user types exit, the loop stops, and the client disconnects from the server.

Section C: Difficulties Faced and How They Were Solved?

Several challenges were encountered while implementing the client-server system. The first issue was a mismatch in IP addresses between the client and server, which prevented connections. This was fixed by ensuring both programs used the same IP address. Another issue was an extra space in the server's SERVER_HOST variable, which caused connection failures. The space was removed, and the connection worked correctly. Additionally, the client initially did not handle binary file transfers properly, which was fixed by modifying how files were read and sent over the network.

Section D: Testing and Results

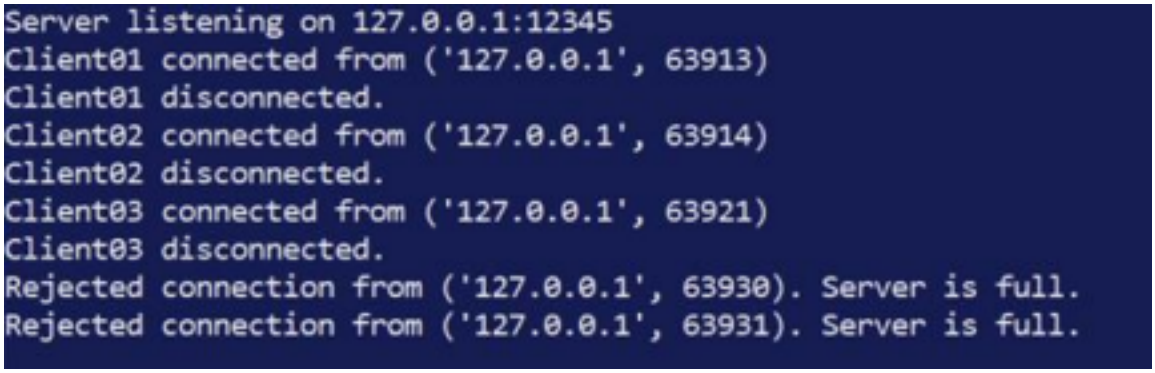
When testing here is the Server terminal whereas you can see the client has connected and the server was set up properly. Each client has been assigned a number properly and name we had one client disconnect as well. We had Client 2 disconnect by closing the terminal and client 1 disconnect the proper way by exiting. It has also show that the server can handle multiple clients(multi-threading).

```
PS C:\Users\Ranjot Sandhu> cd "C:\Users\Ranjot Sandhu\Desktop\server_project"
PS C:\Users\Ranjot Sandhu\Desktop\server_project> python server_fixed.py
Server listening on 10.0.0.101:12345
Client01 connected from ('10.0.0.248', 60658)
Client02 connected from ('10.0.0.211', 60744)
Error with Client02: [WinError 10054] An existing connection was forcibly closed by the remote host
Client01 disconnected.
```

Here is the Client terminal whereas you can see the connection was made properly. The server and client can exchange messages while maintaining the connection. The screenshot demonstrates the server-client interactions. When the client types a string, the server responds with the string followed by 'ACK.' Typing 'status' provides the client status, and typing 'exit' disconnects the client from the server. Server maintains client's connections details which are sent to the client when client sends the status message to the server.

```
ranjotsandhu@I-SEE-YOU Desktop % python3 client_fixed.py
Connected to server. Type messages, 'status' for server status, 'list' for files, 'get <filename>' to download, 'exit' to disconnect.
You: Hello Server!
Server: Hello Server! ACK
You: status
Server: Client01: ('10.0.0.248', 60658) - Connected at 2025-02-07 12:23:05.808313
Client02: ('10.0.0.211', 60744) - Connected at 2025-02-07 12:28:23.446195
You: exit
Disconnected from server.
ranjotsandhu@I-SEE-YOU Desktop %
```

To test if more than 3 clients can connect, we needed more computers so we tested that and as you can see, the server is not allowing more than 3 clients at a time. It will reject them and let you know that the server is full.



Section E: Improvements and Conclusion

The implemented client-server system successfully enables multiple clients to communicate with the server, retrieve status, list files, and download files. The system provides basic functionalities that are crucial for a networking-based application and ensures smooth interaction between clients and the server. The use of multi-threading allows multiple users to connect at the same time without affecting each other's experience.

Even though the system is functional, there are a few areas that could be improved for better efficiency and user experience. Enhancing error handling would make the system more robust, preventing unexpected crashes due to faulty input or connection errors. Implementing a more structured client name management system would allow better tracking of users, especially in cases of reconnections.

Additionally, the system could benefit from a well-designed graphical user interface (GUI) to make interactions more accessible for users who are not comfortable with command-line interfaces. Features such as file transfer progress indicators and improved logging mechanisms would make the client-server interaction more informative and user-friendly.

Overall, this project successfully demonstrates how socket programming can be used to create a basic client-server model. It lays a strong foundation for further enhancements, making it a valuable learning experience in network programming. With additional refinements and feature expansions, this system can be turned into a more advanced and practical application used in real-world scenarios.