Ronav Pillay (z5417181) – COMP3331 ASSIGNMENT

In this Assignment, I have chosen to use Python as my primary choice for implementation as it is a language I am quite well-equipped in and due to its extensive capability in socket programming I thought it would be the best approach. There are two files **client.py** and **server.py**. Client.py handles all client-side functionalities such as client's commands, different file operations and overall, for its communication with the central server. Server.py handles all requests from clients such as managing authentication, checking active peer tracking and the application layer protocol.

In my design, aside from the client-server and peer-to-peer implementation as per the specifications given for the Assignment for the multithreading capability of my design, I used a heartbeat thread and a file server thread. The heartbeat thread continuously sends heartbeats to the server to keep the user marked as active. The file server thread accepts the file transfer requests whilst the client performs other tasks. This approach allows that background tasks don't interfere with the main client's functionality as told in the specification.

Data Structure:

In my code, there are three different dictionaries that are used credentials, active users and published files dictionary. The credentials dictionary is used when the user logs in, the server will load the user's credentials into a dictionary with the username as the key and the password as the value. This is a neat way to just store all the credentials of the user when they login and is very efficient

For the active user's dictionary, it contains three pieces of data, last_seen, tcp_port and address. The last_seen timestamp allows the server to remove the inactive users by using this value and calculating their inactivity time from the previous heartbeat. Then I have stored the tcp_port of the client and their IP address because when the user logs it makes it easier for when or if they try to access other files, the transfers are easier as the server can find through the active users, check their tcp_port and then provide it to the client that wants the file therefore it makes it easier to initiate a TCP connection. I have published files as a dictionary as well because when a user publishes a file, it is important to find out who published the file as well because then it can also be used for file transfers between clients. So, in this dictionary I have made the filename the key to the file and then the value be the username. This also allows for efficiency when publishing, unpublishing files, searching and retrieving the file.

Application Layer Protocol Format:

For my assignment I have chosen to use a JSON message format primarily due me also doing the frontend course this term and wanting to improve my JSON understanding especially in UDP and TCP connections. It also provides a very simple and easy way to process messages and responses. The use of JSON key value structures makes it easy to read and extract key information especially for humans when debugging.

You can see in my code that each request type like login, pub, heartbeat etc all have a message response structure that provides a clear indication of what is being transferred between the client and the server.

```
response = {'status': 'success', 'message': 'Active users listed.', 'users': active_peers}
```
Using this, I can also take the status of the response, the message and other arguments required for that request, and it makes it much easier to deal with.

How each function Works:
Often in the client file depending on the command the respective function is called and then a JSON message I sent to the server with its respective command.

LPF:
The server then takes the request to check published_files dictionary and filters the published files, if there are files found then a success message with the filenames is returned with the response.

LAP:
The server then takes the requests to check the active_users dictionary and compiles it into a list which then returns it back to the client and depending on if there are active users or not a respective response message is returned.

PUB:
First the publish_file() function checks if it's in the clients CWD, then after sending the request to the server it adds the file to the published_files dictionary with the username as the owner. This returns a success request to the client from the server if it is successfully added.

UNP:
Operates the same way as publish with returning a respective response message if the file was unpublished from the file and checks if the user is the owner of the file.

SCH:
The server receives the request and checks published_files for filenames that contain the substring and aren't owned by the user requesting. If matching files are found, then it is return with its respective response message.

GET:
The server receives the get request and checks published_files for the requested filename and checks if the user that published it is active. Then the server gets the publishers IP address and TCP port using the active_users dictionary. Then the server creates a response message

```
if filename in published_files:
    owner = published_files[filename]
    if owner in active_users and 'tcp_port' in active_users[owner]:
        log_event(f"Received GET from {username}")
        update_last_seen(active_users, username)
        owner_data = active_users[owner]
        response = {
            'status': 'success',
            'peer_address': owner_data['address'],
            'peer_port': owner_data['tcp_port'],
        }
    else:
        response = {'status': 'fail', 'message': 'File owner is not active'}
else:
    response = {'status': 'fail', 'message': 'File not found'}
```

Which is then sent back to the requesting client. If there is a successful response the download file is then called with arguments peer_address, peer_port and filename. It is in download file where a TCP connection is initiated between the clients. Then a GET request message is sent to the publisher which is received by the publisher in their file server.

```
tcp_sock.sendall(f"GET {filename}".encode())
```

After this occurs the publisher's function, run_file_server() is constantly listening for any files that are incoming from other clients and then handle_file_request() accepts the TCP connection between the clients. This function now will read the request and extract the filename, check if it is the CWD and if it exists open it in binary read mode then send the contents of the file in chunks over a TCP connection. Once the file transfer is complete, the publisher closes the TCP connection. In the download file the client that receives the data each time the chunk arrives the client is writing the contents to a new file in the CWD with the same name then after that is finished the client closes the TCP connections and the file was downloaded successfully.

XIT:

When the client calls xit, it activates the exit_client() function which tells the global variable **keep_active** to false which then stops the user from sending heartbeats. This closes the connection with the server which then in the server file, load_inactive notices that the client has stopped sending heartbeats therefore remove him from active users. When xit it activated it makes sure that all threads are closed so no threads are still running even though the client has exited the server.

```
finally:
    keep_active = False
    # Close all threads before exiting
    server_sock.close()
    udp_sock.close()
    heartbeat_thread.join()
    file_server_thread.join()
    sys.exit(0)
```