A simplified client-server solution for file management

This report contains the overall workflow of the client-server solution for file management. Before starting with the code, we tried to take time for about one week by discussing how the overall workflow looks like and practicing to connect a simple client and server with the codes which are found on the course material and understanding them. Furthermore practicing how to read and write file for a simple structures and how to create a directory. After having the basics of those things we just implement a simple client and server connection so the client is waiting for a response from the server. Beside this we tried to do a simple coding for the service commands that have been listed on the server but not connecting to the server. Here below its described as how it looks

- creating a simple client and server connection and make sure that the server is connected for the client response without giving any commands
- creating a folder and make sure that the folder has been already created on the current working directory
- creating a write file at the first time and make sure that its able to write a text-file
- creating a read file and make sure that its able to read a text file
- Create a new folder with the specified name in the current working directory
- Move the current working directory to the specified folder.
- register a new user with a username and password for the first time

Now after those tasks which are done separately, we merged them with the simple client and server connection. We passed the blocks as a named tuple and see the response. This time server is able to receive the response from the client and passes the commands even if its not the final point that we need we are able to know that the connection is stable now. This functions are working for the basic functionality for the first time like the read file is able to read a file, the write file is able to write a file and not yet done as the requirements as mentioned.

AFTER BASICS

After having all the basics, we just started to clear out things. Tasks that are done at this time are described as follows.

1. read_file <name>:
   - The Read file is currently working as it reads a file without any requirements
   - Now Its working for reading data from the file <name> in the current working directory for the user issuing the request and return the first hundred characters in it.
2. write_file <name> <input>:
   - This is working currently as the data is written in <input> to the end of the file <name> in the current working directory for the user issuing the request. And furthermore, if no file exists with the given <name>, a new file is going to be created in the current working directory for the user.
3. list:
   - The lists are now generated but looks messy and not arranged on line.
4. register <username> <password> <privileges>:
   - The register function is currently working as it passes a username and password. Here the information related to the user i.e. username and password, are stored in the pickle file. The privilege is not yet included.
   - Now the privileges are included and the user is able to register with the username, password and privilege.
5. login <username> <password>:
   - Log in the user conforming with <username> onto the server and <password>
6. create_folder <name>:
   - Create a new folder with the specified <name> in the current working directory for the user issuing the request.

7. change_folder <name>:
• Move the current working directory for the current user to the specified folder residing in the current folder.

The filesystem, folders, paths, and fileindex

Navigating the file system works by each user object (whether created from User or Admin class) having an attribute current_path which keeps track of the user's location. Ever time the user sends a command to the server which is to be executed in a folder (i.e. most commands), only then does the server actually move to that folder, where it executes the command, and then it returns to root. The server is always based in the root directory, and moves relative to it.

As an example, let's say a user wants to list the contents of some sub-directory of their home folder. When the user logs in, the server is in root, where is can ope the pickle file to check password etc., if the username and passoword is correct, the server sets the user object's current_path attribute to their home folder. The Server does not actually execute os.chdir or any similar command, but only sets the user objects current_path attribute and writes to the client that the user has been logged in succesfully and moved to the home folder. Then if the user wants to move to a sub-directory they send the command ”change_folder name_of_folder”, and again the server does not need to actually move from root, but replies to the client that the user has moved folders. The only actual change on the server is that the user object's current_path attribute is updated to point to the subdirectory. Then if the user issues a ”list” command, it must be executed in the correct location. The first thing the server does inside the corresponding list function is to use os.chdir to go to the path specified by the user objects's current_path attribute, and then the server proceeds to read the contents of the folder in order to list them. After returning from the list function the server can return to root.

The purpose of this system is to make sure requests from different clients do not interfere with eachother. Instead of executing folder changes (caused by commands change_folder and login) directly on the server, we execute these changes on an attribute current_path instead. And commands which must be executed inside a folder are first moved there and immediately afterwards moved back, so even with multiple clients sending commands, the server always starts and finishes at root, but for each user the experience is essentially the same as if os.chdir was executed directly and they were the only person accessing the server.

This is similar to how we handle reading and writing files. If the server just opened a file to read from and it was kept open, commands from different users could interfere. Therefore we make sure to close the file every time, and keep track of the user's position with a separate variable. If the user keeps reading from that file, the server opens the file again, which means it starts from the beginning. Then the server uses the self.index to know how far it needs to read. This simulates the effect of having the file remain open, but we can still close the it between commands.

Problem Solved for User interfere with Another:

One of the requirements was that one client's service requests should not interfere with another's. Our earlier versions of functions all interfered to a large degree, and so we created a system to prevent this.

Navigating the file system works by each user object (whether created from User or Admin class) having an attribute current_path which keeps track of the user's location. Ever time the user sends a command to the server which is to be executed in a folder (i.e. most commands), only then does the

server actually move to that folder, where it executes the command, and then it returns to root. The server is always based in the root directory, and moves relative to it.

As an example, let's say a user wants to list the contents of some sub-directory of their home folder. When the user logs in, the server is in root, where is can ope the pickle file to check password etc., if the username and passoword is correct, the server sets the user object's current_path attribute to their home folder. The Server does not actually execute os.chdir or any similar command, but only sets the user objects current_path attribute and writes to the client that the user has been logged in succesfully and moved to the home folder. Then if the user wants to move to a sub-directory they send the command ”change_folder name_of_folder”, and again the server does not need to actually move from root, but replies to the client that the user has moved folders. The only actual change on the server is that the user object's current_path attribute is updated to point to the subdirectory. Then if the user issues a ”list” command, it must be executed in the correct location. The first thing the server does inside the corresponding list function is to use os.chdir to go to the path specified by the user objects's current_path attribute, and then the server proceeds to read the contents of the folder in order to list them. After returning from the list function the server can return to root.

The purpose of this system is to make sure requests from different clients do not interfere with eachother. Instead of executing folder changes (caused by commands change_folder and login) directly on the server, we execute these changes on an attribute current_path instead. And commands which must be executed inside a folder are first moved there and immediately afterwards moved back, so even with multiple clients sending commands, the server always starts and finishes at root, but for each user the experience is essentially the same as if os.chdir was executed directly and they were the only person accessing the server.

This is similar to how we handle reading and writing files. If the server just opened a file to read from and it was kept open, commands from different users could interfere. Therefore we make sure to close the file every time, and keep track of the user's position with a separate variable. If the user keeps reading from that file, the server opens the file again, which means it starts from the beginning. Then the server uses the self.index to know how far it needs to read. This simulates the effect of having the file remain open, but we can still close the it between commands.

LIST OF OTHER PROBLEMS:

Problems and Solutions

Having these functionality now, have Listed the difficulty that we face to proceed further, which are listed below.

Problem:

The service command, write_file <name> <input is able to write a file at the first time. But when it comes to the next time to write a file, it will continue on the same line.

Further expected result: when it comes on the next time, its expected to write on a new line.

Solution:

The write File is now working as expected. It writes a data in <input> to the end of the file <name> in the current working directory for the user issuing the request, starting on a new line.

Problem

The service command read_file <name> is able to read a file and returns return the first hundred characters but not works for subsequent calls.

Further expected result: Each subsequent call by the client is to return the next hundred characters in the file, up until all characters are read.

Solution: The service command, read_file <name>, is working as expected as it reads a data from the file <name> in the current working directory for the user issuing the request and return the first hundred characters in it. Each subsequent call by the same client is to return the next hundred characters in the file, up until all characters are read. I

Problem:

The service command, list is working basically even if it looks messy.

Further expected result: Making the lists in order and readable.

Solution: The service command, list, is working as expected now. It prints all files and folders in the current working directory for the user issuing the request. This command is giving information about the name, size, date and time of creation, in an easy-to-read manner.

Problem:

The service command register <username> <password> <privileges> is able to register users with a specific password and privilege but not working as expected. It just creates the user folder in any random place and paths are not yet defined.

Further expected results must be defined

The service command register <username> <password> <privileges> is partially working as expected now.

Solution: File structure has been done first as we create a root directory and under that we are going to have a user and admins. Since we might have a number of users and also administrator's, we just specify them as groups of users and Admins so that if the user has a privilege of admin, a folder is going to be created as a root>Admins>username and the same for the ones that have an access level of user. Furthermore classes are created so that a user can get their own information. Now when a user is registered, can get all the information like its password and its privilege.

Problem: The service commands login <username> <password> is able to login a user and passes as user is logged in successfully but it still Logs in with another user password.

More Expected result: when the command login passes it has to check the correct password of the username.

Solution: The service command login<username><password> is working now. By conforming with <username> onto the server if the <password> provided matches the password used while registering, it allows login. If the <password> does not match or if the <username> does not exist, an error message is going to be returned to the request for the client to present to the user.

Now since we are able to know the corresponding password with the specific username when creating the file, the user can successfully login with only the correct password.

Problem : The service command delete <username> <password> has not yet been started. The register and login commands have to work properly to stat with

Expected Result: The service command, delete <username> <password>, has been tried for the first time now as the register and login function works well. But still there is a gap on the paths and not working well. Sometimes its unable to find the correct path since it creates the root on different directory and cant get the pickle file.Now everything seems to work other than the Delete command. Not getting the correct path is not only disturbing the Delete user but also multiple users login. When multiple users login and connected to a server, they start to interfere each other. Like the commands one user passes starts to interfere with the other user. It its changing directory, the directory is going to be changed for all users but not to the specific user.

Solving problems

In order to solve the problem, we just forced to go to the root directory at first whenever a user tries to login by a change directory. Then it will going to catch the path depending on the access level given like if its admin its going to enter to the group Admins by the change directory and matches the corresponding username and the same for users too. This is more modified as follows:

Now we are able to find the correct path and problem solved.

Delete the user conforming with <username> from the server is working now. It will delete the user by conforming the admins password which is currently login.

On the Final stage, trying to check the flow of all commands as it works properly and figure out the things that has left from the requirements and Fixing the ones that are not working properly.

User and Admin class have mostly the same attributes, except for the delete-function which only Admin has access to. Because of this we have all the code in User class, the Admin class inherits everything and also adds the delete-function.

Error Handling:

We have used different error handling scenario depends upon the functionality in this execution. Here are the list of error handling we have used so far:

1. OS Error : If the readfile is not exists in the specified path, it means if we give wrong file it throws this error message.
2. Index Error: If client gives commands that is not specified means it prompts the user with this error and saying corrrect format of giving commands.
3. Attribute Error: When we delete the username, if other than whose privileges is admin try to delete the username, it throws Attribute Error
4. Key Error: If the user try to give commands without logged in, it throws Key error.
5. File not found Error: When changing directory if the file does not exists in that location it throws File not found error message
6. File Exists Error: When we create new folder if that folder name already created, it throws File Exists Error