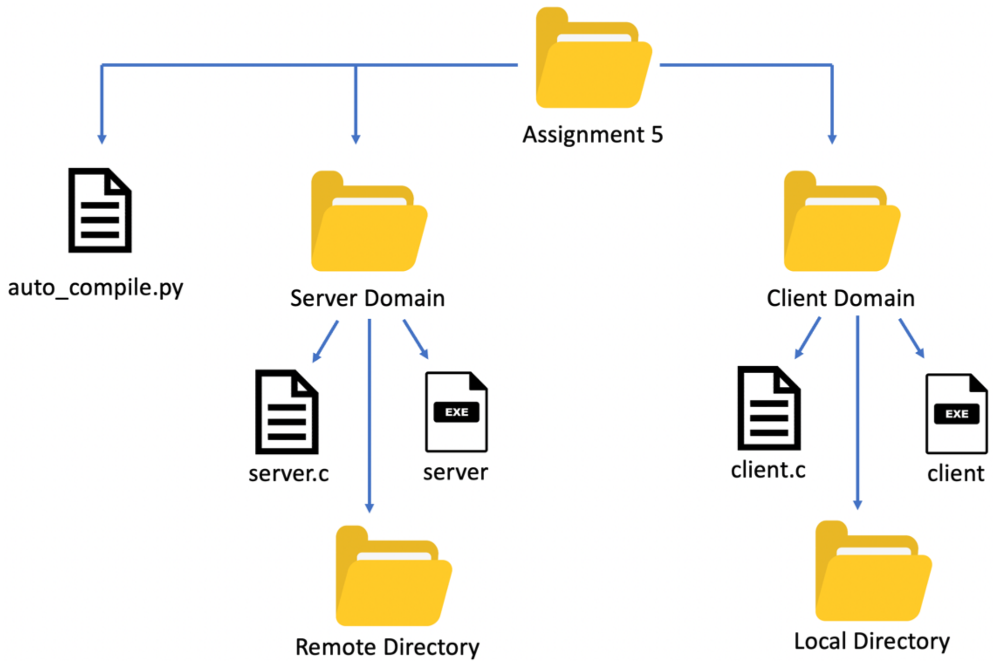
Online Cloud Storage Application

In this assignment, you will create an application in “c” programming language that lets users download and upload files from and to an online cloud storage.

<u>**Overview**</u>:

The assignment should be composed of two c files, “client.c” and “server.c”. Each of these two files will be compiled separately, the compiled executables of each file will be named “client” and “server” respectively. The client source code and executable should inside a folder named “Client Domain” while the server source code and executable should inside a folder named “Server Domain”. All of the client’s local files should be inside a folder named “Local Directory” which should be located inside the “Client Domain” folder. Similarly, all files hosted on the cloud storage should be inside a folder named “Remote Directory” which should be located inside the “Server Domain” folder. Finally, everything (files and folders) should be inside a folder named “Assignment 5”. Please refer to the picture below to visualize the complete design of the files and folders structure of your assignment.

Note: inside the “Assignment 5” folder, there will be a python file named “auto_compile.py” which will be used to compile your “client.c” and “server.c”. Feel free to edit this file and include all the compile switches that you need.



Running the Application:

Running the server: first you need to run the server which will open a server TCP socket that listen for incoming client connections. The server port number should be **9999**. The server IP address should be provided as an argument when you run the server such as:

```
./server 127.0.0.1
```

Running the client: to ease testability and gradeability of your application, your client application will not take commands from the command-line input (the terminal). Instead, your application should read all user commands from a text file. Your client application will expect this text file to be located in the “Client Domain” folder and read the file name as an argument. Furthermore, your client application should receive the server IP address as a second argument. For example, if the user commands text file is named “user_commands.txt” and the server IP address is 127.0.0.1, then you should run the client executable as:

```
./client user_commands.txt 127.0.0.1
```

To support multi-threading: user of your application can start multiple clients to read or write to a file. A server starts a new thread to handle each successful client connection. You can test the multi-clients behavior by starting clients on different terminals on the same machine.

[Terminal 1 on <usr>@openlab.ics.uci.edu]
```
./client user1_commands.txt 127.0.0.1
```

[Terminal 2 <usr>@openlab.ics.uci.edu]
```
./client user2_commands.txt 127.0.0.1
```

Note 1: when you submit your assignment, don’t include any user commands files. The graders will have their own test cases with their own command files.

Note 2: We highly advise you to start coding without multi-threading support first. Then update your code and implement the multi-threading aspect of it.

User Commands:

Your application should support seven different commands which are described below. These commands should be implemented in the “client” code.

Running the client: when a user starts the application, the following message appears on the terminal “Welcome to ICS53 Online Cloud Storage.” and the prompt sign “> ” appears on a new line (note that there is a single space after the “> “ sign).

```
./client user_commands.txt 127.0.0.1
Welcome to ICS53 Online Cloud Storage.
>
```

1- pause <time>: as discussed before, your application should be able to handle multiple users at the same time. Since the user behavior is read from a file instead of taking user input in real-time, an entire scenario of a single user (a single text file of commands) would be executed in a fraction of a second. This behavior would make it difficult to test the concurrency aspects of multithreading in the code. Therefore, your code should support a very simple "pause <time>" command that would pause the execution of the client code for <time> seconds.

Example:

```
> pause 1
>
```

2- append: the user can type "append <file_name>" to start string appending mode on the file "file_name" located in the remote directory. If <file_name> is not found in the remote directory, the application prints "File <file_name> could not be found in remote directory.". Otherwise, the application enters appending mode and the prompt is changed from "> " to "Appending> "

In appending mode, the string entered by the user will be appended to the end of "file_name." **Unless** the string starts with "close" or "pause". The "close" command will close appending mode and return to the "> " prompt. The "pause" command takes one argument, the time in seconds, and pauses the program that amount of time.

For illustration only, **red font** is printed by the application, while **green font** is typed by the user.

Example: assume "hello.txt" exists on remote server with the following one-line content of text: "hello world".

```
> append not_hello.txt
File [not_hello.txt] could not be found in remote directory.
> append hello.txt
Appending> more hello world
Appending> pause 10
Appending> close
>
```

On the remote server, the hello.txt file will have the following content:

```
hello world
more hello world
```

Note 1: "close" command only works in appending mode, and it is not recognized outside of appending mode.

Note 2: intuitively, "append" should only work on file with ASCII-readable content such as ".txt". However, your code should not make this check and leave it up to the user to make such mistake.

3- upload: to upload a file from the local directory to the remote directory, the user types "upload <file_name>". If the file doesn't exist in the user's local directory, the application prints "File <file_name> could not be found in local directory.". Otherwise, the application uploads the file from the user's local folder to the remote folder via a TCP socket. Then the application prints a success message that indicates the file size as shown below.

Example: assume "my_file.pdf" is the only file exists in the user's local directory:

```
> upload my_file.pfd
File [my_file.pfd] could not be found in local directory.
> upload my_file.pdf
12000 bytes uploaded successfully.
>
```

4- download: to download a file from the remote directory to the local directory, the user types "download <file_name>". If the file doesn't exist in the user's remote directory on the server, the application prints "File <file_name> could not be found in remote directory.". Otherwise, the application downloads the file via a TCP socket from the user's remote directory to the local directory. Then the application prints a success message that indicates the file size as shown below.

Example: assume "my_picture.jpg" is the only file exists in the remote directory:

```
> download my_picture.jpeg
File [my_picture.jpeg] could not be found in remote directory.
> download my_picture.jpg
23000 bytes downloaded successfully.
>
```

5- delete: the user can type "delete <file_name>" command to delete <file_name> on the remote directory. If the file doesn't exist in the remote directory on the server, the application prints "File <file_name> could not be found in remote directory.". Otherwise, the server deletes the file from the remote directory and the client terminal prints a success message.

Example: assume "my_picture.jpg" is the only file exists in the remote directory:

```
> delete my_picture.jpeg
File [my_picture.jpeg] could not be found in remote directory.
> delete my_picture.jpg
File deleted successfully.
>
```

Note: we assume that the user has a direct access to the files in the local directory. Therefore, "delete <file_name>" command is used only for deleting files on the remote directory.

6- syncheck: the user can type “syncheck <file_name>” command to get a short report about “file_name”, whether on the local or remote directory.

Example: assume the following four cases:
- file “my_file.pdf” only exist in local directory.
- file “my_picture.jpg” only exist in remote directory and in appending mode by some user.
- file “my_report.csv” exist in both local directory and remote directory.
- file “my_notes.txt” exist in both local directory and remote directory, but the user has modified the file in the local directory after the user has uploaded the file.

The client terminal should print file size in both directories. If the file exists in remote directory, the terminal should also print sync status and lock status. The report format is shown below.

If there is a file that exist in both the local and the remote directories, and both files have same file name, then these two files are considered the same file. If these two files (that have the same file name) have different content, then the file is considered “unsynced” (ex: “my_notes.txt”).

Example:
```
> syncheck my_file.pdf
Sync Check Report:
- Local Directory:
-- File Size: 6700 bytes.
> syncheck my_picture.jpg
Sync Check Report:
- Remote Directory:
-- File Size: 23000 bytes.
-- Sync Status: unsynced.
-- Lock Status: locked.
> syncheck my_report.csv
Sync Check Report:
- Local Directory:
-- File Size: 1200 bytes.
- Remote Directory:
-- File Size: 1200 bytes.
-- Sync Status: synced.
-- Lock Status: unlocked.
> syncheck my_notes.txt
Sync Check Report:
- Local Directory:
-- File Size: 13 bytes.
- Remote Directory:
-- File Size: 13 bytes.
-- Sync Status: unsynced.
-- Lock Status: unlocked.
>
```

Note: two files of the same name might have the same size but with different content. Therefore, the size of the file is not a reliable indicator of the file sync status. To check whether if two files with the same name are the same file, you can use the MD5 has function.

MD5 Hash Function:

MD5 is a well-known hash function that gives you a 16-bytes hash value for a file. If two files have the same hash, then these two files are exact copies of each other (have the same content). If there is even a single bit difference between the two files (even if they in the gigabytes of size), the hash values of these two files will be completely different. Note that MD5 will only consider the file content in computing the hash, not the name of the file.

You can download the implementation of the MD5 algorithm in c from Ronald Rivest webpage (the scientist who designed the MD5 algorithm) via the following link:
https://people.csail.mit.edu/rivest/Md5.c

The code is straight forward and has the "MDString" function to compute a hash for a string and the "MDFile" function to compute a hash of a file. You simply need to modify the "MDFile" function and make it return the hash instead of printing it in the terminal.

7- quit: the user can type "quit" command to close the client socket then terminate the client application (but not the server). To quit the server, you should do "ctrl+c" on the server terminal which would cause the server to close all active client sockets as well as the server socket, then terminate the server application.

Example (on the client terminal):
```
> quit
$ # back to Linux prompt
```

Multithreading Synchronization:

To ensure the synchronization of files between the users, if a user opens a file in appending mode, no other user could issue "append" command on the same file, or any of the three commands that will be discussed next (download, upload, and delete). Therefore, "append" command should lock the file first before opening the file in appending mode. Any other users who try to download, upload, delete, or append the same file, their application should print the following error:

```
> append hello.txt
File [hello.txt] is currently locked by another user.
>
```

The lock of the file is released when the appending user exists appending mode by issuing the "close" command.

Testing the Application:

Let's assume that you have the file "hello.txt" from previous example, and the file is located in the "Local Directory" folder while the "Remote Directory" folder is empty.

Assume "user1_commands.txt" file located in the "Client Domain" folder and has the following commands:

```
download hello.txt
upload helo.txt
upload hello.txt
append hello.txt
more hello world
pause 1
close
quit
```

Assume "user2_commands.txt" file located in the "Client Domain" folder and has the following commands:

```
download hello.txt
syncheck
pause 2
download hello.txt
close
quit
```

Given two client terminals, Terminal 1 runs first:

[Terminal 1 on <usr>@openlab.ics.uci.edu]

```
./client user1_commands.txt 127.0.0.1
Welcome to ICS53 Online Cloud Storage.
> download hello.txt
File [hello.txt] could not be found in remote directory.
> upload helo.txt
File [helo.txt] could not be found in local directory.
> upload hello.txt
13 bytes uploaded successfully.
> append hello.txt
Appending> more hello world
Appending> pause 2
Appending> close
> quit
$
```

Then after 1 second, Terminal 2 runs:

[Terminal 2 on <usr>@openlab.ics.uci.edu]

```
./client user2_commands.txt 127.0.0.1
Welcome to ICS53 Online Cloud Storage.
> download hello.txt
File [hello.txt] is currently locked by another user.
> syncheck hello.txt
Sync Check Report:
- Local Directory:
-- File Size: 13 bytes.
- Remote Directory:
-- File Size: 30 bytes.
-- Sync Status: unsynced.
-- Lock Status: locked.
> pause 2
> download hello.txt
30 bytes download successfully.
> close
Command [close] is not recognized.
> quit
$
```

Note that your application will be auto-graded and therefore each command must print the exact format of messages for each command as described above (including the error messages) on the client terminal. Also, the auto-grader will check if the file content on the Remote Directory matches with the content of the same file that is located in the Local Directory. We will use the MD5 hash function to match the content of two files with the same name.

You will not be graded on what the server terminal prints so you can make the server terminal prints whatever you want.

Summary:

Your application must support the following commands:

```
1- pause <time>
2- append <file_name>
3- upload <file_name>
4- download <file_name>
5- delete <file_name>
6- syncheck <file_name>
7- quit
```

Also your application should handle the following errors:
- "`weird_command`" is any unrecognized command.
  - Error Message: "`Command [weird_command] is not recognized.`"
- "`upload some_file`" that doesn't exist in the local directory.
  - Error Message: "`File [some_file] could not be found in local directory.`"
- "`{download – append – delete} some_file`" that doesn't exist in the remote directory.
  - Error Message: "`File [some_file] could not be found in remote directory.`"
- "`{upload - download – append – delete} some_file`" that is locked (i.e. in appending mode by a user) in the remote directory.
  - Error Message: "`File [some_file] is currently locked by another user.`"

Submission:

To submit your assignment, zip your "Assignment 5" folder into "Assignment 5.zip" then submit your zipped folder to gradescope. Make sure to follow the naming of files and folders as described in the figure above. If you do not name your files and folders properly, your will lose points because your code won't be compiled and executed successfully. Also make sure to 1) **remove any files from the local and remote directories**, and 2) **remove any user text file commands** (such as the "user_commands.txt" used as an example above). The graders will provide their own sample files and text file commands to test your code. Finally, make sure to **follow the exact format of messages** that are printed on the client's terminal as described throughout the assignment.