CIS 41B - Lab 5

Write a simulation of a low-level client-server application where the server responds to a number of client requests.

Overview

The application is in 2 files: server.py and client.py. Both use sockets for communication and both run from the command line. (When we get to low level networking, there's no GUI and no IDE, it's all on the command line.)

The server code represents a server that can support multiple clients.

* The code has a command line argument, which is the number of clients that the server can respond to.
* The code creates a thread to respond to each client request.
* The client can request to:
  + list files and subdirectories in a *server side* directory
  + create a file in the *server side* directory
  + change directory (or go to a new directory) on the *server side*

The client code represents one client of the server.

* Upon connecting to the server, the client requests and receives the current directory where the server resides.
* The client can then make the requests above from the server.

Recommended steps to work on the lab.

**Step 1**. Write a server for one client and then write the client.

* If you normally use an IDE, then use the IDE's text editor to write the server.py and client.py files, but don't run the code in the IDE.
* Write the server to service one client, so don't put in threading code yet.  
  The code accepts 5 types of requests, runs a function to process each request, and sends back 1 result.  
  The following table shows the tasks that the server needs to do and the results:

|  |  |  |
| --- | --- | --- |
| Request | Task | Result |
| Get current directory | Find the path of the current directory | The path |
| Change to a new directory | Move from current directory to the new directory | - Success: client is at new directory  - Fail: new directory is not valid |
| List current directory | Find files and subdirectories in the current directory level (not recursive) | - List of subdirectory and file names |
| Create a new file | Create an empty, new file | - Success: file is created  - Fail: file already exists |
| Quit |  |  |

* Write the client to make each of the 5 requests:
  + Upon start up, automatically send the request to get the server’s current directory.
  + Then write a function to:
  + prompt, read in, and validate the user's choice to: change directory, list directory, create new file, quit. Keep re-prompting until you get a valid choice.
  + Each request is a single message that's sent to the server.
* For each request, write a function to display the single result from the server (see sample output)
  + For getting current directory: print the current directory name.
  + For changing to a new directory: print the new current directory name or print error message.
  + For listing a directory: print the current server directory name first, then print all subdirectory and file names or print "empty directory" message.
  + For creating a new file: print acknowledgment with the directory name, or print an error message.
* Open up a command (terminal) window to run the server, then open another command (terminal) window to run the client. This is because you need the client and server to be 2 different processes. For this lab they are both on the same system, but they represent 2 processes that can be run across the network.

[Side note: Welcome to the world of command line debugging, which many experienced programmers choose to use because, in their mind, only wimps need an IDE. In case you're new to the command line, here are some tips:

* Run the code until it fails, look at the error message and line number, go to the IDE (which is only a text editor at this point) and fix your code at or near the error line, then *don't forget to save* before re-running the code.
* A well-placed print statement in the code will tell you lots of info.
* If you get stuck in a loop somewhere, try control-c. And if that doesn't work because your client is waiting for the server and the server has hung (or vice versa), then close the command window and start again.
* The up arrow key to re-run the previous command (without having to re-type) will be your friend. ]

Make sure step 1 completely works before going to step 2. You don't want to add multithreading on top of partially working client-server code.

Also, we're writing a baby server here, so only use test cases where the data being sent across is less than 4K bytes. If you don't know how much data is more than 4K bytes, the exception will tell you.

**Step 2**.

In the server code, add 1 thread and have the child thread respond to the single client you currently have.

* The task for the thread starts from the response socket. This means you have the same socket (or port) for all clients, and just the response part is different for each client.
* Since we're writing a simulation of a client server application (not a real server that needs to be up 24/7), the server code will terminate when there's no client or no more client to serve. To prepare for multiple clients, add time out code for the listening socket so that it doesn't wait forever if there's no client. In real life, the server would wait forever for the clients.
* Make sure the server with thread works just as well as when you were in step 1, before starting step 3.  
  Test the timeout to see that it works. It's useful to have the server print "timed out" so you know that the time out happened.

**Step 3**.

Add code to the server so it accepts 2 command line arguments: the max number of clients and the timeout time of the timer. Since the command line is from the user, do user input validation. Check that:

* there are 2 command line arguments
* the max number of clients <= 4
* the timer time is between 3 and 30 seconds

Suggestion for testing:

* Store the server.py in a directory with 1-2 subdirectories and a few files. Run the server code at one command / terminal window.
* Store a copy of client.py at 2 different directories that are not the server directory. Open 2 other command / terminal windows for the 2 clients. Each client is in a different window at a different directory. This simulates 2 different computer systems for the 2 clients.  
  Run the 2 client.py file, and let the 3rd connection time out.
* Run different requests to the server at the 2 windows to see that the 2 clients can do different things at the "same" time.

Sample output

The following steps were run in the order shown.

Server window:

* runs in the “test” directory, which is the simulated server directory
* command line with input arguments 3 (num of clients) and 10 (timer for timeout)
* at start up, identify itself and show its port number

base) PS C:\Users\Clare\desktop\test> python lab5\_server.py 3 10

Server is up, hostname: localhost port: 5640

Connecting to client at port: 5640 # when a client connects, identify port

sending C:\Users\Clare\desktop\test # and print current directory that’s sent to client (at client request)

Connecting to client at port: 5640 # same with client 2

sending C:\Users\ Clare \desktop\test

timed out # no client 3, times out after 10s

Client 1 window:

* Runs at “desktop” directory, which simulates a directory on a different computer system
* At start up, identify the IP address and port that it uses to connect to server

(base) PS C:\Users\Clare\desktop> python lab5\_client.py 5640

Client connect to: 127.0.0.1 port: 5640

c. change directory # client prints menu for user

l: list directory

f. create file

q. quit

Enter choice: l # list current directory, which is server’s home directory

Directories and files found under C:\Users\Clare\desktop\test # start with path of current directory

dirA

dirB

fileC.txt

lab5\_server.py

\_\_pycache\_\_

c. change directory

l: list directory

f. create file

q. quit

Enter choice: c # change directory

Enter path, starting from current directory: dirB # prompt user for new path

New path: C:\Users\Clare\desktop\test\dirB # acknowledge with new path

c. change directory

l: list directory

f. create file

q. quit

Enter choice: l # list new directory, which is now current directory

Empty directory

c. change directory

l: list directory

f. create file

q. quit

Enter choice: f # create new file

Enter filename: fileB.txt

File created in C:\Users\Clare\desktop\test\dirB # acknowledge with location of new file

c. change directory

l: list directory

f. create file

q. quit

Enter choice: l # list current directory to confirm new file

Directories and files found under C:\Users\Clare\desktop\test\dirB

fileB.txt

c. change directory

l: list directory

f. create file

q. quit

Enter choice: c # change directory

Enter path, starting from current directory: ../dirC # invalid path

Invalid path

c. change directory

l: list directory

f. create file

q. quit

Enter choice: c # change directory

Enter path, starting from current directory: ../dirA # note path can go up to parent

New path: C:\Users\Clare\desktop\test\dirA

c. change directory

l: list directory

f. create file

q. quit

Enter choice: l

Directories and files found under C:\Users\Clare\desktop\test\dirA

fileA.txt

c. change directory

l: list directory

f. create file

q. quit

Enter choice: # client 1 idles while client 2 runs

Client 2

* Runs at “lab5” directory, which simulates a directory on a different computer system.
* At start up, identify the IP address and port that it uses to connect to server.

(base) PS C:\Users\Clare\desktop\lab5> python lab5\_client.py 5640

Client connect to: 127.0.0.1 port: 5640

c. change directory

l: list directory

f. create file

q. quit

Enter choice: l # list current directory

Directories and files found under C:\Users\Clare\desktop\test # server’s home directory

dirA # and not dirA, which is where client 1 is

dirB

fileC.txt

lab5\_server.py

\_\_pycache\_\_

c. change directory

l: list directory

f. create file

q. quit

Enter choice: c

Enter path, starting from current directory: dirB # change directory

New path: C:\Users\Clare\desktop\test\dirB

c. change directory

l: list directory

f. create file

q. quit

Enter choice: l # list directory

Directories and files found under C:\Users\Clare\desktop\test\dirB

fileB.txt

c. change directory

l: list directory

f. create file

q. quit

Enter choice: f # create new file

Enter filename: fileB.txt

File already exists # file exists, so no new file created

c.change directory

l: list directory

f. create file

q. quit

Enter choice: # client 2 idles

Back to client 1:

c. change directory

l: list directory

f. create file

q. quit

Enter choice: l # list directory

Directories and files found under C:\Users\Clare\desktop\test\dirA # client 1 is still at dirA

fileA.txt

When both clients have quit, then the server also ends.

Make sure to test for valid command line arguments also. This was not shown in the sample output above.

Turn in client.py and server.py