CIS 41B - Lab 5: sockets, review of system calls

Write a client-server application where the server can respond to a number of clients and send information about files and directories on the server.

Overview

The application is in 2 files: server.py and client.py. Both files will run from the command line in a command or terminal window.

The server:

* accepts 2 command line arguments: the number of clients it can serve, the number of seconds before time out
* creates a thread to respond to each client
* lets the client look up files and directories of a (supposedly remote) directory tree that's on the server side

Upon connecting to the server, the client requests the current directory where the server resides. Then the client can request the following from the server:

* f: show files and directories in the current directory
* d: show all subdirectories (recursive walk) of the current directory tree
* c: change the current directory to a new directory
* q: quit

Recommended steps to work on the lab

1. Write a server for 1 client and write the client.

* Write the server that can work with 1 client, so don't put in threading code yet
  + Accept 5 types of request messages: get current directory, change current directory, list all files in the current directory, list all subdirectories recursively in the current directory, quit
  + Write 4 functions to do the first 4 tasks and send back the result. Each request should only send back *1 result*.
    - For get current directory: send back the path of the server's current directory, which is also the client's current directory when the client first logs in.
    - For change current directory: accept a new directory and change the client's current directory to the new directory. Then send back 'success' or 'new directory is not valid'.
    - For list files: send back the list of files and directories in the client's current directory.
    - For list directories: send back the list of directories after a recursive walk from the client's current directory.
* Write the client to interact with the user and make each of the 4 requests
  + Upon start up, *automatically* send the request to get the server's current directory and print it out
  + Write a function to print a menu, then prompt, read in, and validate the user's choice to: change current directory, list files, list directories, quit. Keep re-prompting until you get a valid choice.
  + When there is a valid choice that's not the quit choice, send a request (a single message) to the server.
    - The request message has: 1) one of the 3 tasks that the server is supposed to do  
       2) for the change directory task only, the name of the new directory   
      The request should not contain the client's current directory because it's the server's job to keep track of where the client is within the server's directory tree.
    - After receiving the server response, print the appropriate result for the user (see sample output)
* To run / test your code, open 2 command windows or terminal windows on your system. Each window is a separate process: one window is to run the server and the other window is to run the client. Ideally the client and server are on different computers, but for this lab your testing will be with 2 processes on the same computer.

At the command line of one window, type the command line to start the server first. Then type the command to start the client at the other window.

[ Welcome to the world of command line debugging, which many experienced programmers choose to use because, in their mind, real(!) programmers don't use an IDE. In case you're new to this kind of debugging, here are some tips:   
-- Run the client py file until it fails, look at the error message, go to the IDE (which is only a text editor at this point) and fix your code, then *don't forget to save* before re-running the code.

-- If you get stuck in a loop somewhere, try control-c to end the process.

-- And if that doesn't work because your client (or server) is waiting for the other side and won't listen to you, then close the command window and start again.

-- The up arrow key to re-run the previous command 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 that's the result will tell you.

2. In the server code, add 1 thread and have the child thread respond to the client request.

* The thread starts from the response object. This means you have the same socket for all clients, and just the response part is different for each client.
* To prepare for multiple clients, add time out code for the listening socket. Since the listening socket's accept() method is a blocking method, add a time out so that it doesn't wait forever if there's no client.  
  You're writing a server that will shut itself off (will end) when there are no more clients to serve. In real life, the server will not have this time out so that it can stay up 'forever' to wait for client requests.
* 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. Have the server print a time out message so you know that the time out happened. (See sample output.)

3. Add code to the server so it has N number of threads (N = 3 is a good number to test with) to work with N possible clients. Suggestion for testing:

* Run the server code in a command/terminal window as before.
* Open up 2 other command/terminal windows for 2 clients. Each client is in a different window or different process.
* Let the 3rd connection time out since there will not be a 3rd client trying to connect.
* Run different server requests at the 2 windows, switching back and forth between the client windows, to see that the 2 clients can request different tasks at the "same" time and get the appropriate responses from the server.
* It might be good to change the server time out to more than 3 sec, unless you're really fast at starting the 2 clients.

But don't make the timer too long (3600 for example) because you also want to see what happens when there's no client connection.

4. Add code to the server so it accepts 2 command line arguments: the max number of clients and the time of the timer.

* Since the command line is from the user, do user input validation:
  + there are exactly 2 command line arguments
  + the max number of clients < 5
  + 3 < timer time < 120 sec
* If any of the 3 conditions above is not valid, print an appropriate error message for the invalid case and end the program.

There's a lot of freedom on how to code the 4 steps outlined above because you're about to 'graduate' as a fairly experienced Python programmer and you can handle it.

See sample output on next page.

Server window:

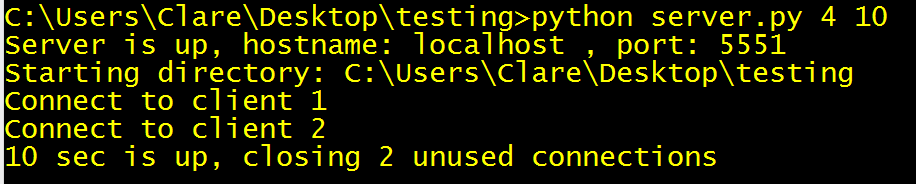
- command line with 4 clients and 10 sec timeout as input arguments

- at start up, identify itself and show starting directory

- when there is a client request to connect, respond to client and print connection status (2 clients in this example)

- after 10 sec, print that the other 2 connections are closed

- the 2 connections with clients remain opened to respond to clients

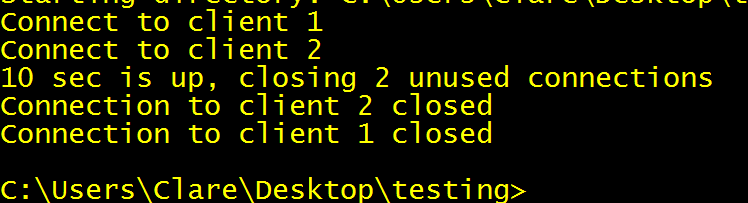


Client windows: The following table shows a test run going back and forth between client 1 and client 2

|  |  |
| --- | --- |
| Client 1 window | Client 2 window |
| - At start up, show id of server , then show current directory  - Loop to print menu, send request based on user choice, and print returned data from server, until user chooses 'q'  - Here the user chooses to see files in the current directory  Note: the directory is printed first, then the files. | In the mean time, client 2 is connected and is idle |
| Now client 1 is idle while client 2 is tested | - Here the user chooses to change directory to dirA Note: the new directory is confirmed with a printout.  - Then the user chooses to list files in the new current directory Note: the new directory is printed, then the files. |
| Back to client 1  - Here the user chooses to list directories recursively  Note: the current directory is still testing, which is different than for client 2  Note: the recursive walk showing levels of subdirectories | In the mean time, client 2 is idle |
| Now client 1 is idle | Client 2  - Here the user continues to change directory down to dirB  Note the confirmation print of the new directory  - Then the user list files of dirB  Note: there is no file or directory in dirB  - Then the user does a recursive list of directories in dirB  Note: there is no directory so no recursive walk result |
| Client 1  - the user lists files in the current directory  Note: client 1 has been in the testing directory the whole time | Client 2 is idle |
| Client 1 is idle | Client 2  - Here the user changes directory back up to the original 'testing' directory  Note the confirmation of being back in the testing directory'  - Then the user does a listing of the testing directory  Note: the listing is the same as when client 1 does the listing, showing that the 2 clients are now in the same directory  - Then the user changes to a non-existing directory  Note: client 2 does not change directory as a result of asking for an invalid directory |

When the user in both clients chooses 'q', each client process terminates, and the server also terminates with an acknowledge message.

Continuing with the server window from above: note the 2 last messages to confirm and identify the client that ended the connection.



Test cases with command line errors:

You can print any descriptive error message you like. I chose the Unix/Linux Usage format as a personal preference.

