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. [Good news: no GUI and no OOP for this lab]

Overview

There are 2 files: server.py and client.py. Both files will run from the command line in a command / terminal window.

The server:

* accepts 2 command line arguments: the number of clients it can serve, and 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

Upon connecting to the server, the client can request the following tasks from the server:

* show files and directories in the current directory
* show all subdirectories (recursively) in the current directory
* go to a new directory
* quit

Recommended steps to work on the lab

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

* Write the server that can work with 1 client, so no need for threading code yet.
  + Accept 4 types of request messages:   
     change directory, list current directory, list recursively the subdirectories, quit

It’s up to you what the request message looks like, but a request is only 1 message from the client.

* + Write 3 functions to do the first 3 tasks and send back the result. The function should send back only *1 result*.
    - *Change directory*: accept a new directory path and change the client's current directory to the new directory. Then send back your choice of 'success' or ‘fail’ status. (What causes a ‘fail’ status when changing directory?)
    - *List current directory*: send back the list of files and directories in the client's current directory. Only for the current directory, don’t go down subdirectories.
    - *List recursive*: send back the list of *subdirectories* from a recursive walk of the client's current directory.
* Write the client to interact with the user and make each of the 4 requests to the server.
  + When the client app comes up, print a menu of available commands that the user can type in, along with an explanation of what the command does:

ls list current directory

lsr list subdirectories recursively

cd dir\_name go to dir\_name

q quit

* + Write a function to prompt the user for a command, read in and validate the user's choice:  
    the command must be one of the 4 choices above, and the command cd should have 1 word after it.
  + 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   
      It’s up to you what the request message looks like, but it 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, use 1 of 2 functions to format the result in a user friendly format and print the result for the user (see sample output).  
    The 2 functions to process the response are:  
    - function to print the response from the change directory request: either print the new directory or print no such directory.  
    - function to print the response from the 2 listing requests: print whether it’s a listing or recursive listing, then print the current directory, followed by the files/directories.
  + Then loop back to prompt the user for another command, until the user chooses the q command.
* To run / test your code, open 2 command / 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. Theoretically the client and server are 2 processes 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 to start the server first. Then type the command to start the client at the other window.

To test your code, create a test directory with 2-3 levels of subdirectories and add a couple files in some of the directories. See sample output with the test directory tree.

[ Welcome to the world of command line debugging, In case you're new to it, here are some tips:   
-- The IDE serves as a text editor only. If there is an error when the code runs, fix your code in the IDE, then *don't forget to save* before re-running the code on the command line.

-- 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 terminal / command window and start again.

-- The up arrow key to re-run the previous command on the command line 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 let you know.

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.
* Use 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 correct 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 that it can serve, and the time before the server times out.

* Since the command line is typed in by the user, do user input validation:
  + there must be 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 error message that describes the invalid case and end the program.
* Make sure to test for different types of invalid input from the user.

Sample output: 1 server window (output in blue) and 2 client windows (output in orange and green)

C:\Users\Desktop\test>python server.py 3 4 start server with 3 clients, 4 sec time out

Server is up, hostname: localhost , port: 5551

Starting directory: C:\Users\Desktop\test test is server’s home directory

C:\Users\Desktop\test>python client.py start client 1

Client connect to: 127.0.0.1 port: 5551

Commands:

ls list files

lsr list directories recursively

cd dir\_name go to dir\_name

Enter choice: ls

Listing of C:\Users\Desktop\test client starts at the server’s home directory

client.py

dirA

server.py

Enter choice:

C:\Users\Desktop\test>python client.py start client 2

Client connect to: 127.0.0.1 port: 5551

Commands:

ls list files

lsr list directories recursively

cd dir\_name go to dir\_name

Enter choice: lsr

Recursive listing of C:\Users\Desktop\test client starts at the server’s home directory

C:\Users\Desktop\test\dirA

C:\Users\Desktop\test\dirA\dirB

C:\Users\Desktop\test\dirA\dirB\dirC

Enter choice:

4 sec is up, closing 1 unused connections connection for client 3 times out

Enter choice: cd dirA/dirB client 1 goes to dirB

New path: C:\Users\Desktop\test\dirA\dirB

Enter choice: ls

Listing of C:\Users\Desktop\test\dirA\dirB

dirC

fileC.txt

Enter choice:

Enter choice: ls

Listing of C:\Users\Desktop\test client 2 is still at test directory

client.py

dirA

server.py

Enter choice: cd dirA client 2 goes to dirA

New path: C:\Users\Desktop\test\dirA

Enter choice: ls

Listing of C:\Users\Desktop\test\dirA

dirB

fileA.txt

fileB.txt

Enter choice:

Enter choice: cd ../.. client 1 goes back up 2 levels to test directory

New path: C:\Users\Desktop\test

Enter choice: ls

Listing of C:\Users\Desktop\test

client.py

dirA

server.py

Enter choice:

Enter choice: lsr

Recursive listing of C:\Users\Desktop\test\dirA client 2 is still at dirA

C:\Users\Desktop\test\dirA\dirB

C:\Users\Desktop\test\dirA\dirB\dirC

Enter choice: q

Connection to client 2 closed when client 2 quits, server acknowledges

Enter choice: cd dirA/dirB/dirC client 1 is still at test directory and goes to dirC

New path: C:\Users\Desktop\test\dirA\dirB\dirC

Enter choice: ls

Nothing found

Enter choice: q

Connection to client 1 closed when client 1 quits, server acknowledges and ends