CSE-3300 Programming Assignment #1

Client and Server Online Game: Jumble

In this assignment, you need to use socket programming to implement an online game Jumble.

If you are not familiar with the game. Read the following Wiki page.

https://en.wikipedia.org/wiki/Jumble

You can download a simple Python implementation of the game from our HuskyCT course page, and modify the code to write both client and server.

You can also download a text file which has a list of English words. This text file will be used by the server to pick words for a client to guess.

Moreover, you can also download both server code and client code and make changes for your purpose. These codes are from the book “Programming Python”. The server is implemented using thread to allow concurrency. Note you need to implement the server side of the Jumble game using thread to allow multiple users to play the game at the same time.

The following Python programming techniques are needed for the assignment.

- Reading content of a text file to a list
- Generating random numbers
- TCP socket programming

Test your code and make sure the server can serve multiple players simultaneously. Note the words should be generated independently for each client, so that normally different clients are guessing different words.

The following pages are from the book “Programming Python”. They contains the detailed information about the server code.

Threading Servers

The forking model just described works well on Unix-like platforms in general, but it suffers from some potentially significant limitations:

Performance
: On some machines, starting a new process can be fairly expensive in terms of time and space resources.

Portability
: Forking processes is a Unix technique; as we've learned, the `os.fork` call currently doesn't work on non-Unix platforms such as Windows under standard Python. As we've also learned, forks can be used in the Cygwin version of Python on Windows, but they may be inefficient and not exactly the same as Unix forks. And as we just discovered, `multiprocessing` won't help on Windows, because connected sockets are not pickleable across process boundaries.

Complexity
: If you think that forking servers can be complicated, you're not alone. As we just saw, forking also brings with it all the shenanigans of managing and reaping zombies—cleaning up after child processes that live shorter lives than their parents.

If you read Chapter 5, you know that one solution to all of these dilemmas is to use *threads* rather than processes. Threads run in parallel and share global (i.e., module and interpreter) memory.

Because threads all run in the same process and memory space, they automatically share sockets passed between them, similar in spirit to the way that child processes inherit socket descriptors. Unlike processes, though, threads are usually less expensive to start, and work on both Unix-like machines and Windows under standard Python today. Furthermore, many (though not all) see threads as simpler to program—child threads die silently on exit, without leaving behind zombies to haunt the server.

To illustrate, Example 12-7 is another mutation of the echo server that handles client requests in parallel by running them in threads rather than in processes.

Example 12-7. PP4E\Internet\Sockets\thread-server.py

```
"""
Server side: open a socket on a port, listen for a message from a client,
and send an echo reply; echoes lines until eof when client closes socket;
spawns a thread to handle each client connection; threads share global
memory space with main thread; this is more portable than fork: threads
work on standard Windows systems, but process forks do not;
"""

import time, _thread as thread                  # or use threading.Thread().start()
from socket import *                            # get socket constructor and constants
myHost = ''                                     # server machine, '' means local host
myPort = 50007                                  # listen on a non-reserved port number

sockobj = socket(AF_INET, SOCK_STREAM)                  # make a TCP socket object
sockobj.bind((myHost, myPort))                          # bind it to server port number
sockobj.listen(5)                                       # allow up to 5 pending connects

def now():
    return time.ctime(time.time())                      # current time on the server

def handleClient(connection):                           # in spawned thread: reply
    time.sleep(5)                                       # simulate a blocking activity
    while True:                                         # read, write a client socket
        data = connection.recv(1024)
        if not data: break
        reply = 'Echo=>%s at %s' % (data, now())
        connection.send(reply.encode())
    connection.close()

def dispatcher():                                       # listen until process killed
    while True:                                         # wait for next connection,
        connection, address = sockobj.accept()          # pass to thread for service
        print('Server connected by', address, end=' ')
        print('at', now())
        thread.start_new_thread(handleClient, (connection,))

dispatcher()
```

This `dispatcher` delegates each incoming client connection request to a newly spawned thread running the `handleClient` function. As a result, this server can process multiple clients at once, and the main dispatcher loop can get quickly back to the top to check for newly arrived requests. The net effect is that new clients won't be denied service due to a busy server.

Functionally, this version is similar to the `fork` solution (clients are handled in parallel), but it will work on any machine that supports threads, including Windows and Linux. Let's test it on both. First, start the server on a Linux machine and run clients on both Linux and Windows:

```
[window 1: thread-based server process, server keeps accepting
client connections while threads are servicing prior requests]
[...]$ python thread-server.py
Server connected by ('127.0.0.1', 37335) at Sun Apr 25 08:59:05 2010
Server connected by ('72.236.109.185', 58866) at Sun Apr 25 08:59:54 2010
Server connected by ('72.236.109.185', 58867) at Sun Apr 25 08:59:56 2010
Server connected by ('72.236.109.185', 58868) at Sun Apr 25 08:59:58 2010

[window 2: client, but on same remote server machine]
[...]$ python echo-client.py
Client received: b"Echo=>b'Hello network world' at Sun Apr 25 08:59:10 2010"

[windows 3-5: local clients, PC]
C:\...\PP4E\Internet\Sockets> python echo-client.py learning-python.com
Client received: b"Echo=>b'Hello network world' at Sun Apr 25 08:59:59 2010"

C:\...\PP4E\Internet\Sockets> python echo-client.py learning-python.com Bruce
Client received: b"Echo=>b'Bruce' at Sun Apr 25 09:00:01 2010"

C:\...\Sockets> python echo-client.py learning-python.com The Meaning of life
Client received: b"Echo=>b'The' at Sun Apr 25 09:00:03 2010"
Client received: b"Echo=>b'Meaning' at Sun Apr 25 09:00:03 2010"
Client received: b"Echo=>b'of' at Sun Apr 25 09:00:03 2010"
Client received: b"Echo=>b'life' at Sun Apr 25 09:00:03 2010"
```

Because this server uses threads rather than forked processes, we can run it portably on both Linux and a Windows PC. Here it is at work again, running on the same local Windows PC as its clients; again, the main point to notice is that new clients are accepted while prior clients are being processed in parallel with other clients and the main thread (in the five-second sleep delay):

```
[window 1: server, on local PC]
C:\...\PP4E\Internet\Sockets> python thread-server.py
Server connected by ('127.0.0.1', 58987) at Sun Apr 25 12:41:46 2010
Server connected by ('127.0.0.1', 58988) at Sun Apr 25 12:41:47 2010
Server connected by ('127.0.0.1', 58989) at Sun Apr 25 12:41:49 2010

[windows 2-4: clients, on local PC]
C:\...\PP4E\Internet\Sockets> python echo-client.py
Client received: b"Echo=>b'Hello network world' at Sun Apr 25 12:41:51 2010"

C:\...\PP4E\Internet\Sockets> python echo-client.py localhost Brian
```

```
Client received: b"Echo=>b'Brian' at Sun Apr 25 12:41:52 2010"

C:\...\PP4E\Internet\Sockets> python echo-client.py localhost Bright side of life
Client received: b"Echo=>b'Bright' at Sun Apr 25 12:41:54 2010"
Client received: b"Echo=>b'side' at Sun Apr 25 12:41:54 2010"
Client received: b"Echo=>b'of' at Sun Apr 25 12:41:54 2010"
Client received: b"Echo=>b'life' at Sun Apr 25 12:41:54 2010"
```

Remember that a thread silently exits when the function it is running returns; unlike the process `fork` version, we don't call anything like `os._exit` in the client handler function (and we shouldn't—it may kill all threads in the process, including the main loop watching for new connections!). Because of this, the thread version is not only more portable, but also simpler.