PIC 16, Spring 2018 – Assignment 10R

Assigned 6/7/2018. Code (a single `.py` file) due 12 p.m. 6/11/2018 on CCLE. Hand in a printout of this document with the self-assessment portion completed at the final exam on 6/11/2018.

In this assignment, you will improve your chat program with the same partner. Most importantly, your send and receive operations will be performed using separate sockets in separate threads in order to improve performance.

Task

Improve your chat program to work as demonstrated in this video. Again, I suggest that you and your partner develop a *single* program together on one computer. Only when it is working well between two Python consoles on the same computer should you begin to test it between your two computers.

1. In the original version of the program, the same socket was used for both sending and receiving; its time was split between the two. Now, we'll need two sockets[1]: one for sending, and the other dedicated to continuously receiving messages.
    a. In your main thread, connect a second socket after the first (copy and paste)
    b. Write a method (either in your main class or a subclass of `Thread`) that continuously receives messages and displays new incoming message in a label.
    c. Invoke this method in a separate thread.

   Just for this you get **50 points**. Demonstrate it by showing me that your program sends messages between two computers with no noticeable delay.
2. It would be nice to see a history of the chat rather than the most recent message only. For **10 points**, add a read-only `QTextEdit` and `append` to it rather than displaying incoming messages in the label. Keep displaying status messages (e.g. "`<connected>`") in the label, though. Preface incoming messages with "`Received: `" and outgoing messages with "`Sent: `".
3. Using a *regular Python console* (rather than IPython console), you will probably get an error message that warns you to "`Make sure 'QTextCursor' is registered…`" when you append an incoming text to the `QTextEdit`. I looked this up online and found out that Qt doesn't appreciate you appending to the `QTextEdit` from a separate thread. Fix this for **5 points**. (Hint: my solution was to have a `QTimer` append any new text to the text box every 0.1s.)
4. Now that you have your outgoing messages recorded in the text box, you don't need to leave them in the `QLineEdit`. Clear the `QLineEdit` as soon as you send a message for **5 points**.
5. The first instance of the GUI to be opened – the one that will act as the server – probably either doesn't show up or isn't responsive while the server socket's `accept` method is blocking. For **10 points**, make the GUI show up and be responsive even before the connection with the second instance of the program has been established. (Hint: I ran the method that sets up the networking in a separate thread. In my program, it's also separate from the thread that continuously receives messages.)
6. You don't want the user trying to send any messages before the connection is established or after it is closed. For **5 points**, disable the `QLineEdit` during these times; it should only be enabled when the connection is ready.
7. If you're like me, at this point you have a boatload of issues that prevent the program from executing properly *an arbitrary number of times* without opening a new Python console. You've probably left one or more sockets open, and you might have left a separate thread or two running. Here are some of the top suggestions I can give you for fixing these issues.

[1] Actually, a single socket can send and receive data simultaneously (in two separate threads). I didn't realize that what I first wrote the code and assignment. So you can get by with just one connected socket if you prefer.

a. Under normal program operation, the easiest sockets to `close` are the server sockets used to `accept` incoming connection requests. Since you only need them once, just remember to `close` them as soon as `accept` returns. If you don't, you may not be able to use these ports again the next time your run the program.
b. It's pretty easy to close the sockets used to *send* data, since these are only used in the main thread. Just remember to close them when the GUI is closed (hint: you saw an event for this in the GUI tutorials…).
c. It's a bit tougher to close the sockets used to receive data, since these are `rece`iving in a separate thread. If the thread used for continuously receiving data is in an infinite loop, it will keep going even after your GUI has closed. (You can check how many threads are running in a Python interpreter before and after you execute your program using `threading.activeCount();` the number should be the same.) My solution involves checking a termination condition each loop iteration, and making sure that termination condition is satisfied when either of following occur:
    i. the GUI is closed, or
    ii. the sending socket of the other computer is closed (which you should already know how to detect from last time, because you had to display "`<connection closed>`").

    Once you're out of the loop, you can `close` the socket.
    (You might have read about daemon threads on your own. Putting the receive loop in a daemon thread might work, but it's not my solution. I wanted the thread to finish and the socket to be closed cleanly, not forcibly.)
d. This should account for everything during normal use. But it's a different story in the abnormal case that the client never `connect`s, that is, if you try to close your program while a server socket's `accept` method is blocking. Trying to `close` it or `del`ete its reference from a separate thread won't work. There are only a few options I can think of.
    i. You could set a timeout on the server socket (see `settimeout`). (I have several reasons why I didn't use this approach; ask if you're interested.)
    ii. You can `connect` to it from a separate thread within the same program instance. Essentially, you pretend that you're the second instance of the program (who, for whatever reason didn't `connect` before the first instance was closed) just to get the `accept` method to return. When that happens, you can `return` from the thread's target method without going any further.

    (Again, you could make sure you only call `accept` in a daemon thread, but this didn't work for me. I think that even if the daemon thread is killed, it's possible that the server socket keeps waiting because `socket` is not implemented entirely in Python; it is just a Python front-end to a lower-level socket API.)

These things can be a real pain to debug, especially without the hints I just gave. I restarted consoles about 100 times (literally) in the process. Hopefully this gives you some appreciation for how tricky threads and sockets can be. These aren't things to toy around with when writing important code. You need to know what you're doing. I'll assign the last **15 points** according to how robust your program is.

Self-Assessment

Demonstrate the operation of two instance of your program on a single computer *and* between two computers by taking a video with your cell phone. (The single computer test is for catching bugs in part 7.) If you want credit for robustness, indicate what you are doing to test the different scenarios addressed in part 7. Upload it somewhere (e.g. YouTube, Dropbox) and write the address below (and include as a comment in your code). Check off the tasks you completed above and indicate your total score below.