CS 346 (Fall 21): Cloud Computing

Project #2

due at 5pm, Thu 16 Sep 2021

Overview

In this project, you will write a few small programs that will give you a little experience with socket programming. They are **very simple,** and you're going to see their limitations - but it's a good start.

The first one, `dumb_net.py`, is going to be like `telnet`, but even simpler: your program will only read a single line from the user, never more than that - and when it sends the line to the other side, it will add a special line ending. This will allow you to make HTTP requests of web servers, just like we've done with `telnet`.

The second one, `one_to_one.py`, will be the most dumb of chat applications: a single server will talk to a single client. It won't support multiple clients at once. And worst of all, it won't let you type freely; the only thing that it will allow is for the endpoints to exchange messages!

The third one, `echo_server.py`, will be an application which simply replies back to the client exactly what was sent. It will connect just fine to the client side of `one_to_one.py`, but it will never say anything new; it will simply send back what it was given. But the big advantage of this program is that it will be able to talk to multiple clients at once, because it will be multithreaded.

Through these programs, you will get a very basic understanding of how sockets work, and also an understanding of why multithreaded operation is so critical for network tools.

Running Your Code

We'll be writing standard Python code in this project, but we'll be running it from the command line. If you want to run it on your own computer (not inside a container), that's permissible - but you'll need to use your terminal (not your IDE) to run it.

Beware that, once we start writing code that acts as a server, you may get a firewall warning. This is entirely normal - your computer probably hasn't seen you use Python as a network server before, and it wants to check to see if you're aware what's going on. Give Python permission.

Of course, I would love it if you developed your code in a container - it will be great practice for the future. Use a UNIX text editor, and get used to running commands from the shell. But **beware!** Your containers get thrown away when you're done with them - and this happens automatically if you use the `--rm` option on `docker run`. Don't make a mistake and **accidentally destroy your source code!** Be careful to keep a copy of it, outside the container, for safekeeping.

Requirement: Shebang Line

We're getting practice for code we're going to write that has to run when we're not around. Get into the habit of adding a shebang line to every script. **If you don't, you will lose points.**

Debug Output

You are **encouraged** to add debug `print()` statements to your program. I will not be auto-grading this, so I don't have a defined output format; thus, you are allowed to print whatever you want. You will find, especially when you are just learning, that having statements like "Connection completed" in your output will really help you understand what's going on.

`dumb_net.py`

Your first program will be the dumbest of network clients: it connects, sends a single line of input, reads any responses, and then terminates. Once your code is working, you'll be able to use it to connect to a web server, like we did with `telnet` during class - but this will be **much** more limited than even `telnet`.

> **ALERT**
>
> Please, **do not connect to public servers** while testing your code! Your early versions of the code are going to be missing many features and not working well; why waste other people's resources by making broken connections to their web servers? Instead, run your own webserver inside a container (as we've done in class), and test your code against that.
>
> Once your code is working properly, feel free to test it a couple of times against a public server.

Your program must take two command-line parameters (just like telnet): the first is the host to connect to, and the second is the port. (If you didn't use command-line parameters in your Python programs before, see the Appendix to this spec.)

Your program must connect to the given server and port, read a single line of input from the keyboard, and then send it to the server. Note that, in order to use this as an HTTP client, your line will need to have a line ending; in order to be valid HTTP you **must** send the string `"\r\n"` (not just `"\n"`) as the line ending.[1]

Receiving the Response

After you send your one line of data to the server, you must read **all data** sent back to you. Note that you **must not** make assumptions about how much will be sent; you **must** write a loop which reads however many blocks are sent. Keep looping until you've hit the end of the data.

Note that it is common (but not guaranteed) that the server will send a long chunk of data, all at once - meaning that on every single call to `recv()`, you will get a full buffer, until the end. At the end, you will receive a partial buffer (however much is left over) and then an empty one (to mark the end of the data).

However, **do not assume** that this will always be the case. It is **perfectly valid** for a server to send you data in small pieces; so long as you're not getting an empty buffer from `recv()`, you must continue to ask for more.

(spec continue on the next page)

[1] When I found this out, while preparing this project, I was quite surprised. If this was necessary, how was it possible that `telnet` worked as an HTTP client? I did a little research, and found out that `telnet` always sends `"\r\n"` as its line ending, no matter what the line ending is on the client side: `https://www.freesoft.org/CIE/RFC/1123/31.htm`

`one_to_one.py`

In the second program, you will write both a server and a client in the same file. As before, the program will take two parameters, a hostname and a port. However, in this case, if the hostname is the special string `server`, then you will create the server instead of acting as the client.

Your server must bind itself to the required port; for now, use `"0.0.0.0"` as the address, so that your server will respond to connections from any address. On the client side, attach to the host and port given. **Do not hard-code the hostname on the client side.** While you may find that you're always using the same hostname every time you run a test, get used to having a parameter which supplies it. After all, why write a tool which has more limitations than are really required?

Once connected, the server and client work the same way: they alternate between reading a line from the user (and sending it down the socket) and then reading one response from the socket, and printing it to the user. However, they start at different points in this loop: when they first connect, the client will be reading from the user, and the server will be waiting for the first message from the network; when the client finally sends a message, they will trade roles.

Keep the connection open until the user types a blank line of input, or until you receive an empty buffer from the socket.

On the client side, when the connection closes (or you close it), you may terminate the program. On the server side, when the connection closes, you must go back and `accept()` another connection.

Observations

Perform an experiment with your code, once it's working: run the server in one terminal, and the client in a second. Then open up a third terminal, and open up **another** client at the same time. Try to use both clients, and see what happens.

What does the second client see? What happens to the second client when the first client closes? Can you explain why?

Also, do some experimentation with a single client. You've realized that, as things are designed currently, you must type one line of input at the client, and then one line of input at the server, and then one at the client, going back and forth. What happens if you type several lines of input at the client before you type anything at the server? What do you see?

Record your observations, along with your explanation of why they happen, in a PDF file. Name it `blocking_calls.pdf`, and **submit it, along with your code.**

(spec continue on the next page)

echo_server.py

Your third program will be a replacement for the server from the previous program. **You do not need to write a new client;** the client from the previous program should connect to this server just fine. This server, however, has the ability to talk to **many clients at once.** The downside is that it will no longer read any input from the user (so you can't use it to chat); instead, it will simply echo back to the client anything that it was sent.

To do this, we will use the `threading` library in Python. This library allows us to do another thing while a blocking call (such as `accept(), recv(),` or `sendall()`) is waiting. (Note that this does **not** allow us to do two things at once, however.)

`threading` has many useful features and calls, but for this, we're going to do something very simple: every time that you `accept()` a new socket, I want you to do all of the `send()`s and `recv()`s for that socket on another thread. To do this, you will need to make sure that all of the processing for your socket (including closing it at the end) is inside a function; for this example, we'll call the function `worker()`. The function must take one or more parameters; for this example, I'll assume that it takes a single socket, plus one other piece of data:

```
def worker(sock, other_data):
    ... does whatever is necessary ...
    ... expect a loop here ...
    ... calls sock.close() before it returns ...
```

Now, in our main thread (that is, where we're doing the `accept()` calls), we can kick off a new thread like this:

```
threading.Thread(target=worker, args=(sock,"abc_123")).start()
```

After this, the main thread, and each worker thread, will all share the processor; while only one can run at any given moment, as soon as a thread blocks, another thread can run (if it has something to do).

Appendix: Python Command-Line Arguments

In Python, you can access the command line arguments to your program through an array of strings. Import the `sys` library, and access the array `sys.argv`. Element `[0]` of the array will always be the name of the script (exactly as the user typed it), and the rest of the elements (if any) will be the arguments, in order:

```
import sys

print(f"Command name:     {sys.argv[0]}")
print(f"First argument:   {sys.argv[1]}")
print(f"Second argument:  {sys.argv[2]}")
```

(spec continue on the next page)

Appendix: Understanding `threading`

Not sure exactly what the `threading` library is doing? Try out the following code, and see what happens:

```
#! /usr/bin/python3

import threading
import time

def print_some_stuff(message):
    for i in range(10):
        time.sleep(1)
        print(message, i)

def main():
    threading.Thread(target=print_some_stuff, args=("abc",)).start()
    threading.Thread(target=print_some_stuff, args=("def",)).start()
    threading.Thread(target=print_some_stuff, args=("foo",)).start()
    threading.Thread(target=print_some_stuff, args=("baz",)).start()
    time.sleep(15)    # if the main thread terminates, the whole program dies

main()
```