COEN 177L

Omar Garcia

Final Project

Testing the Performance of Multi-threading Connections

The project that I decided to undertake for my final assignment was initially large in scope: I wanted to implement a software-defined satellite with the classical readers-writers problem and test its feasibility. While that project may have been too large in scope, in the efforts to complete it I ran across another set of studies that I thought could lead to interesting conclusions: how is a server running with one connection per thread affected (speed-wise) when the number of concurrent threads increases?

My experiment needed, first, to create an application to allow connections from other computers and to create multiple threads. For this, I used C++, along with the Berkeley socket API (for TCP/IP connections) and the std::thread library (for concurrency). I then wrote 2 classes: the "network" class and the "worker" class. My pseudocode then goes as follows:

```
server-side:
initialize global variable (num_connections) to 0
    main():
        initialize global variable and mutex (num_connections) to 0
        create new instance of "network" class w/ port number
        call run() method of "network" class

    network class(port number):
        create and bind a socket on the given port
        run():
              while(true):
                    listen for incoming connections
                    accept an incoming connection
                    create a new “worker” for that connection
                    pass new connection’s file descriptor to the “worker”
                    under mutex: ++num_connections

  worker class(socket_fd):
        set up the given connection to timeout on failure
        handle the connection using recv() (blocking I/O) and send()
        when connection ends:
              under mutex: --num_connections

client-side:
  main(IP address):
        create a socket
        connect to the given IP address
        t_start = system clock time now
        send messages and receive responses,
              then close the connection
        t_end = system clock time now
        total system time =  (t_end - t_start)
        print(total system time)
```

The actual code will be attached as separate files to this report (named as “sudosat.cpp”, “network.hpp”, “worker.hpp”, and “test_client.cpp” respectively). A few important notes: the use of the “num_connections” condition variable is twofold: it becomes 0 when there are no connections remaining, so it is a good check to confirm that all connections have been completed, and it forces the threads to interact with each other (i.e. wait for access to the variable under the mutex). I wanted to measure any delays in the connection between the client and the server, so I made sure that the client’s calculated time difference was in terms of “wall-clock time” (i.e. real time) elapsed -- as opposed to the more commonly-measured CPU time (since my client is I/O-bound, this would not be very useful).

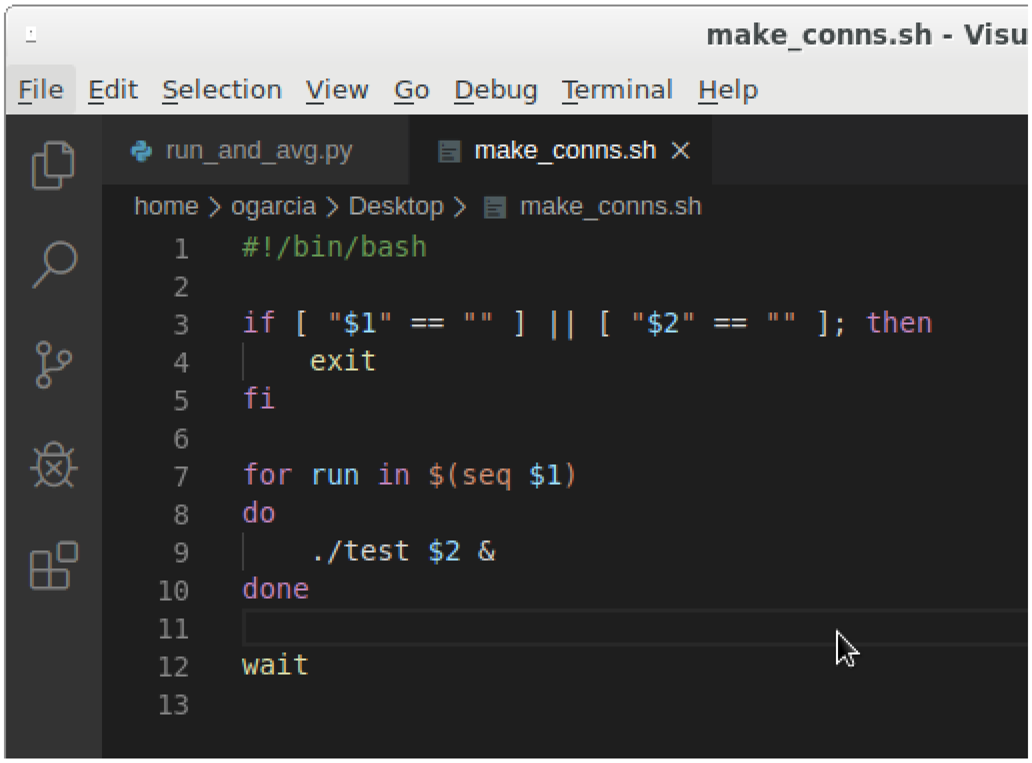
Once the ability to connect externally and the ability to handle multiple connections was verified, I decided to attempt to track the performance of my server when handling multiple concurrent connections; but I quickly realized that 2 issues prevented me from readily testing (taking, for example, the test case with 32 concurrent clients):

- It would be virtually impossible for me to run 32 clients in parallel by hand;
- and having to average out 32 client’s runtimes (in milliseconds!) is very time-consuming.

So, I wrote some helper programs that help to illustrate the OS concept of inter-process communication -- data travels from the network protocol stack to C++, which passes the results to the shell, which passes the results to Python.

The shell program written takes in 2 inputs: the number of concurrent clients, and the destination IP. (One could run it with, for example, `source make_conns.sh 32 192.168.0.1`).

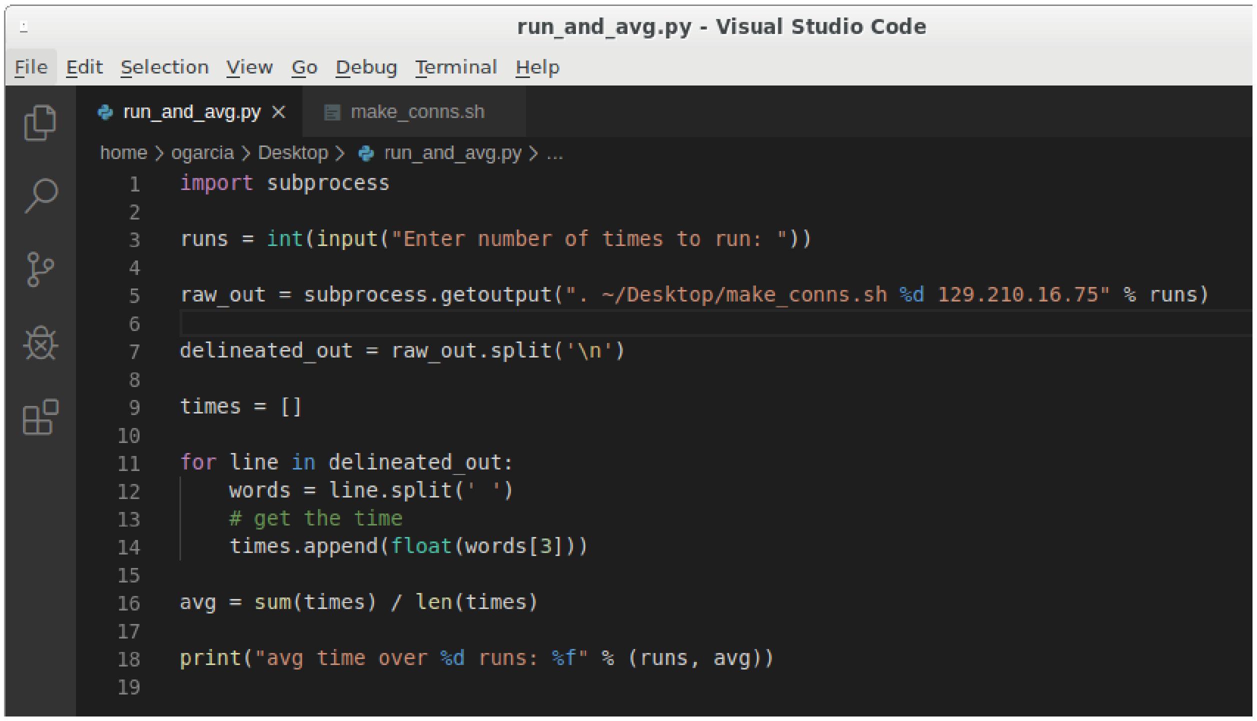
It then uses the '&' key to concurrently run ./test (the output file of compiling "test_client.cpp") the number of times given, and passes the IP to the output file as a command-line argument.


make_conns.sh - Visu
File Edit Selection View Go Debug Terminal Help
run_and_avg.py
make_conns.sh
home > ogarcia > Desktop > make_conns.sh

```
#!/bin/bash

if [ "$1" == "" ] || [ "$2" == "" ]; then
    exit
fi

for run in $(seq $1)
do
    ./test $2 &
done

wait

```

Figure 1: The shell script that runs the client in parallel.

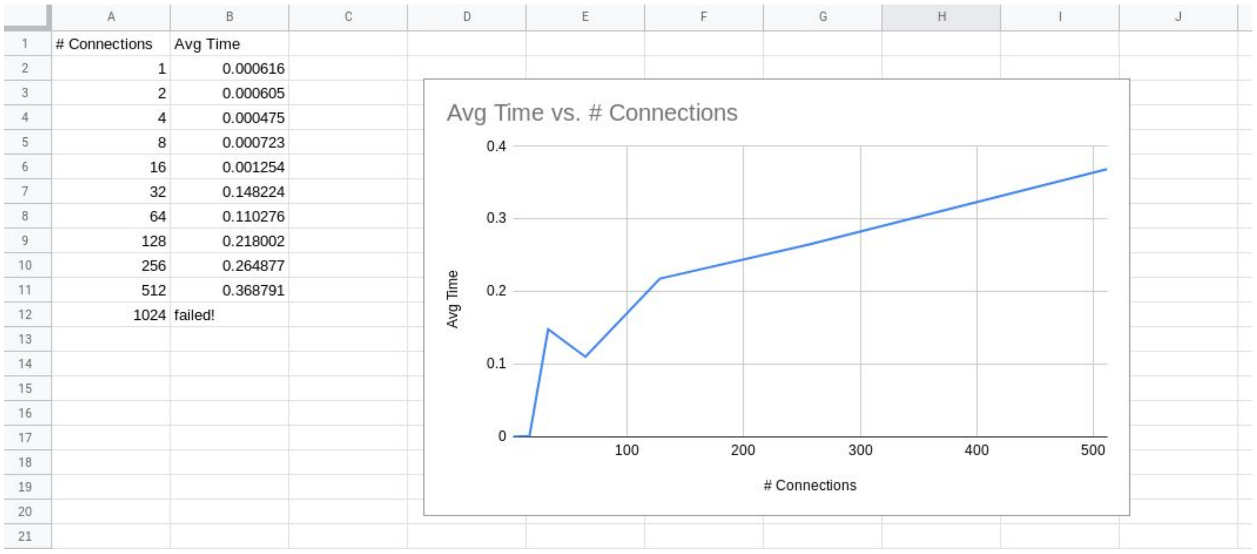
The Python program written is quite simple: it asks the user how many connections to run, and calls the above shell script with the user's response as an argument. After running the shell script, Python takes the script's output (which is the output of the C++ client) and extracts the runtime of each client, then averages and prints it.


run_and_avg.py - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
run_and_avg.py
make_conns.sh
home > ogarcia > Desktop > run_and_avg.py > ...

```python
import subprocess

runs = int(input("Enter number of times to run: "))

raw_out = subprocess.getoutput(". ~/Desktop/make_conns.sh %d 129.210.16.75" % runs)

delineated_out = raw_out.split('\n')

times = []

for line in delineated_out:
    words = line.split(' ')
    # get the time
    times.append(float(words[3]))

avg = sum(times) / len(times)

print("avg time over %d runs: %f" % (runs, avg))
```

Figure 2: The Python program written to do data parsing.

Then it came time to do testing and find out how my server responded to multiple connections. After running many different sizes of connections, I grabbed the data shown below.



# Connections	Avg Time
1	0.000616
2	0.000605
4	0.000475
8	0.000723
16	0.001254
32	0.148224
64	0.110276
128	0.218002
256	0.264877
512	0.368791
1024	failed!

Figure 3: The data retrieved from my program suite.

The data pretty clearly shows that, as the number of concurrent connections (i.e. threads) increases, the amount of time spent responding to each request increases. This is likely due to a few reasons:

The first is how the program does I/O in the first place; recv(), in contrast to other implementations of I/O retrieval, is blocking I/O, which means that it takes control of the program until the I/O operation completes. With a large number of concurrent clients, however, the number of concurrent I/O operations is very large, which may lead to some of them getting buffered by the OS and thus an overall delay in sending or receiving to any client.

The second is the inter-thread use of the condition variable for the number of connections: when each "worker" thread reaches the end of its execution, it tries to grab the lock corresponding to that variable so it can lower the count by 1. But if there are a large amount of connections attempting to close at the same time, there may be a wait time associated with attempting to get access to/hold that lock.

There is also some variance in the data, particularly around the data point of 64 concurrent connections. This could be due to any number of reasons, the most influential being that the OS will allow different processes to run at different times and for different lengths of time for each run that is performed. All of those unique changes affect the runtime, which is typically small (around milliseconds) and thus more susceptible to variance. Variance also occurs on the client side: Running processes in parallel using the shell essentially means a lot of calls to fork(), which depending on the OS's load could execute almost simultaneously or not.

In conclusion, the running of multiple concurrent threads has shown to have a negative impact on per-connection performance (as well as being a resource drain on the OS, as can be shown by the program running out of memory when trying to respond to 1024 client simultaneously). This is an interesting finding and is a good explanation for why other server implementations (e.g. event-driven and nonblocking I/O, thread pooling) are more typically used for commercial applications.

<u>To run the provided C++ programs (make sure all are in same directory):</u>

```
>> g++ -std=c++11 -pthreads -o sudosat sudosat.cpp
>> g++ -std=c++11 -o test test_client.cpp
>> ./sudosat [PORT NUMBER]
>> ./test [SERVER IP ADDRESS]*
```

* port number in “test_client.cpp” is defined as 5000, so change it if using some other port number