Advanced Programming 1 - 89-210 - Ex5

January 15, 2017

Due date: 16.1.16

1 Intro

In this exercise, we wish to scale the server by handling more than one client (driver). This could be a major change, or not, depending on how you planned the previous exercise. As we scale, we would like also to make better use of our server strength, by using its multiple cores (Even if you have one core it is fine, as usually servers are hosted on other computers which suits the application purpose. Besides, we would like to practice multi threads programming). Another change will be the migration to TCP protocol, which fits best this scenario.

2 Implementation

2.1 Multiple Clients Support

If you planned correctly, you should be able to transfer your program to support multiple clients with minimal changes. Even if not, it shouldn't consume too much time. Practically, what differs the input from last exercise regarding this context is the number after the '1' command (which indicates the receiving of new drivers). You need to support numbers other than 1 (after the first one). This means that we should be able to enter for example 2, and receive, one after the other, 2 different clients (drivers). The rest of the application should also be able to handle this extra driver like sending him trips, be aware of its location etc...

2.2 Better Availability Support

As we discussed in class, a server is a computer which has to be available the maximum time possible. In contrast to last exercise, where we had only specific times where we handled requests from the client, we now wish to be available as long as possible. For this reason, every part of the server which is responsible to communicate with clients will have its own socket in its own thread. This includes:

- The socket which communicates with every client.
- The socket which receives new drivers can still be open only for the time sequence after receiving the signal of new drivers, and handle them sequentially.
  - It is important of creating a new socket for every client (as we spoke in class).

- Think what can go wrong now with several threads open in parallel? How the main thread will know there is a new message awaiting in the thread handling a particular client?
- The sockets in this exercise have to use the TCP protocol. The relevant code is attached as last time. You may use it, make adjustments or not...
  - Pay attention, that you can not use it as is. It was built for another application which supported only one client. Make the relevant changes for supporting more than one client as we spoke in class. If it's easier for you, just build your own tcp class.

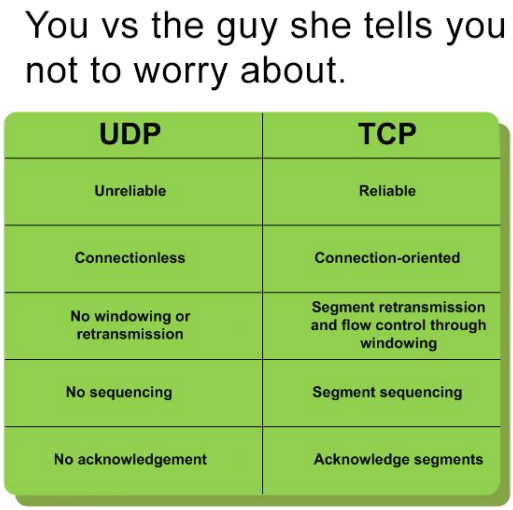
2.3 Calculate Fast, Parallel

Notice that, there is some time window between receiving a trip, until assigning it to a driver. Now that we learned threads, let's separate the task of calculating the trip path from the general flow of the program.

What you need to do in practice is to open a thread for every new trip in which you will calculate the path between the two points of the trip. When the time arrive (the begging of the trip) we'll have to make sure that the calculation is over (and if not, wait for it) and then we'll be able to send it to the client.

Let's make things more interesting. The new limitation of the grid size is changing, the new limit is 1000*1000.

- Make sure the time to calculate an ordinary path between the two extreme edges takes you milliseconds (on the original 10*10).
- Same thing for 100*100. You shouldn't even fill it.
- On a 1000*1000 grid it won't be so fast. Could take several seconds. But, as we distributed our program, we can move on with it and not get stuck with the program until it finishes to calculate the path.



3 Example

input/output example:

server:
3 3
1
1,1
3
0,1,H,R
3
1,1,S,B
2
1,0,0,0,1,2,30,4
2
0,0,0,2,2,1,20,1
1
2
9
9
9
9
9
9
9
4
0
(2,2)
4
1
(0,1)
7

Translation:
3 3 - a 3 on 3 grid
1 - one obstacle
1,1 - at point: (1,1)
3 - insert a vehicle
0,1,H,R - id=0,Cab_Type=1 - Normal Cab,Manufacturer=Honda,Color=Red
3 - insert a vehicle
1,1,S,B - id=1,Cab_Type=1 - Normal Cab,Manufacturer=Subaro,Color=Blue
2 - insert a trip
1,0,0,0,1,2,30,4 - id=1,x_start=0,y_start=0,x_end=0,y_end=1,num_passengers=2,tariff=30, time of start - 4
2 - insert a trip
0,0,0,2,2,1,20,1 - id=0,x_start=0,y_start=0,x_end=2,y_end=2,num_passengers=1,tariff=20, time of start - 1
1 - expecting drivers
2 - expecting two drivers
9 - new time: (1). Assigning ride 0 to driver 0. No movement

9 - new time: (2), expected new location (driver 0): (0,1)
9 - new time: (3), expected new location (driver 0): (0,2)
9 - new time: (4), expected new location (driver 0): (1,2). Assigning ride 1 to driver 1
9 - new time: (5), expected new location (driver 0): (2,2). (driver 1): (0,1)
9 - new time: (6), no movement
9 - new time: (7), no movement
4 - get driver location
0 - driver with id=0
4 - get driver location
1 - driver with id=1
7 - exit

4 GIT Usage

Keep practicing. Starting to understand the benefits? No more mails. No more dropbox backups. Lets make it more interesting.
Start playing around with branches. We spoke on the branching model in class and the correct use case. Here is a blog post explaining what we covered in class with more details: `http://nvie.com/posts/a-successful-git-branching-model`
In your log file you submit, I want to see your usage of branches.

5 Code, Code, Code

5.1 Structure

As your code grows larger and larger, we wish to arrange it a little bit. C++ doesn't have packages like JAVA does, but we can still arrange our classes hierarchically. Make use of folders for aggregating common classes together.

This is not necessary (and won't be graded), but I believe it will just help you.

5.2 Logging

A very major part of programming is debugging. When entering a multi-threaded environment debugging just gets harder. IDEs aren't optimal for debugging multi-threaded programs. One option is to make usage of printing. lots and lots of them. But it got several disadvantages. A common practice is logging. It is not a small subject but not a big one neither. Logs can be produced by you, or by using an external library. You can read about it here: `https://en.wikipedia.org/wiki/Tracing_(software)` and for an opinion blog: `http://vasir.net/blog/development/how-logging-made-me-a-better-developer`. There are several libraries for logging in every language. Specific in C++ some common ones:

- boost: `http://www.boost.org/doc/libs/1_62_0/libs/log/doc/html/index.html`
- easylogging: `https://github.com/easylogging/easyloggingpp`
- and more...

There are pros and cons for every one of them. chose wisely. Pay attention that the second one latest version uses C++11, so if you are planning of submitting with it, use an older version which make use of C++98.

This is not mandatory, but it's an extremely important tool programmers have and use on a daily basis.

Usage of logging is this exercise can lead up to 10 bonus points. Please mention that in an extra file with your submission named *log_usage.txt.* Explain which log software you made use of, to what extent, what features you used, what kind of logging you used (INFO, DEBUG, ...) and if it helped you. If you do make use of logs, submit your code with all your logs messages. A nice feature logs libraries usually have is the option to shut down the logs from a configuration file. Make sure to shut down all your logs so they won't appear when we check your exercise.

6 Submission:

A zip containing several items:

- All your code in hierarchical way. The makefile should be outside the directory so we could easily compile your code from your submission folder.
- Details file.
- Log file of your git usage like last time. This time I expect to see usage of branches.
- Extra '*log_usage.txt*' file for the bonus part (optional).

7 Notes

- Don't be late.
- Submit your work via the submit system, with your personal user account. ONLY ONE SUBMISSION PER GROUP. A duplicated submission will be penalized. Same goes for a submission without 'details.txt' file (or deprecated) [in every submission].
- Change your compiling system to generate two executable files: 'server.out' and 'client.out' so I can run them as following:
  - user@user$ ./server.out 40000
  - user@user$ ./client.out localhost 40000
  - user@user$ ./client.out localhost 40000
  - ... (as many as desired)
  - in different shells of course
- Don't forget to add the compiling flag for using the posix threads: '-pthread'.
- You have several thread using same resources (for example shared arrays which indicates for a request from a client). Make sure you handle them correctly by using mutex, and try avoiding deadlocks.
- Debugging in a multithread environment is harder then usual. Printing can help you (And logging much more).
- As last exercise we spoke about a time limit of 5 seconds, and this time you can get bigger grids, which could take more time to calculate, the new limit (the total run time of the program) is 60 seconds. But, it won't really need all of it (it's the upper bound).

8 Grading Notes

- General working flow - communication between client and server and correct expected output (*60%*)
- Git usage (*20%*)
- C++ correct code writing (*10%*)
- Releasing all allocated memory (on both server and client) (*10%*)
- Up to 10 extra points (Bonus) for using logging - depends on the extent and quality of the usage.