Advanced Programming 1 - 89-210 - Ex4

January 2, 2017

Due date: 2.1.16

1 Intro

In this exercise, you will be adding a client and a server to your program using sockets. The client represents a driver, which registers to the 'Taxi Center', and gets trips based on its location. The server is left with the rest of the logic from the previous exercise. This exercise you will use UDP connection we studied in class. Please make sure you understand this concept. In the following exercise we will switch to TCP connection as it is more appropriate to this scenario (why?). We want you to experience both connections and for this reason we will use both.

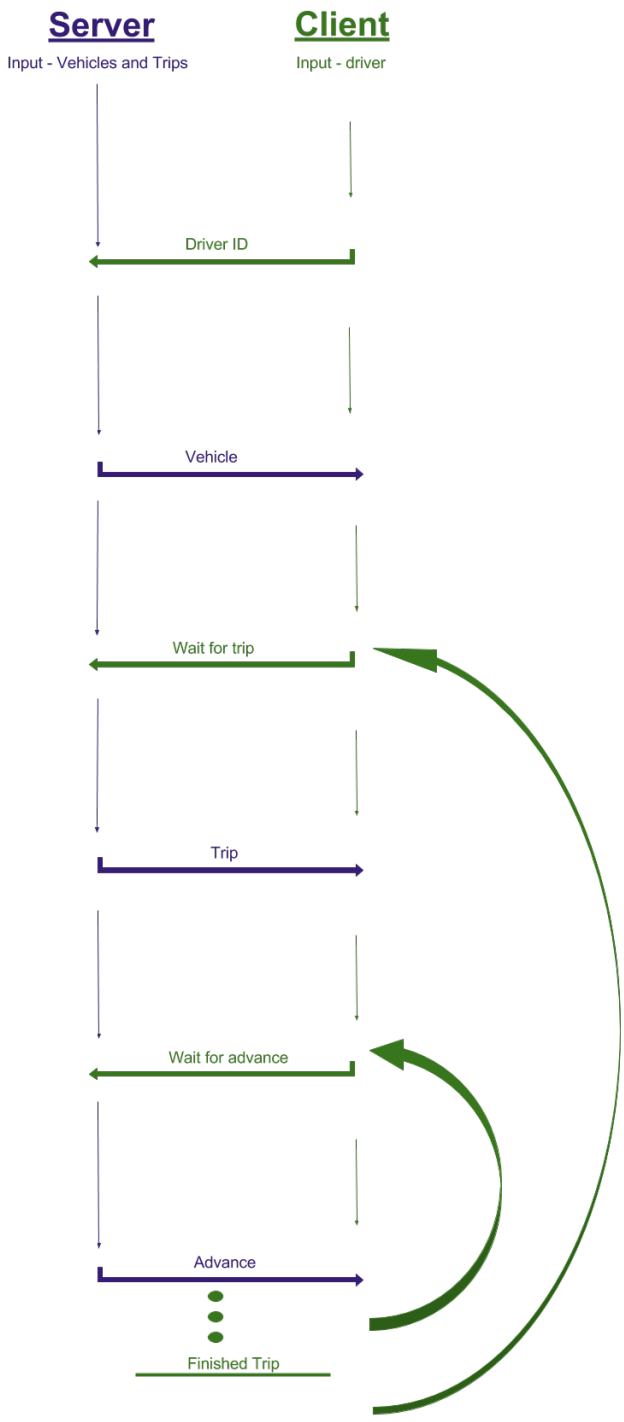
2 Implementation

2.1 Client

The client here will be a driver. Every client will run in a different process. First, it needs to receive the driver's information from the console, exactly as in last exercise - except one change (explained in the next section). It will then create a driver object, and send it to the server via the socket you created. The server will respond with the corresponding vehicle. Afterwards, the client will wait for a trip. Then, every time he gets a 'go', he will move one step on the grid, and wait for next 'go'. This is opposed to previous exercise. The general interaction between the client and the server is displayed in the diagram below.

2.2 Server

Most of the server code is already implemented from last exercise. You need to remove the reception of drivers inside the server, and instead, having a new operation - which will get the number of drivers it's expecting to get, and register them inside its lists. The process of receiving a driver, will follow a response in the form of a vehicle. The rest of the operations should stay the same.



General flow of server with multiple clients (should be the same with one client)

2.3 Implementation details regarding both sides:

- You are provided with UDP classes and a generic Socket class. You are free to use the provided code, as well as to change it to work better for your needs, or to ignore it all together. If you do use them, make sure you understand how they work and how to use them. Make sure to use them correctly (polymorphism) for minimal code change next exercise. The code can be found under the resources section.
- Another change introduced in this exercise is time. You need to maintain some 'clock' which will indicate the time of the world. It will start from 0, and each advance command you'll need to make a time dependent operation (like advancing the vehicles assigned with a trip).
- As the server won't handle the creation of drivers anymore, the operation '1' from last exercise is changing. The 1 operation will be restricted to only 1 usage, which will happen before the beginning of the movement. So, the first '1' indicates the beginning of receiving new drivers, afterwards the number of expected drivers (again, in this exercise only 1). Afterward we need to expect to receive via a socket the drivers and send them back the requested vehicle (by id). Next exercise we'll deal with more than 1 client (and driver).
- In this exercise I will test you only on one client and one server. In next exercise, you'll need to support more than one client, so:
  - You should prepare for this change in the general flow.
  - If you have time, you can even do it this time, if you plan correctly, the changes in the code would be minimal.
- The basic data transfer is just a bit vector, so we will use serialization for turning this to and from classes. Make use of the examples we have seen in class using the boost library (or some other solution, if you prefer)
- You can assume that the stage of receiving drivers will start only after all the other relevant data (of vehicles) was already registered.
- In contrast to the previous exercise, here we won't suddenly arrive to the destination point. We advance the 'clock time' each turn - every time the server gets a '9' signal. So each advance (='9') the driver will move its vehicle a block based on its vehicle implementation (Taxi = 1 block, Luxury = 2 blocks).
  - Time of the game start from 0. Every time we get the '9' signal, time advances by 1.
  - When the time of the trip arrives, the TaxiCenter will only send the relevant taxi the relevant trip and only next time we get '9' he will make the first move.
- There is a time limit to the program of 5 seconds (for both server & client).
- When the server gets its closing signal, it will send some message to the client so he will know to shutdown itself as well.
- Make sure to close all sockets when the server gets its exit status.
- Make sure to delete all allocated memory (in both client and server).

- No need of submitting unit-tests this time, but you'll be asked for updated unit-tests submission in the final exercise - which should cover all your code. You should definitely keep them updated as you work on your code (even just for your own debug purposes)

2.4 Misc

The input to your client and server should look like the following:

```
$ ./server.out (port)

$ ./client.out (ip) (port)
```

An easy way to test your program is by running it on the same computer. You can address it by *localhost* or *127.0.0.1*. But, there's no reason it won't work on different machines as well.

Be aware that usually, on professional environment, those parts run of different machines. Everything is based on the requirements, and the scaling you need for your application.

3 Example

input/output example:

```
server:
3 3
1
1,1
3
0,1,H,R
2
0,0,0,2,2,1,20,1
1
1
9
9
9
9
9
2
1,2,2,0,1,2,30,8
9
9
9
9
9
9
4
0
(0,1)
7
```

client:
0,30,M,1,0

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
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
1 - expecting one driver
9 - new time: (1). Assigning the ride to a driver. No movement.
9 - new time: (2), expected new location: (0,1)
9 - new time: (3), expected new location: (0,2)
9 - new time: (4), expected new location: (1,2)
9 - new time: (5), expected new location: (2,2)
2 - insert a trip
1,2,2,0,1,2,30,8 - id=1,,x_start=2,y_start=2,x_end=0,y_end=1,num_passengers=2,tariff=30, time of start - 8
9 - no movement, new time: (6)
9 - no movement, new time: (7)
9 - new time: (8). Assigning the ride to a driver. No movement.
9 - new time: (9), expected new location: (1,2)
9 - new time: (10), expected new location: (0,2)
9 - new time: (11), expected new location: (0,1)
4 - get driver location
0 - driver with id=0
7 - exit

- The driver's input is identical

More examples on piazza resources (and submit system)

4 GIT Usage

It's now time to start practicing the usage of git. You already know the basics (learned in class), but you're encouraged to continue and learn more from the vast information on the Internet. One site I personally like (and always goes back to) is: `https://www.atlassian.com/git/tutorials`.

Chose some platform to upload your code to. For example you have Atlassian BitBucket - `https://bitbucket.org` for free, or Github `https://github.com/` (checkout their free student's pack). Register (Every person has to register so I could see the work everyone did afterwards) and start working.

Submission:

A zip containing several items:

- All your code inside a directory. Compilation should be done using a makefile (ONLY). The makefile should produce 2 programs: *client.out* and *server.out*.
- A details file.
- A description file of your git usage. This can be produced using the terminal (inside a git registered directory) as follows:
  - git log –graph > log.txt
  - The 'log.txt' file is the output of that command which should be submitted inside the submission zip.

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].
- For Compiling with boost serializable library:
  - For Makefile: just compile with the "-lboost_serialization" flag
  - For CMake: (there are several ways, this is what worked for me)

```
set(BOOST_ROOT CACHE PATH "/usr/include/boost/") # or equivalent location
find_package(Boost 1.58.0 COMPONENTS serialization system) # or equivalent version
add_library(core ${core_SRCS})
add_executable(exe main.cpp)
target_link_libraries(exe core ${Boost_LIBRARIES})
include_directories(${Boost_INCLUDE_DIR})
```

- 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 5555
  - user@user$ ./client.out localhost 5555
  - in different shells of course

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%*)

Good Luck!