Lab 1a) Line based TCP client

Introduction

A line-based Application Communication Protocol (ACP) is quite common to use as in simple applications.

Before you start implementing, **READ** the entire assignment.

Requirements

You are to write a simple client. The protocol is **TCP**, and on the application-level, the protocol is a **simple character-based** protocol. A simple character-based protocol is just using standard ASCII characters, i.e. CHAR. Furthermore, messages are delimited by a newline, '\n'. Of course, this causes some limitations and an ASCII based protocol is not really what should be used today (look at UTF8), but for this **SIMPLE** assignment, it's sufficient. Our focus is communication, not how to handle all languages in the world. Furthermore, use the reference server to test your client. The reference server (subject to change), is reached via IPv4 13.53.76.30:5000 or XYZ:5000 (TBD). The server handles ONE client at a time. But multiple clients can queue up for service, cf. Listen().

Furthermore, newline == '\n' or 0x0a.

When a client connects, then the server sends a string which protocol(s) it supports. In this case, the string should be "TEXT TCP 1.0\n". If the server handles multiple versions, each version is sent on a separate line. When the last version has been sent an empty line with just a newline '\n' should be sent.

```
'TEXT TCP 1.0\n'
'TEXT TCP 1.1\n'
'\n'
```

If the client accepts ANY of the protocols, it has to respond with 'OK\n'. If it does not, then it should disconnect.

In the next stage, the client is to read a string that is randomized at the server, see the example further down. The sting is constructed from three parts; '<OPERATION> <VALUE1> <VALUE2>\n'.

OPERATIONS are;

- add/div/mul/sub for these VALUE1 and VALUE2 have to be integers.

- fadd/fdiv/fmul/fsub for these VALUE1 and VALUE2 are floating-point values.

Once the client has read the string, it should calculate the result and send the response back to the server, in the corresponding format, integer, or floating-point. For floating-point use "%8.8g "for the formatting. Note that div would normally result in a floating-point value, however here it's truncated to an int.   For example, if the result is 13 the string sent from the client should be: '13\n'

The server reads this and compares it with its result, if they match the server sends back a string with 'OK\n', otherwise 'ERROR\n'. Note when comparing floating-point values, we cannot do an absolute comparison, but we must look at the difference. This is due to how computers save numbers in memory. Ie. To compare value X with Y, use D=abs(X-Y), and if D<0.0001 they are considered equal, otherwise they are different.

Once the server has responded, it can close the connection and return to wait for the next client. The client should read the response from the server and present it to the user.

Below you find two examples,   using *nc* as a client.   To highlight the newline characters, which are easy to print but difficult to see, I've added placeholders '{\n}' to show them.

```
pal@pal-ROG-Strix-GA35DX-G35DX:~$ nc 13.53.76.30 5000
TEXT TCP 1.0{\n}
{\n}
OK{\n}
fadd 39.112001 66.199031{\n}
32{\n}
ERROR{\n}

pal@pal-ROG-Strix-GA35DX-G35DX:~$
pal@pal-ROG-Strix-GA35DX-G35DX:~$ nc 13.53.76.30 5000
TEXT TCP 1.0{\n}
{\n}
OK{\n}
mul 9 26{\n}
234{\n}
OK{\n}

pal@pal-ROG-Strix-GA35DX-G35DX:~$
```

Furthermore; your client should accept a string as input, the string should be in the <IP>:<PORT> syntax. For example;

```
pal@pal-ROG-Strix-GA35DX-G35DX:~$ ./client 127.0.0.1:5000
Host 127.0.0.1, and port 5000.
...
pal@pal-ROG-Strix-GA35DX-G35DX:~$./client doesnotwork.com:5200
Host doesnotwork.com, and port 5200.
...
```

Execution

Start your work from the provided GIT repo; [https://github.com/patrikarlos/np_assignment1 (Links to an external site.)](https://github.com/patrikarlos/np_assignment1).

```
pal@pal-ROG-Strix-GA35DX-G35DX:~$ cd tmp
pal@pal-ROG-Strix-GA35DX-G35DX:~/tmp$ git clone https://github.
com/patrikarlos/np_assignment1
Cloning into 'np_assignment1'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 11 (delta 1), reused 8 (delta 1), pack-reused 0
Unpacking objects: 100% (11/11), 4.44 KiB | 4.44 MiB/s, done.
pal@pal-ROG-Strix-GA35DX-G35DX:~/tmp$
```

This contains starting files for a client (clientmain.cpp), a test (main.cpp), a server (servermain.cpp, not used in this assignment), a Makefile (instructions for make to build the solution/s), and two supporting files (calcLib.c and calcLib.h).   The calcLib.* files are used to build libcalc, a supporting library for the test and server solution.

NOTE: it is NOT recommended to store ANY but temporary data in the /tmp folder, as ITS IS ERASED during every boot.

It's recommended that you make use of macros to enable/disable printing of debugging information. See the example in the repo.

The output from the client should be minimal when DEBUG is not activated. In that scenario it should print what IP, or DNS,   and port it connects to. The assignment was given by the server, and the result the server gave on the assignment (OK or ERROR).   Below you find a couple of examples, bold indicate information that came from server.

```
pal@pal-ROG-Strix-GA35DX-G35DX:~$ ./client 127.0.0.1:5000
Host 127.0.0.1, and port 5000.
```

```
ASSIGNMENT: add 1 2
OK (myresult=3)
```

Now, a case when DEBUG was activated.

```
pal@pal-ROG-Strix-GA35DX-G35DX:~$ ./client 127.0.0.1:5000
Host 127.0.0.1, and port 5000.
Connected to  127.0.0.1:5000 local 127.0.0.1:37122
ASSIGNMENT: add 1 2
Calculated the result to 3
OK (myresult=3)
```

Submission

Your submission **must** be in the form of a *tar.gz* file, containing your solution in a folder called np_assignment1. Cf,

```
pal@pal-ROG-Strix-GA35DX-G35DX:~$ cd /tmp/
pal@pal-ROG-Strix-GA35DX-G35DX:/tmp$ ls np_assignment1/
calcLib.c  calcLib.o  clientmain.cpp  libcalc.a  main.o    serv
er           servermain.o
calcLib.h  client     clientmain.o    main.cpp   Makefile  serv
ermain.cpp  test
pal@pal-ROG-Strix-GA35DX-G35DX:/tmp$ tar -zcvf pal-assignment1.
tar.gz np_assignment1/
np_assignment1/
...np_assignment1/calcLib.o
pal@pal-ROG-Strix-GA35DX-G35DX:/tmp$ ls -la pal-assignment1.ta
r.gz
-rw-rw-r-- 1 pal pal 34231 jan 18 11:13 pal-assignment1.tar.gz
```

The solution should have a well-utilized git repository, source code, and a makefile that is used to build the solution client. Start from the repository above, then you will have the folder, Makefile, and templates for the solutions. Make sure that you track the progression of the solution with commits to the repository, of course, you should not commit things that did not work, or that was just for testing.   I will not examine your repository handling, but I'll check that YOU made progress. I.e. it should not be a clone of my repository, and a commit 'solved'. Once you have completed and tested the solution, run make clean, then compress the entire folder (Links to an external site.), including the git repository. The compressed file (archive) should be named <bth_id>-assignment1.tar.gz. Then submit that archive here.

Testing

As I'll have several solutions to test, I've automated a large part of the testing. The automation requires that you follow the naming standard, and that 'make' and 'make clean' works. That the solutions are called client and server, nothing else. Both should accept IP:PORT as the only argument.

I'll test both of your solutions against other implementations, hence, protocol compliance is critical.