Lab 2a) Line based TCP server

Introduction

Implement a server for the client implemented in Lab1a.

Before you start implementing, **READ** the entire assignment plus Lab1a, sketch a design, and reflect on it.

Requirements

Same communication protocol as in Lab 1a.

1. The server should accept IP:port the first command-line argument.
2. The server should handle ONE client at a time.
3. Five (5) clients should be able to queue for service, the sixth should be rejected.
4. The server should only support protocol "TEXT TCP 1.0".
5. If a client takes longer than 5s to complete any task (send a reply), the server should terminate (send 'ERROR TO\n' to the client), close the connection. Note, at the moment the reference servers do not support this, but it's pending implementation.

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", followed by an extra '\n'.

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

In the next stage, the server generates a random assignment. Use the calcLib library, and the example in the git repo to simplify how to do this. This information should be placed in a string that will be sent to the client. 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.

For floating-point use "%8.8g "for the formatting, i.e.

```
/* Declarations, on top */
double fv1,fv2,fresult;
```

```
int iv1,iv2,iresult;
char msg[1450];

/* within request loop */
char *op=randomType();

/* clear string for sending */
memset(msg, sizeof(msg),0);

if(op[0]='f'){ /* We got a floating op  */
  sprintf(msg, "%s %8.8g %8.8g\n",op,fv1,fv2);
} else {
  sprintf(msg, "%s %d %d\n",op,iv1,iv2);
}
```

Once the client has read the string, it will calculate the result and send the response back to the server, in the corresponding format, integer, or floating-point. Note that div would normally result in a floating-point value, however here it's truncated to an int, i.e.

```
iresult=iv1/iv2;
```

Below you find some examples, '\n' added to highlight the new line. Black indicates strings from the server, orange from the client.

```
add 10 3'\n'
13'\n'
OK'\n'

sub 10 3'\n'
7'\n'
OK'\n'

div 10 3'\n'
3'\n'
OK'\n'

fadd 10.1 3.1'\n'
13.2'\n'
OK'\n'

fadd 0.27849822 0.54688152'\n'
0.82537974'\n'
OK'\n'
```

```
fdiv 0.27849822 0.54688152'\n'
0.50924782'\n'
OK'\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=|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 server may start service before completing the current client.

Execution

To get started with the assignment, and minimize the challenge, start from the same repository as in Lab1a (https://github.com/patrikarlos/np_assignment1 (Links to an external site.)). You can continue using the one from Lab1a, or start anew.

There should not be much output from the server during its execution, only print information if something failed. Note, that if there is a problem with a client, that should not affect the overall server operation.

It's recommended that you make use of macros to enable/disable printing of debugging information.

Submission

Your submission **must** be in the form of a *tar.gz* file, containing your solution in a folder called np_assignment1. See Lab1a for example.

The solution should have a well-utilized git repository, source code, and a makefile that is used to build the solution server. 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.