ECE 358, Spring 2014 — (Lab) Assignment 2
Due Fri, May 23, 11:59:59 PM

(Follow the submission instructions at the end. Mention both group members in the README.)

1.(50 points) Objective: familiarize yourself with the socket interface.

You are to implement client and server programs that use the socket API. The server first reads `stdin` for group information — group IDs, each followed by a list of student IDs and names that belong to that group, with the input terminated by `EOF`. The server is able to deliver a student name given its group number and its id within the group. A client queries the server for a student's name, and prints the returned information to console using `printf()`.

Programming language

You must use either C or C++ to realize this assignment. So no Java, Python, Jython, etc. Your code must compile with the GNU C/C++ compiler on `ecelinux`. We will evaluate your code on `ecelinux`. You're not allowed to copy-n-paste any helper code from the Internet, your friends, your parents, etc. You should build everything "from scratch" on your own. (But you're certainly welcome to consult code that is available online. You should credit such sources via comments in your source-code.)

The Server

The server program, when first run, picks an open port on which it can run. It prints out its publicly reachable IP address or domain name and this port number (with no embellishment whatsoever — the IP address/domain name and the port number only) to `stdout`. E.g., if your server runs on ecelinux2, the printout might look like: ecelinux2.uwaterloo.ca 10093. It then reads from `stdin` information on groups. It keeps reading till it sees an `EOF`. See the sample server run below.

Then, the server accepts client commands. The server is to process the following commands:

1. `GET <GROUP_ID> <STUDENT_ID>`: returns the name of the student with id `STUDENT_ID` in the group numbered `GROUP_ID`.

   In case of an invalid group or student id, the text `ERROR_<GROUP_ID>_<STUDENT_ID>` where `<GROUP_ID>` is replaced by the group number and `<STUDENT_ID>` by the id of the student within the group, is sent back to the client.

2. `STOP_SESSION`: signals the server that the client will stop sending data.
3. `STOP`: tells the server to terminate; i.e., the server process dies. Termination must be graceful.

Keep in mind that more than one client may attempt to communicate with the server at any given instant. You must handle this somehow, and your server should not misbehave if this happens.

Also, a `STOP` implies a `STOP_SESSION`. After a client sends a `STOP_SESSION`, it is unable to communicate any further with the server.

The Client

The client program sends `GET` commands to the server to retrieve students names based on their group number and id within the group. It reads which group and student ID it should perform a `GET` for by reading from `stdin`. It prints non-error responses it receives from the server to `stdout`. It prints error responses it receives to `stderr`.

If the client receives an `EOF` on `stdin`, it sends a `STOP_SESSION` command to the server, and terminates itself. If the client reads the special string STOP on `stdin`, it sends the `STOP` command to the server, and terminates itself. If one (any) client sends a `STOP` command, this causes the server it is communicating with, to terminate gracefully.

The client is provided the following two command-line arguments, in order. (1) the IP address or domain name of the server. Your client should be able to handle both. (2) The port number on which the server runs.

Transport Layer Protocol

You must provide two implementations of your socket-communication. One using the stream-oriented protocol `TCP` (stream socket) and the other, the connectionless protocol `UDP` (datagram socket). So, you will have one client implementation each for TCP and UDP, and one server implementation each for TCP and UDP.

Makefile

Your submission must include a makefile with at least the following targets: udp, tcp and clean. The targets udp and tcp should build two executables: `client` and `server`, for the appropriate client/server pair. The target clean should remove all object, executable and temporary files. That is, the directory should be left with source code only.

Sample server run

```
% ./server
129.97.56.13 5672
Group 1
1 Alice
2 Isambard Kingdom Brunel
Group 223
14 Bob Barker
Group 215
455 Charlie Horse
^D
```

Note that “129.97.56.13 5672” is printed to `stdout` as soon as the server is launched, as the IP address and port number on which the server runs. The remainder is the group specification, terminated by an `EOF`. The server then continues to run quietly servicing clients, till it is terminated with a `STOP` command from a client. When it receives a `STOP`, it quits quietly.

Sample client run

```
% ./client ecelinux3.uwaterloo.ca 5675
1 1
Alice
```

```
223 14
Bob Barker
215 455
Charlie Horse
23 11
error: 23 11
thisiscrazy
error: invalid input
^D
```

Note that the two error messages are output to `stderr`, and the non-error messages are output to `stdout`.

Coding Standard

Your code must follow a programming style that you can choose freely; but you have to explicitly specify which one in comments in your code towards the start. It is important that the code is easy to read, with meaningful variables names. Example coding conventions are at `https://code.google.com/p/google-styleguide/`.

Submission Instructions

- Your submission is made to the appropriate dropbox on Learn.
- It must be a single zip file. You can use the "`-r`" option to `zip` to recursively zip a folder.
- The name of your submission must be: `ece358a2.zip`. Our script will look for this, and if it does not find it for you, you get an automatic 0 (see marking scheme below).
- Unzipping your `ece358a2.zip` must result in a single folder called `ece358a2`, that has no sub-folders. All your source-code, Makefile and README must be in that single folder.
- Include a README file with the names of your two group-members.

0.1 Evaluation

Our grading scheme:

a) Marking script breaks, e.g., because submission not structured as per instructions, makefile does not work, program crashes, etc. — automatic 0.

b) 5% if you implemented something meaningful that compiles and runs when given no input.

c) 30% if you implemented something meaningful that compiles and runs, and accepts our inputs as specified (but does not necessarily produce the correct outputs).

d) 90% if your program runs and meets all the functional requirements based on our tests.

e) 100% if, in addition to meeting the functional requirements, your code is pretty and well-documented.