CS454/654 Assignment 2 (Milestone Assignment)

Due: 17 February 2016 (4:00pm)
Returned: 26 February 2016
Appeal deadline: 4 March 2016

1 Milestone

To help get you started on Assignment 3 early and to help familiarize you with the socket primitives you will need to use, you are required to implement a milestone that consists of a simple server capable of handling requests from multiple clients, where clients can join or leave at any time. This server only implements a simple service, converting strings to title-case, so we will not use the procedure database or a (complex) protocol. This milestone is to be done individually.

A simple implementation of title-case is expected for this milestone. Server will convert a string to title-case by capitalizing the first character of every word and converting the rest of the word to lower case. Words are separated only by white space; all other characters count as part of a word.

```
String: document specification FOR CS454 a2 milestone
Output: Document Specification For Cs454 A2 Milestone
```

1.1 Client

The client simply connects to the server and forward strings that are to be converted to title-case. In response, the client expects another string with the same length. More specifically, the client should gather strings as input from the user and use them as requests to the server. Note the client should accept strings from the user once it is run rather than requiring them to be passed as command line arguments. You should also insert a 2 second delay between successive requests by using the `sleep()` procedure. The delay should be on sending the requests to the server. The client should be able to receive and buffer new strings from the user during this delay. Following is the correct behavior of the program:

```
<blank line> <enter string> <Press Enter>
<blank line> __
```

No delay after pressing enter and the second input line appearing. This won't happen if you have not employed a scheme to multiplex between sending requests and reading input. It is recommended that you use threads for reading input and sending requests.

The location of the server is passed through environment variables `SERVER_ADDRESS` and `SERVER_PORT`, which are explained in the next section. Use the `getenv()` procedure to obtain these two values.

A request message is composed of `string_length` followed by `text` where `string_length` is a 4-byte value for the length of the string including the trailing `NULL` terminator (that is, the value is `strlen(text) + 1`), and `text` is the characters that make up the string (including the trailing `NULL` terminator).

A reply is composed of `string_length` followed by `text` where `string_length` is a 4-byte value for the length of the string including the trailing `NULL` terminator (that is, the value is `strlen(text) + 1)`, and `text` is the characters that make up the title-case string (including the trailing `NULL` terminator).

Please note here that these message formats are for the communication between the client and server. The user will not enter length of the string, but you should be able to calculate them yourselves.

The output on the client end should be of the following form:

```
test string
Server: Test String
```

Note that the first line indicates the string entered at the client whereas the second line indicates the string returned by the server. The phrase `Server:` is prepended by the client after receiving the title-case string from the server.

On end-of-file (EOF), a client should wait to receive the server's reply to the last request sent by the client to the server.

1.2 Server

The server should accept connection and processing requests from clients. The format of processing requests was given in the previous section. You should print the string received from the client at the server. You should use `select()` to multiplex processing and connection requests from clients. If we run multiple clients simultaneously, we should see that requests are interleaved. Your client should print out the string received in response from the server for each request (pre-pended with the string `Server:`) so we can verify that your server is properly using the `select()` call to multiplex different clients. You can assume an upper-bound of 5 simultaneous clients for a server.

The server process must reside at a known machine and port number so clients can contact it. Since the port will be assigned dynamically (see below), we will use environment variables. When the server starts, it must print two lines:

```
SERVER_ADDRESS <machine name>
SERVER_PORT <port number>
```

where `<machine name>` is the machine name where the server is executing and `<port number>` is the port that the server is listening on. Be sure to use this format for output. We may try to use scripts to automate part of the marking process, and any changes in output may cause our scripts to fail. These environment variables should be set on the command line at the client machines before those processes are started. The client must read both these values from the environment to properly connect to the server which may be running on a different machine than the client.

To prevent different processes from trying to use the same port number for your server processes, which could cause any number of unexpected problems, you should bind to the next available port (zero) and the ip `(INADDR_ANY)`. This causes the socket to be bound to the next available port number, which will prevent port conflicts. You can then obtain the actual port to which the socket has been bound.

For our purposes, if an attempt to accept a request from a client fails, then you may assume the client has quit and close the server end of the connection. This is not considered a fatal error in the server; it should continue to execute, accepting new connections and processing requests from clients. You can assume that the server never terminates and runs forever.

1.3 Development and Submission Details

You must develop this assignment so that it compiles and runs on the machines with the same architecture and operating system as `linux.student.cs.uwaterloo.ca`. Ask the MFCF consultants on the third floor of the Math and Computer Building for an account if you do not already have one. All compilation and testing will take place on these machines. You will receive an automatic 10% penalty if your assignment does not compile.

You should produce a `Makefile` that produces the above executables. If there is additional compilation or execution instructions that we will need to run your assignment, please put this information in a `README` file that should be submitted with your assignment. No documentation is needed for this milestone.

You will hand in your assignment (source code, `README`, and `Makefile`) using the `submit` command.

1.4 Requirements

You are required to implement the simple client-server system as described. You should do your development in C/C++ using TCP/Sockets[1]. The end result of your assignment should be two executable files called `stringClient` and `stringServer` that implement the simple client-server system described above. The following steps will be taken to grade your submission.

1. `make` to compile your client and server. Please specifiy in the `README` file.
2. `./stringServer`
3. Manually set the `SERVER_ADDRESS` and `SERVER_PORT` environment variables on the client machine. (Use `setenv` if using C shell)
4. `./stringClient` (The client may not be running on the same machine as the server)
5. Enter string(s) on client machine

1.4.1 Evaluation

Your grade for this milestone will depend on meeting the following requirements:

1. Submission of README, makefile and source code.
2. Code should compile on `linux.student.cs` environment. Automatic penality of `10%` if this is not the case.
3. Outputs on the client and server:
   - all output should go to STDOUT
   - string and the title-case string (returned from server) on the `client`
   - Machine address and port on the server with the appropriate format
   - strings on the server side as received from clients
   - Nothing else should be printed on the client and the server
4. Server's ability to:

[1] Sockets tutorials will be provided through the course web site.

- correctly convert strings to title-case
- multiplex successfully between clients (up to a maximum of 5)
- dynamically use an available port

5. Client should function correctly when running on a machine different than the server.

1.5 Useful Resources

POSIX threads (pthreads) is a standardized interface for threads on UNIX systems and a great tutorial with code examples can be found here

`https://computing.llnl.gov/tutorials/pthreads/`

A more general tutorial on threads including details of different threading interfaces is given in

`http://www.cs.cf.ac.uk/Dave/C/node29.html`

If you use any public/web-based source code repository hosting service such as GitHub or Bitbucket, you are responsible for ensuring that your code is not publicly available.