CS 460 - Rock/Paper/Scissors Game with Sockets

Purposes:

- Learn about sockets, and specifically about Berkeley sockets
- Learn about TCP connections
- Create your own networking protcol
- Write a program in C, or C++, using sockets and TCP
- Have some fun creating a networked game

Requirements:

Independently design a protocol for, and implement your protocol, for a rock/paper/scissors game to be played by two users from separate computers. The game may also be played by the users logging into the same computer.

The rules of the game are:

- Each player randomly picks one of: rock, scissors, or paper.
- Each player displays their choice so that a change in selection is not possible. This is usually accomplished by displaying the selection at the same time (e.g. on the count of three).
- The winner is determined by the following rules:
  - Rock wins over scissors (a rock can smash scissors)
  - Scissors wins over paper (scissors can cut paper)
  - Paper wins over rock (paper can cover a rock)
  - Identical selections are a draw

Your programs:

- Must use Berkeley sockets to implement a two player version of the game.
- Must use TCP.
- Must compile and run on Linux Lab systems.
- Must be implemented with a server program and a client program that take arguments.
  - To play the game, start the server up on a system, then, start up two copies of your client program for the two players.
  - The server and client programs should be able to run on the same computer, with the server and one client on the same computer, or with the server and two clients all running on different computers.
  - The server code takes one parameter, a port number.
  - The client code takes two parameters, a hostname and a port number (in that order). The hostname is the name of where the server is executing. The "hostname" argument should be able to be specified as a name like AUS213Lxx (where "xx" is a number between 1 and 35) or 157.201.194.2yy (where "yy" is 01 to 35). Example commands to start a client might be rpsClient AUS213L5 4567
    or: rpsClient 157.201.194.210 4567
- Must be able to play more than one game (entering r, s or p) with an opponent
  - Should have a quit command.
  - Just assume the player wants to play another game, **don't ask (prompt) if they do or don't want to play again.**

- Should handle error conditions such as (but not limited to): the wrong number of command-line arguments and the wrong user input.
    - Give a usage statement if the command line has the wrong number of parameters
        - rpsServer #no argument, give a usage statement
        - rpsClient #no arguments, give a usage statement
        - rpsClient AUS213L20 #missing argument, give a usage statement
    - Give an error message if a bad port number or unusable port number is given. The error message should tell what the problem was.
    - Give an error message if the host was unreachable.
    - Give an error message if no RPS server is running at the specified host and port number.
    - Yes, this lab is **asking that you don't do something nice like** use default values or prompt the user for the missing information. [These requirements help with trying to automate some testing of the servers and clients, and, help you see the error messages that are generated for certain failure modes.]
- The interface may be as simple as:
    - Giving a prompt for a player to enter their choice as 'r', 's', or 'p' (and q for quit)
    - Displaying the player's choice that is typed in at the keyboard
    - Displaying the results of the round or a termination message
    - Allow the players to play another round
- You may wish to keep track of, and display, the wins/losses/draw for each player. This is not required.

Your protocol:

- Should handle the requirements listed above for your server and client.
- Is to be documented in the I-Learn discussion board where you upload your server and client code to be reviewed by peer reviewers.
- Your protocol description should allow another student in the class to be able to implement a client to work with your server or implement a server to work with your client.

You may wish to play your game against a lab assistant or other student, where they run your client which communicates to your server, to test your code. If you do, talk to them! Ask them what they entered so that you don't miss something like being told that both of you win, or both of you lose.

Submit a copy of your programs using the **submit** program (**two** command line submit commands, one for the server and one for the client; the following does not work: submit myServer.cpp myClient.cpp). Use the sample headers found in `/home/cs460/labRPS` that start with "rps_..." for the programs to be submitted properly.

The submitted code should include the following documentation:

- Appropriate comments in the code.
- Links or a reference to code/materials you used.
    - Yes, you may use a sample C or C++ server and client from off the Internet to get started with this assignment (see links below). Typically, these clients would echo a message or something along those lines.
    - Don't copy and paste code that implements anything other than a very basic TCP client and server.

Helps:

To use types.h and socket.h use:

```
#include <sys/types.h>
#include <sys/socket.h>
```

You may use threads to implement this program but there is no requirement to do so. Your server may sit and wait for a client to communicate with it, and then sit and wait for the other client to communicate with it. One of the biggest problems students run into with this program is dealing with a TCP connection that is a steam-of-bytes and the fact that TCP can split messages up **or** combine messages. If messages are not of a fixed length, or there are no message termination character(s), programs usually get confused. One option is to implement "buffered reader" type functionality. Another option is to design your protocol so that sends and receives are matched between the clients and server to avoid receiving two messages at a time (usually the simplest option for this assignment). It is also suggested that you don't complicate the assignment with a protocol that could handle all kinds of different features. Concentrate on doing the basic assignment first.

Using telnet to connect to a server may be useful in debugging the server.

If you get a 'bind failed' message when trying to start your server, make sure you don't have a copy of your server still running (use a ps command to find out). You might also need to wait a couple of minutes for the TCP TIME_WAIT period to expire; instead of waiting, you could just use another port number. Properly closing connections will help with this issue.

The names of the systems in the Linux Lab have an L ('ell') in them, with capital letters the names would be: AUS213L2.

There are various guides to network programming on the web. (If you find a really good one, let us know.)

http://beej.us/guide/bgnet/ (This is one I quite like.)

http://www.linuxhowtos.org/C_C++/socket.htm

The book's web site at: http://wps.aw.com/aw_kurose_network_3/, has a link to a "short course" on Unix Programming. Go to "Student Resources" -> Authors' Recommended Links -> Chapter 2, Unix Network Programming.

Unix Networking - Kurose - http://gaia.cs.umass.edu/ntu_socket/

William Stallings has "A detailed programmer's introduction" on sockets - http://williamstallings.com/DCC/DCC7e.html

The following site: http://www.freesoft.org/CIE/index.htm says: "The Internet Encyclopedia is my attempt to take the Internet tradition of open, free protocol specifications, merge it with a 1990s Web presentation, and produce a readable and useful reference to the technical operation of the Internet."

Various manual pages may be of help:

man 7 tcp
man socket
man setsockopt

(If there are entries in multiple sections of the manual pages you have to specify a section number when giving the 'man' command. Try 'man socket.' Then look for socket in sections 2 and 7. (For information about manual pages and what the various sections contain, you might see: http://en.wikipedia.org/wiki/Manual_page_%28Unix%29)

If you run into segmentation faults you most likely need to review pointers and c-strings. You might take a look at the code in /home/jonesro/cs460/pointer_pointers.