# CptS 455 Project #1
# TCP/IP Socket Programming
# **Assigned**: Sept. 21, 2015
# **Due**: Oct 12, 2013, 11:59PM (turn-in using course Project Turn-in page)

September 21, 2015

## 1 Overview

In this project you will develop a client and server that interact over a TCP connection. The specified protocol involves use of several different kinds of serialization of protocol data so that you become familiar with the programming techniques used to deal with each of them.

## 2 Requirement specification

Specifically, the client and server are C programs that need to meet the following requirements:

1. The client program needs to take two command-line arguments that specify the name (or IP address) of the server and the port on which the server is listening. The server program needs to take a command-line argument that specifies the port on which it is to listen.

2. You will start the server first. Once started, it should not quit until exited using a "Ctrl-C" signal (from keypad). When the client starts it will connect to the server.

3. Once the client has connected, the server will send `Welcome` to the client as a new-line-terminated (`\n`) string (the 1st communication of our little application protocol). Note that a null terminating character should not be sent when a newline-terminated string is specified. Remember that the client code must be prepared for the server to say something other than this; if so it should close the connection, print an error message, and exit.

4. After receiving the welcome message from the server, the client will prompt the user to enter a numerical ID number and a name using the keyboard – i.e. from standard input. This ID and the name will then be sent in that order to the server as 2 newline-terminated strings (2nd communication of the application protocol).

```
char buf[SIZE];
// len = length of message
buf[len - 1] = '\n';
...
send( fd, buf, len );
```

5. The server, after receiving the ID number and name from the client, will look up the pair of them in a table and send a validation message `Success` if the pair is found in the table, or send `Failure` otherwise, to the client (3rd communication of the protocol) – whichever one is sent is sent as a newline-terminated string.

6. If failure occurs the server will close the socket. The client should not send to a closed socket, so it must properly detect the occurrence of the `Failure` message.

7. After receiving a `Success` message from the server, the client will prompt the user with a message asking for a password. This password will be sent to the server by first sending the length (of the password) as a 2-byte binary number in *network byte order* (remember to use `htons()` on the sending side and `ntohs()` on the receiving side) followed by the string (not null terminated). Note that this is a different way of sending a string than used previously in steps. And again, no null character is sent at the end of the string. An example follows:

```
uint16_t length = 20;
// convert and send
network_byte_order = htons(length);
send(fd, &network_byte_order, sizeof(uint16_t), 0);
```

8. The server, after receiving the password from the client, will verify that the password matches the previously received name and ID number in the table in your program code. The server will either send the message `Congratulations <name>; you've just revealed the password for <ID number> to the world!` or the message `Password incorrect`. For this message use the encoding of the previous step – a two-byte length in network byte order followed by the string. The server will then close the connection.

9. The client, after receiving the result message, will print out the result, close the socket, and exit.

# 3 Design and implementation aspects

1. Don't do anything fancy for storing ID numbers and names, just make an array of a few ID numbers and names in your code.

2. A null terminating character should *not* be sent when a newline-terminated string is specified.

3. Check the return values of all system calls, print an appropriate message and exit if an error occurs. Also, remember that `send` and `recv` on TCP sockets do not necessarily send and receive as much data as was requested to be sent/received.

4. Make sure that buffers are big enough for the values that may be stored in them.

5. Do not send null terminating characters for the strings you send. Note that as a result of this functions like `strlen` and `strcpy` are not appropriate ways to copy data (as `strcpy` is meant to copy only null-terminated strings, and `strlen` stops counting characters as soon as it finds a null character). Investigate `memcpy()` instead.

6. Do not send more characters over the network than specified in the protocol above. For example, something like

   ```
   char buf[BUFSIZE];
   ...
   send (s, buf, BUFSIZE);
   ```

   is almost certainly not correct.

7. When the server closes its *communication* socket it resumes listening for client connections.

8. To go from a hostname or an IP address expressed in dotted decimal notation as a string to a value suitable for storing in a `struct sockaddr_in` for use in a call to connect you use the library function `getaddrinfo()`. Its use is described in Donahoo and Calvert second edition. If you have the first edition of the book it describes the `gethostbyname()` function which it is also possible to use – or just read the man page for `getaddrinfo()`.

# 4 Grading

Your project will be graded by the TA looking for: correct function; correct use of TCP send/recv; comments explaining what is going on in each major section of code – examples of major sections of code include handling the command line parameters; establishing the listening socket in the server; handling the input

for the initial contact from the client; handling the interaction for the password interaction from the client.

You should also avoid repeating very similar pieces of code – instead move what would be repeated into functions with appropriate parameters. For example, there should be a function that reads a string that is prefixed by a length; there should be a function that reads a newline-terminated string; etc.

Finally, this is a group project; your submission will be graded on its technical content, and your contribution will also be graded. Every member of the group is responsible for knowing the entire submission – why things are done as they are, how the code works, etc. Your group will meet with the TA and be quizzed about these things. This part of the grade is worth 10 points.

You will need to complete a survey at the end of the project in which you rate the contribution of members of your group. Extraordinary contribution will receive up to 4 points additional credit; lack of contribution may receive up to 4 points deduction from the group score.

Your final grade then is:

```
code (0-90 -- group)
plus interview (0-10 -- individual)
plus participation (-4 - +4 -- individual)
```