CS 2200 – SPRING 2016

MICRO PROJECT 4 - NETWORKING

Introduction

In this assignment, you will be creating a TCP/IP client and server. The server will take strings given to it by the client and scramble them using a function that we provide. The client will take a string as a command line argument, send it to the server, receive the scrambled string back, and print the scrambled string to standard output. You have been provided with the following files:

- **server.c** The server source code with the scramble method and some basic code filled in
- **client.c** The boilerplate client source code with some constants defined for your use
- **Makefile** A simple makefile that compiles the client and the server sources

Note: We have already included the headers for all the functions you will need in the source files. More information on your task and the functions you will be using can be found below.

Your Task

1) The Client

You should create a C program that meets the following requirements:

1. The program binary should be named “client”
2. Take two arguments on the command line: the IP address of the server, followed by the string that it will send to the server
3. Use the specified IP address to connect to the server on port number **2200**. The following functions will be useful:
   a. `inet_aton(…)`
   b. `socket(…)`
   c. `connect(…)`
4. Once connected, the client should send the string. Then it should wait to receive the response from the server. The following functions will be useful:
   a. `send(…)`
   b. `recv(…)`
5. After receiving the response from the server, the client should (in either order) close the connection to the server and print the response string to stdout. The following functions will be useful:
   a. `close(…)`
   b. `printf(…)`

Note: We strongly suggest using the Linux man pages to find more information about these functions. If you cannot access a man page for some reason, try Googling the function name followed by man, and you should be able to find the same information. Here is a link that might be useful.

More details on creating a TCP/IP client program using the Sockets interface may be found in the textbook and your class notes.

The command line arguments should work like this:

```
./client 192.168.1.2 "Hello World"
```

where 192.168.1.2 is the IP address of the computer where the server is running, and "Hello World" is the string to be sent to the server. Note that the quotes are not part of the string; this is how you can make command line arguments that have spaces in them (as well as some other characters that normally have special meanings on the command line), which you might find useful. So when we put "Hello World" on the command line, the second argument passed to the process is H-e-l-l-o-(space)-W-o-r-l-d. If we instead gave this command line argument:

```
./client 127.0.0.1 Goodbye World
```

then the second argument (which is the string being sent to the server) would simply be G-o-o-d-b-y-e. W-o-r-l-d would not be part of the second argument; it would be a third, separate argument to the process.

You can also take advantage of a special IP address, 127.0.0.1, which is a special IP address called "localhost" or "loopback" and is reserved to mean "this same machine". So if you run your client and server on the same computer, you can simply use the loopback IP address for your IP address parameter. You may want to try putting them on different machines and using a real IP address at least once, just so you can see your program really working! However, using the loopback IP like this is sufficient for testing your code. (There are reasons this may not be sufficient for testing large, complex, commercial-quality network code, but it should be fine for this small program.)

2) The Server

You should create a C program that meets the following requirements:

1. The program binary should be called "server"
2. The server does not take any arguments on the command line.
3. The server should listen for incoming connections on port 2200 using the INADDR_ANY value for the server's IP address (which basically means it will accept incoming connections on any IP address). The following functions will be useful:
    a. `socket(…)`
    b. `bind(…)`
    c. `listen(…)`
4. Once the socket is listening, you may accept connections. Have your server accept a connection, handle it (details in the next step), and then loop back to accept another connection. You may have the server do this in an infinite loop, always going back to wait for a new connection. (You can terminate the server by sending it a SIGINT signal, i.e., pushing Ctrl-C.) You will find the following function useful:
    a. `accept(…)`
5. For each connection that is accepted, you should receive a string from the client, call the function we have provided to scramble the string, send the resulting scrambled string back to

the client, and then close the connection. In addition to the function we are providing, you will find the following functions useful:

a. `recv(…)`
b. `send(…)`
c. `close(…)`

Again, refer to the man pages for more information on how to use these functions.

More details on creating a TCP/IP server program using the Sockets interface may be found in the textbook and your class notes.

3) The Protocol (sending and receiving strings)

For reasons that are best explained beyond the scope of this document, it is bad practice to call recv() once and assume that the bytes returned are all the data you are looking for. You need to call recv() in a loop, getting more bytes, until you have the entire message (which in this case is simply the string). Therefore you need a way to know when you got the whole message. (Note: when recv() returns 0 is not a good way to determine you got the whole message, again for reasons best explained beyond this document.) In essence, you need to know how many bytes to receive and then call recv() until you got them. There are three basic ways to do this. You *must* use one of these three in your assignment, but which of the three you choose is up to you.

a. Fixed Length

You may simply decide that you will always send exactly a certain number of bytes. When you send the message, you send exactly that many bytes. When you receive, you call recv() in a loop until it has received a total of that many bytes. Of course this means for strings shorter than that length, you need to "pad" the end of the string so that you have enough bytes to send. (Take a moment to think about what a good padding character world be. A wise choice will make your code easier than a poor choice.) This also means that there is a strict upper limit on string lengths, since a single string cannot exceed this length. For this project, an upper limit of 256 characters will be an acceptable choice.

b. Framing

Another option is to send a fixed-length integer value that contains the number of characters in the string, followed by sending that many characters. On receiving, you need to first receive the integer (which is fixed length), then read the integer and receive that many more bytes. This still imposes a length limit, but 2^32 is a pretty large limit and you don't need to waste time sending "padding" characters. However, sending integers has endianness issues, so you need to be sure to use the htonl() and ntohl() functions appropriately on the integer value.

c. Delimiter

The third option is to use a delimiter character. The sender will send all the characters of the string followed by the delimiter character. That delimiter indicates that that is the end of the message. The receiver can keep receiving data until one of the characters received is the delimiter character. This has the advantage of no length limit, but is slightly harder to code. (Take a moment to think about what a good delimiter character world be. A wise choice will make your code easier than a poor choice.)

Testing

To test your code, first build the executables using the `'make'` command. If you want to individually build the client or the server, you may type `'make client'` and `'make server'` instead.

Then on a terminal window, type in:

```
./server
```

This should start your server, which will wait to accept a connection at port number 2200. If you have implemented the server correctly, you will be able to connect the multiple clients to the server without any issues.

Now, open another terminal window and type in:

```
./client <Server IP address> <The string you want to send>
```

For testing on your local machine, you may want to run the following example;

```
./client 127.0.0.1 "Hello World"
```

This should print the response: 'Uryyb Jbeyq' on standard out.

Deliverables

You should create a deliverable tarball, which contains the following:

- client.c (source code for "client")
- server.c (source code for "server")
- any other source files needed to compile your project
- Makefile (that compilers both "client" and "server")

MAKE SURE TO DOWNLOAD A COPY OF YOUR T-SQUARE SUBMISSION AND TEST IT TO ENSURE IT STILL WORKS! WE WILL NOT BE ACCEPTING ANY EXCUSES WHERE THINGS BROKE WHEN SUBMITTING THE PROJECT!