CS 306 – Linux/UNIX Programming – Spring 2015

Lab #4
(Due: 6/11/15)

The subject of this programming assignment is writing a simple *client-server* application that uses (network) *sockets*. The client-server application that you are to write is to allow one to *copy a file* from the (remote) server machine to the local machine. The particular *application protocol* that your client and server are to implement will be referred to as the *remcp application protocol*. It is specified below.

Since this lab involves both a server program and a client program, it is to be implemented in *two separate files*, to be named: (1) `lab4-server.c` and (2) `lab4-client.c`. Each file must be able to be compiled into an executable, which mean each must contain a `main`, etc.

Assume that compilation will produce the following executables:

- server: `remcpd`
- client: `remcp`

remcp Application Protocol:
A key element of client-server systems is the *application protocol* that both the client and the server must adhere to. An application protocol specifies exactly what messages must be exchanged between the client and the server, the order in which they are to be exchanged, and their format. If either the client or the server fails to exactly follow the protocol, they will not function together properly. Not only will your client and server programs be tested with each other, your client will be tested against the server solution, and your server will be tested against the client solution. If you fail to precisely follow the *remcp* protocol, your programs may fail to function with the solution programs. This will mean that your programs are incorrect, and will cause you to lose points. Having clear protocols for applications, and having programs that precisely adhere to those protocols, is critical for client-server applications. How useful do you think it would be to have an http server that didn't strictly adhere to the http protocol, so it worked only with one particular browser?

The *remcp application protocol* specification is as follows:

- Upon establishment of a TCP connection, the client sends: `<remcp>\n`
- The server responds with: `<remcp>\n`
- The client then sends: `<`*shared secret text*`>\n`
- If the secret matches, the server responds by sending: `<ok>\n`
  If it does not match, the server closes the connection.
- The client responds by sending the (argument) *file pathname* followed by a newline
- If the file is readable, the server responds by sending: `<ready>\n`
  If it is not readable, it sends `<error>\n` and closes the connection.
- The client prepares to receive the file and sends: `<send>\n`
- The server now writes the contents of the specified file to the client, and the client reads the file contents and writes it to the local file copy.
- Once the entire file contents has been transmitted by the server, it closes the connection.

In the protocol description, `\n` represents the `'\n'` (newline) `char`. All other characters should be taken literally. E.g., `<ok>\n` represents sending exactly the five `char`'s: `'<'`, `'o'`, `'k'`, `'>'`, `'\n'`.

Remember that system calls do not understand C data types such as strings, and we virtually never send null chars (`'\0'`) over sockets. That means that `<ok>\n` is not to be "turned into a C string" by appending a null char; that will break the protocol and cause your client/server to not function with the solution versions (so it will be wrong!). If you need to turn `<ok>\n` into a string for comparisons, you must do it after it is received.

TCP connections provide data as *byte streams*, meaning you cannot tell where one "message" ends and the next one starts without additional effort I.e., `read()` will give you whatever is in the socket buffer, whether it came from one `write()` or twenty `write()`'s. The *remcp* protocol includes newlines at the end of each message, but it has also been desgined so that only a single mesage at a time should be `read()` (prior to the file being send). You should spend some effort thinking about exchanging the messages through sockets using `read()` and `write()` for this protocol, so you do it as easily as possible (but also transfer the file efficiently).

Client Program:
Your *client program* must a *single command-line argument*. That argument is to be the following two components, separated by a *colon* (`:`):

1. The *"IP address"* of the server machine to connect to, in *dotted decimal/quad* format. (Recall that this is not truly an IP address, since IP addresses are 32-bit binary numbers.)
2. The *pathname of the file to copy*, using a *relative path*, relative to the users home directory.

An example call would be:

```
remcp 131.230.133.20:cs306/lab3.c
```

What this call is to do is create a local copy of the file `~/cs306/lab3.c` stored on the (remote) machine 131.230.133.20. The copy is to be named `lab3.c` and be created in the CWD of the running client program.

When your client program is executed, it should:

1. decode the command-line argument to get an IP address and file path
2. initiate a TCP connection to the specified server machine at the protocol port
3. start the *remcp application protocol*, sending the specified text, secret, and *pathname*
4. if the file cannot be read on the server, print an error message that file could not be copied and terminate
5. if file can be read, create a file by that name in the CWD
6. then enter a loop, reading all data sent through the socket and writing it to the new file
7. once the connection is closed, the program should terminate (thus closing the file)

Server Program:
The server program is not to take any command line arguments.

When run, it should:

1. create and set up a listening *TCP socket*;
2. go into an infinite loop accepting connections from clients;
3. for each new connection, follow the *remcp application protocol* to get a *file pathname*
4. if the file cannot be read, send the appropriate error message and close the connection
5. otherwise, open the file and loop, reading blocks from the file and writing the data to the client via the socket
6. close the connection to the client
7. continue accepting connections

Since the the server function for each client is fairly quick, you need write a server that is only able to handle a single client connection at a time—what is called a *sequential server*. (When a client may stay connected for extended periods, it is better to have a *parallel server*, which can handle multiple clients at the same time.)

Shared Secret:
Because the server process will be running as you, it is important to provide some *security*. Otherwise, anyone could potentially connect to your server and copy/read your files. In order to provide some basic security, your client and sever will use a *shared secret*. The client will send the secret to the server in order to authenticate itself and be allowed to connect. The shared secret can be any text string (of any length). See below for the shared secret string that your submitted code must use—but use a different secret during your development and testing of course!

Additional requirements and information:

- You must use Linux/UNIX system calls *for all socket I/O*, but you can use C library I/O functions *for terminal I/O such as error messages*.
- You will need to include header files such as `<sys/socket.h>` and `<arpa/inet.h>`. Read the man pages for the socket-related calls to determine what is required.
- As the *port* for your service, use 3060 (a high, normally unused port number). Use `#define` syntax to create a symbol `PORT` to define this port number in both programs. Note that when testing this on the CS Dept. workstations you may have to temporarily use a different port to avoid conflicts with other student's programs. Be sure to set it back to 3060 before submission.
- Be sure to write portable code by using the `htons()` and/or `htonl()` functions to convert ports into *network byte order*.
- Remember that a dotted quad input address (as a string) must be converted to a true 32-bit IP address before it can be used in a socket address. You should use the function `inet_aton()` (`inet_addr()` is the older, but now deprecated function to do this).

- To allow the server to easily be run on different machines, use the "address" `INADDR_ANY` to specify that the server should listen on all local interfaces.
- Your *shared secret* is effectively a kind of password/passphrase, which must be common to both the server and client. To make it easy to change, you are to use `#define` syntax in both the client and server to create a symbol `SECRET` to hold your secret word/phrase. To make is easy to test your programs, set your secrets to "`CS30615spr`" before submitting your programs.
- For your initial testing, you can run the server and client(s) on the same machine, using *loopback IP address* `127.0.0.1`. It would also be a good idea to try testing using separate client and server machines if possible. If you do not have two machines of your own, you can use machines in the CS Dept. Lab. Be aware that the SIUC firewalls will prevent you from accessing a server running on the CS workstations from outside the SIUC network (and perhaps from outside the CS Dept. network). Even on your own machines, firewall settings may have to be modified to allow you to access your server from another machine.
- Your program must error check *all appropriate* system and library calls. In case of errors, it must print appropriate error messages to standard error.
- Any errors in the client should cause the client program to terminate.
- Errors setting up the socket should cause the server to terminate with an appropriate status. However, errors while handling a connection should simply cause the connection to be closed. The server should continue running until it is terminated (via *control-c* or `kill`).
- Network programs must be written so that they can deal with the client or server or network "crashing" at any point in the exchange. You do not ever want either your client or server to end up "hung" as a result of a dropped (closed) connection!
- While the messages that make up most of the *remcp* protocol are short, the server must eventually be able to read and store a complete file path. How big does this require your buffer be? It turns out that is a difficult question. There is a variable `PATH_MAX` in the header `<linux/limits.h>` that is supposed to help with that. However, paths can actually be longer than that limit. Nonetheless, use that variable to determine the size of the server's buffer for reading messages from the client. Don't forget that the number does not include a possible `'\n'` and `'\0'`.

You are to submit your two *C source file*, named `lab4-server.c` and `lab4-client.c`, using the Lab #4 Dropbox on the course D2L website. Do not submit any additional files! In particular, do not submit "project" files created by an IDE you may be using. Make sure the files have the correct names!

If you have worked in a team, please note your team members in a comment at the top of your source code file.

If you create/edit your file on a Windows machine, you must make sure that it is in proper Linux/UNIX format when you submit it or you will lose points. So no carriage-returns!

CS 491 students:
You are to implement a *parallel server*, which can handle multiple clients at the same time. The basic idea is to have all the code be in a function, e.g., `handle_client()`. Now, when a connection is to be processed, your server first calls `fork()` to create a subprocess for the connection, running `handle_client()` in the process. The parent can continue in the main server loop, accepting another connection. Don't forget to have the server eventually *collect* any children is creates!

Development hints:
Students are one again strongly urged to *develop their program in stages!* This is even more important since you are dealing with both a client and server. One possible sequence of stages is:

1. Implement a basic server and client pair, which merely connect and then terminate the connection: The client must take the IP address argument, convert it to a true IP address and call `connect()`, and see if the connection succeeded. The server will simply immediately close connection.
2. Now extend the server and the client to exchange the intial *remcp* messages, perhaps having them print out the received messages for now.
3. Extend the server and client so they send the correct protocol messages up through the filename, but don't deal with file contents.
4. Extend the client to create the new/copy file, but have it just write a single fixed line into it.
5. Now extend the server so it writes the file contents to the client. To start, you can just have the client read one block and write that to the file.
6. Finally, extend the client so it reads the entire file contents from the server (until the connection closes) and writes it to the new/copy file.