ENCE360 Lab 4: Sockets

1. Objectives

This lab focuses on sockets, and is built around creating an 'echo server' where a client connects (using TCP) to an echo server and can send messages, to which the server will respond with the same message.

There are two files, **client.c** and **server.c**, as well as pre-compiled binaries **client**, and **server**. Your task is to implement the missing parts of both, note that you should test each part thoroughly before moving to the next.

Before starting on any code, first familiarize yourself with what is expected by running the pre-compiled binaries. Start three terminals, on one – run the server using:

./server_demo

on the other two, run a copy of the client

./client_demo localhost

*W*hat happens when you send messages from either client to the server? (Type into the prompt and press enter)

There is a Makefile provided for convenience with compiling.

2. Preparation

Download the “Lab 4 sockets” zip archive from Learn.

3. The client

Your task here is to complete the socket connection code for the routine *int client_socket(char *hostname)* things you will need to do include:

1. Initialise **TCP** a socket using 'socket'
2. Setup the parameters of *their_addrinfo* and use *getaddrinfo* in order to lookup the IP address from the host name.
3. Finally, connect to the server using *connect*

The client can be compiled using *gcc client.c -o client -std=gnu99* ***-lreadline*** Test this out using the pre-compiled server as before.

4. Accepting connections

Your task here is to implement *accept_connection* using the *accept* function, test that this works properly by adding in a print statement in the server when a connection is accepted. Test this out using the client/server as shown earlier.

The server can be compiled like so:
gcc server.c -o server -std=gnu99

5. Handling requests

Your task here is to implement *handle_request* function according to the following algorithm:

1. read message from socket (use recv or read)
2. write message back to socket (use send or write)
3. repeat 1-2 while data is succesfully read

Ensure all resources/sockets are closed at the end of *handle_request* and all memory allocated (if any) is free'd.

As before, test that this is working correctly before moving on. What happens when you try to connect with two clients at once?

6. Forking

In order to fix the problems identified with handling two connections at once, we're going to run *handle_request* in a **fork()** - so, modify the routine *handle_fork* in order that every time we accept a connection – instead of directly handling it in the main process, instead – we **fork()** a child process to handle that one connection and continue to wait for more connections in the main process.

Note: re-use the *handle_request* function to do the actual communication in the child.

7. Optional chat server

If you've successfully completed the rest of the lab, modify the server (and possibly the client!) to turn it into a chat server. You'll need to do several things to the server:

1. Use threads instead of a fork
2. Store a (global) list of current connections (and protect it with a lock)
3. Modify handle_request to send messages to **all** currently connected connections

Once you've done this try it out with the client. What is the problem with the client? If you've done all of this successfully, modify the client to read messages on a separate thread, too.