Project 3: Network Programming

Overview

In this project, you will implement a TCP client and server that does three-way handshake, a well-known protocol sequence used in communication networks. This project has three parts:

1. *The client and single-threaded server*. In this part, you should implement the client and the server, and your server is implemented as a single-threaded program. The client can be reusable for part 2 and 3.
2. *Multi-threaded server*. In this part, you should improve your server implementation by making it concurrent with multithreading.
3. *Event-driven server*. In this part, you should implement a concurrent server with event-driven techniques.

Project 3a: The Client and Single-threaded Server

You will implement two files in this part: the client (**tcpclient.c**) and the server (**tcpserver.c**). The client takes three arguments as input: the server address, server port number, and an initial sequence number. The server takes in a server port argument, and should be started and be listening on the server port of your choice.

You will implement a three-way handshake communication using TCP as follows:

- **Step 1:** The client sends **"`HELLO X`"** to the server, where `X` is an initial sequence number.
- **Step 2:** The server receives and prints the **"`HELLO X`"** message from the client, and sends "**`HELLO Y`**" to the client, where `Y` is `X+1`.
- **Step 3:** The client receives **"`HELLO Y`"** message from the server, prints the message to screen, and does either one of the following:
  - o If Y = X+1, the client sends **"`HELLO Z`"** to the server, where Z=Y+1, and closes the connection.
  - o If Y != X+1, the client prints an **"`ERROR`"** message, and closes the connection.
- **Step 4:** If the server receives the **"`HELLO Z`"** message from the client, and prints the message to screen. In addition, if Z != Y+1, prints an **"`ERROR`".** The server then closes the connection.

For simplicity, you can assume that the client and the server do not crash during the communication. You must also ensure that the format of all messages (e.g. **"`HELLO X`"**) is correct.

Compile: You should use gcc command to compile both files (`gcc -o tcp$ tcp$.c`, replace `$` with `client` or `server` to compile respective program).

Your implementation **has to work** in the Vagrant VM that you used in the previous assignments. You can run multiple clients and the server on the same machine. The server listens on its port, and when a client connects, a temporary port is assigned to the client-server communication.

Generally, the server and a client in your implementation should work on any two hosts in the network. That being said, since you are running all within one virtual machine, you can assume the server is listening on a port of your choice, and the clients and server run on the same machine with same IP addresses. You can use a local loopback address 127.0.0.1 as the default IP address for all clients and the server.

Project 3b: Multi-threaded Server

In this part, you will enhance your TCP server in part 3a (**tcpserver.c**), such that the new server (**multi-tcpserver.c**) can handle multiple concurrent client requests using *multi-threading*. In the server, when a client connection is accepted (via the `accept` API), a separate thread should be created to handle steps 1-4 above using the `socket` descriptor returned by the `accept` call. We have provided a sample C program for multi-threading (**sample_thread.c**) in module in which you learnt threads..

Compile: You should again use gcc to compile multi-tcpserver.c (`gcc -o multi-tcpserver multi-tcpserver.c -l pthread`). This will compile using the pthread library and create the `multi-tcpserver` executable.

To make several concurrent client requests, use the script **tcp.sh** that we provide. To run the script, type '`./tcp.sh <server_hostname> <port> <clients>`', where `server_hostname` is the hostname where your server is executing (you can set this to *127.0.0.1*) and `port` is the port number that the server is listening to, and `clients` is the number of client requests. For example, `./tcp.sh 127.0.0.1 1234 10` creates ten clients that connect to the server listening on port 1234.

Project 3c: Event-driven Server

In part b, you had implemented a multi-threaded TCP server that supports 3-way handshake. In this part, you are required to write a single-threaded server (**async-tcpserver.c**) that would monitor multiple sockets for new connections using an event-driven approach, and perform the same 3-way handshake with many concurrent clients. The functionality of this server should be the same as that in part b.

Hint: Instead of using threads, you will use the select() function to monitor multiple sockets and an array to maintain state information for different clients. Read Beej's guide to network programming (http://beej.us/guide/bgnet/) on how to use the select() function. We will also provide some sample code on select() for you.

Your new async-tcpserver.c implementation should still take the same command line arguments as tcpserver.c. Also, tcp.sh should work as before to start multiple clients to test the concurrency of the server.

Here are some general hints/guidelines for your server implementation:

- The server should handle two different events for *each client*: Step 2 (three-way handshake first message "HELLO X") and Step 4 (three-way handshake second message "HELLO Z").
  - You should place the code for event handling logic into two functions named `handle_first_shake` and `handle_second_shake`. There will be **static analysis** in our autograder that checks for these event handler functions.
- You can assume that there will be no more than 100 concurrent client connections, and hence maintain an array of size 100 on the server side. Each entry (implemented as `struct`) in the array can be uniquely identified based on file descriptor (FD) and contains necessary information to finish the 3-way handshake, for example, X in Step 1. We call this array of structs the *client state array.*

- Run a select() lookup on the server. Initially, there should only be one FD of interest in the read set, which corresponds to the file descriptor of the socket in which connections are to be accepted.
- Whenever a message is received from a client, use the FD_ISSET to figure out which file descriptor (currentFD) corresponds to the socket in which the incoming message has arrived. You should consider three cases:
    - **Incoming connection establishment**. If currentFD corresponds to the socket where connections are to be accepted, your code should (1) accept the new connection, (2) remember the new file descriptor (newFD), corresponding to the connection established, in your *client state array*, (3) make newFD non-blocking using fcntl(), and add it to your file descriptor read set for select() using the FD_SET.
    - **Three-way handshake first message.** If currentFD corresponds to one of the sockets of established connections, check the *client state array* (a simple loop through the entire array is okay) to see if the X value for this client is assigned. If not, this must be the first "HELLO X" message. You should remember this X value in the entry of this client in the *client state array*. Complete step 2 of the handshake protocol stated in project 3a (the first event for a client). Make sure you call `handle_first_shake` function to handle this event. Our code analyzer will check that `handle_first_shake` calls the `recv` and `send` socket APIs.
    - **Three-way handshake second message.** If currentFD corresponds to one of the connections established, and the X value for this client is assigned in the *client state array*, then this must be the "HELLO Z" message. You should make sure that "Z" in this message is 2 larger than the remembered X value. Complete step 4 of the handshake protocol stated in project 3a. At this point, you can remove this entry from the array (the second event for a client). Make sure you call `handle_second_shake` function to handle this event. Our code analyzer will check that `handle_second_shake` calls the `recv` socket API.

Side Note: Static Analysis

Static analysis is one of the many stages that your code goes through at compile time. In our project 3c autograder, we implemented a static analyzer program that gets invoked when we compile your code. This allows us to inspect both the content and the structure of your implementation in project 3c. To ensure that you have implemented an event-driven code, we check:

- whether or not your code uses a `pselect` or `select` system call.
- whether or not you implemented a `handle_first_shake` function (which should contain `recv` and `send`) and a `handle_second_shake` function (which should contain `recv`).
- whether or not you invoked `handle_first_shake` and `handle_second_shake` based on *client state*.

If you are curious to learn more about how static analysis works, you should take CIS-547 as an elective.

Submission instructions for socket programming

Part a, b, and c have different deadlines, and each one takes a week to complete. You should complete them in the order of a, b, followed by c.

- In part a, create a folder called *project3a*, and put your code as **tcpclient.c** and **tcpserver.c**.
- In part b, create a folder called *project3b*, and put your code as **tcpclient.c** and **multi-tcpserver.c**.
- In part c, create a folder called *project3c*, and put your code as **tcpclient.c** and **async-tcpserver.c**.

Again, note that across all three parts, your TCP client should remain unchanged and you can duplicate the tcpclient.c across all submissions. In all submissions, **you need to include a Makefile** that compiles all the .c files. Your Makefile should generate the binary `tcpclient` (parts a,b, and c), `tcpserver` (part a), `multi-tcpserver` (part b), and `async-tcpserver` (part c). You can modify and reuse Makefiles from earlier projects.

For each part, commit and push your code to the correct directory for your git repository. Create tar balls **project3a.tar.gz**, **project3b.tar.gz** and **project3c.tar.gz** and upload on Coursera for auto-grading. **Note that in part b and c, even if you pass our auto-grader, significant points WILL be deducted if you did not use multi-threading (for part b) and event-driven (select(), for part c) correctly. Our part c autograder in fact will give a score of 0 if the code structure does not conform to our static analysis.**