Project 3: Network Programming

Overview

In this project, you will implement a Transmission Control Protocol (TCP) client and server that does a 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 will be reused for part B and C.
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 single-threaded but concurrent server using event-driven techniques.

Advice

In this project, you will learn and implement synchronous (3a, 3b) and asynchronous (3c) communication between the server and clients. Please make good use of TA hours, Open Office Hours, and Recitations. They are excellent supplements to the lectures and have many practical system programming tips to improve your project. It is highly recommended that you read the homework first, participate in our recitations and start early. There will be 2 recitations: the first will focus on part A and part B, and the second will focus on part C.

Please be aware that the example code provided for each part of this project only provides examples of certain features, and is NOT a starter template for each part of the project. Features demonstrated by the example code can be seen below -

Project example code	**Features demonstrated in the example code**
Part A	Socket communication
Part B	Use of pthread library
Part C	Asynchronous I/O

Based on our past experience, asynchronous communication can be unintuitive and students need to have a different mindset to organize communication logic. This will be the main focus of the second recitation.

Moreover, you are highly encouraged to utilize Piazza to discuss the project with classmates and TAs. Before asking the question, please search in Piazza to see if there are any similar questions asked before. If it is about a bug you are facing, it is likely that it is discussed in the *Common Mistakes* document. Please check all suggestions listed there. If you plan to ask about your code/implementation on Piazza, please consider filling in the following template to make the communication efficient:

- GitHub URL:
- Description of Question:
- Expected Behavior:
- Observed Behavior:

Note that the example code and the *Common Mistakes* document can be found under the resources section in Coursera.

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 3 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 specified server port.

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.

For printing to screen (stdout), make sure that you do "**`fflush(stdout)`**" right after you do the print. You could use printf, fputs, or other ways to print, but **`fflush`** must be used after that. This is because printing in Linux is buffered and may happen asynchronously. If a process does not exit gracefully, your intended prints may be lost. **`fflush`** ensures that your print executes immediately. In our autograder, we terminate the server process using SIGKILL but it evaluates your server output.

Compilation: You should use the `gcc` command to compile tcpclient.c (`gcc -o tcpclient tcpclient.c`) and tcpserver.c (`gcc -o tcpserver tcpserver.c`). Please always make sure that all compile warnings are fixed, as the autograder might fail to compile otherwise.

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 clients and the server all run on the same machine with the same IP. You can use the loopback address 127.0.0.1 as the default IP address for the server and all clients.

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 the course module in which you learnt threads.

Conceptually, 3b is just a refactoring effort from 3a. You would be putting all the client communication code into a thread function, which should be passed into `pthread_create`.

Compilation: 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. Again, you should resolve all warnings before submitting to Coursera.

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 server's concurrency.

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 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 (e.g. 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. 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.

One of the challenges with 3c is that your code needs to handle client interleaving. For example, the server may get the first message from client1, followed by client2 and then client3. However, the second message from client2 might arrive at the server before client1 or client3. The autograder will be creating this interleaving by modifying the TCP client code used. The information below will show you how to set up your client and server code to test this interleaving before you submit to the autograder.

Step 1: Run our concurrent request Python code. A python script has been provided with the sample code. To use this script, start your server, then run:

python3 ./concurrent-requests.py *portNumber*

where *portNumber* is the port used for the server. This script will run your client program 100 times. The script should complete without errors and the server should still be running. If either program crashes, then there is likely an error in your file descriptor management. While this script will highlight major issues, it does not confirm that the server printed out all of the handshakes correctly. You can add counters for each time `handle_first_shake` and `handle_second_shake` are called and print out the result. If both numbers are 100, your code is basically working and then you can test the interleaving.

Step 2: Introduce delays to your TCP client code to test your implementation. To create the interleaving, you need to add code to your TCP client. The example code provided below will result in a random delay (up to 100 msec) before the second message is sent from the client. The details of the code are in the comments. This delay will result in the server receiving the second message in a different client order than the first message. If you manage your file descriptors correctly, then the script should exit without errors, the server should still be running, and the number of calls to first and second handshake should be 100. If you do not get these results, check your file descriptor management. If you have questions about this testing, post on Piazza or attend an office hour.

When you are ready to submit your code to the autograder, remove any extra print statements from the server (such as printing out the number of handshake calls). For the client, you can submit either your original code or the modified code. The autograder will replace it with the test version.

Code to add client delay:

Add the following include statement at the top of your code with the other include statements:

```
// Need to get timestamp to seed random number
#include <sys/time.h>
```

Add the following code just before the client sends the second message to the server

```
// Extra code to create client interleaving
// timestamp is only to seed srand() so that each client sleeps a different
time
struct timeval timestamp;
gettimeofday(&timestamp, NULL); // get current timestamp

// use usec part of timestamp to seed random numbers
// cannot use seconds because clients spawn too quickly to cause a variation in
the time
srand((int)timestamp.tv_usec);

// get a random number between 0 and 99
int ranSleep_ms = (rand() % 100);

// usleep takes number of usec, mult by 1000 to get ms
usleep(1000 * ranSleep_ms);

// Code to send the second message to the server should be here...
```

Submission Instructions for Project 3

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 Gradescope for auto-grading. Refer to project 2 instructions to refresh your memory on how to generate a compressed tar ball.

The Autograder

For each part of this assignment, our autograder runs 3 tests: proxy test, static analysis test, and load test. Passing the first two tests is mandatory: your score would be 0 if you don't pass them.

The proxy test spawns 1 server instance and 1 client instance. Then we run a proxy server (not your code) that relays messages between the client and the server. During this relaying process, we check for:
(1) the existence of message X, Y and Z through socket communication
(2) the correctness of the X, Y, Z values, which reflects the correctness of your handshake mechanism.

The static analysis test runs for part b and part c. It checks for whether or not your code adopts the correct architecture. For part b, we expect that you use pthread to perform multi-threaded handling

Below are the static analysis checks that are performed by the auto-grader:

Project	**Static analysis checks performed by the auto-grader**
3b	Usage of pthread library functions
3c	1. Use of a `pselect` or `select` system call 2. Implementation of a `handle_first_shake` function ○ which should involve `recv` and `send` 3. Implementation of a `handle_second_shake` function ○ which should involve `recv` 4. Whether `handle_first_shake` and `handle_second_shake` is invoked based on the *client state*

Finally, the load test spawns 1 server instance and 20 client instances. We then randomly generate 20 sequence numbers X for the clients to send to the server. We record the print statements (output to stdout) generated by each client and the server -- each client should print out **`HELLO Y`**, and the servers should have 20 lines of **`HELLO X`** (with different X values), and 20 lines of **`HELLO Z`**. Again, you should make sure that these lines match exactly this format, and that **`fflush`** is called right after each print statement. If you are getting 90+ out of 100, chances are that the last few **`fflush`** did not execute before the autograder terminated the server or client instances. Therefore, one solution to this is to put a line of **`fflush`** right before you close the socket on the server side. Another possible solution is to add signal handlers for **`SIGINT`** and **`SIGTERM`**, and put a line of **`fflush`** in these handlers. These measures are usually unnecessary, but if you do encounter this problem in the autograder, these are possible solutions.

Note: Static analysis is one of the many stages that your code goes through at compile time. If you are curious to learn more about how static analysis works, you should take CIS-547 as an elective.

Good Luck!