Project 3
Network Programming

CIT5950 Staff

Contents

Project Overview

In this project, you will implement a simplified 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:

- **Part A**: The client and single-threaded server

  In this part, you should implement the client and the server, with the server implemented as a single-threaded program. The client will be reused for part B and C.

- **Part B**: Multi-threaded server

  In this part, you should improve your server implementation by making it concurrent with multithreading.

- **Part C**: Event-driven server

  In this part, you should implement a single-threaded but concurrent server using event-driven techniques.

Do note that this is an *individual* assignment; you **may not** collaborate with others on this assignment. We encourage you to use Version Control in the provided course repository, but do note that this respository should be and remain *private* indefinitely. Please avoid publishing your project in a personal/public repository to maintain academic integrity.

Key Words for Specifications

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** in this document are to be interpreted as described in RFC 2119.

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. We highly recommended that you read the instructions for each part first, read the manual pages for new system calls, participate in our recitations, and start early. We will hold 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. The following table highlights the features that each example code demonstrates:

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. Please budget extra time for this final project part as it is the most challenging.

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

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

1 Project 3a: The Client and Single-threaded Server

Overview

For Part A, you will implement both a server and client which will communicate over the network by exchanging messages and checking that the messages are in the correct format.

1.1 Requirements

1.1.1 Source files and arguments

- You **MUST** implement two source files in this part: the client (`tcpclient.c`) and the server (`tcpserver.c`).
- You **MUST** provide a makefile that will create the executables `tcpclient` and `tcpserver`. You **SHOULD** include `all` and `clean` rules in your makefile.
- The client **MUST** take three arguments as input: server IP address, server port number, and an initial sequence number, in that order.
- The server **MUST** take a single argument as input: the server port number. The server **MUST** be listening on this port for the entirety of execution.
- The client and server **SHOULD** gracefully exit if a user provides an incorrect number of arguments. You **MAY** print a helpful message regarding the expected syntax.

1.1.2 Handshake Protocol

You **MUST** implement the following three-way handshake communication:

1. The client **MUST** initiate communication and send the message `HELLO X` to the server, where `X` is an initial sequence number (one of the three arguments).
2. The server **MUST** receive and print the `HELLO X` message from the client to standard output. The server **MUST** send a response `HELLO Y` to the same client. The value of `Y` **MUST** be `X + 1`. You **MAY** handle wrap-around (this edge case will not be tested or checked for, however).
3. The client **MUST** receive the `HELLO Y` message from the server and print it to standard output. The client **MUST** check the received value of `Y` and compare it to the expected value of `Y` (namely, `X + 1`).
    - If `Y = X + 1`, the client **MUST** send `HELLO Z` to the server, where `Z = Y + 1`. After sending this message, the client **MUST** close the connection.
    - If `Y != X + 1`, the client **MUST** print an error message to standard error that starts with `ERROR`, then close the connection. The error message **MAY** provide further details on the reason of the error.

4. If the server receives the `HELLO Z` message from the client, the server **MUST** print it to standard output.

    - If `Z != Y + 1`, the server **MUST** print an error message to standard error that starts with `ERROR`. The error message **MAY** provide further details on the reason of the error.

    Regardless of the received message, the server **MUST** close the connection. The server **MUST** continue to `listen` for new connections and `accept` them.

1.1.3 End of Line characters

- The `HELLO` message **MUST NOT** contain the EOL character `'\n'` when transmitted over the network.
- You **MUST** print each received message on a separate line.
- You **MUST** flush the output immediately after writing the `HELLO` message as described in the previous two items. 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. In the autograder, we terminate the server with `SIGKILL`, then evaluate the server output.

1.1.4 Miscellaneous

- You **SHOULD** follow the Coding Guidelines, which is in the Project Resources Module of Canvas.
- You **SHOULD** compile with the `-g` and `-Wall` flags.
- You **SHOULD** handle any warnings produced by the compiler.
- You **MAY** assume that the client and the server do not crash during the communication. That is, you do not need to consider the case where the network fails to send the complete message or drops network packets.
- You **MUST** ensure that your client and server do not crash during execution (e.g. segmentation faults due to string (mis-)management or errors from system calls)
- You **MUST** ensure that the format of all sent messages (e.g. `HELLO X`) is correct.
- You **SHOULD** check the return value for all system calls and report an error (this is to help you debug; we are not checking for any error messages outside of the handshake protocol).
- You **SHOULD** clear all buffers before using them.
- Your implementation **MUST** work in the course Docker container.
- You **SHOULD NOT** make any assumptions about the listening port number on the server.

- 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 (see the Requirements section above), and clients and the server all run on the same machine with the same IP. You **MUST** use the loopback address as the default IP address for the server and all clients for the purposes of grading.
- You **MAY** create a custom server signal handler for the `SIGINT` signal. If you choose to do this, the signal handler **MUST** end by terminating the server.

1.2 Example Output

Here is some example output using `121` as the starting sequence number.

```
//Server
HELLO 121
HELLO 123
```

```
//Client
HELLO 122
```

1.3 Suggested Approach

This is a suggested approach. You are free to implement the solution in any manner you want.

1. Start with the server initially. In the `main` function, check the arguments and setup appropriate variables for them.
2. Create the listening `socket`, `bind` it to the port, and begin `listen`ing on it. You should review the demo code for this module for an example. At this point, your server should start and not crash or exit. Send an interrupt signal to terminate the server.
3. In a loop, continue listening for new connections and `accept` them. `accept` is a blocking system call, so the server will block here until a client connects.
4. Start working on the client. As with the server, check the arguments and setup appropriate variables for them.
5. Create a `socket` and have it `connect` to the server. At this point, you should be able to run the server followed by the client and see the client connect.
6. Begin implementing the handshake protocol. Start by having the client construct the appropriate initial handshake message `HELLO X` and `send` it to the server. On the server side, `recv` the message and print it to the display. Now when you run the server and client, you should see the server print the client's initial message.

7. In the server, setup the response message (`HELLO Y`) and `send` it to the client.
8. In the client, `recv` and print the server's message.
9. The client now needs to check if this message has the expected sequence number. If it is unexpected, print the error message; otherwise, construct the `HELLO Z` message and `send` it to the server. You can test the error handling by (temporarily) hardcoding the server to always send the wrong response.
10. The client is now complete, so `close` the socket and end the program.
11. The server has one final check to do: ensure the `HELLO Z` message has the expected sequence number. Handle both scenarios and `close` the socket to this client. Then return to the top of the loop for a new client.

1.4 Submission

- Your solution **MUST** compile when running `make clean` followed by `make`. Please do check this before submitting. Submissions that fail to compile will receive a 0.
- You **MUST** submit all source and header files, and the makefile, in a tarball (`.tar.gz`) to the Gradescope assignment CIT 5950 Project 3a.
- Your tarball **SHOULD NOT** contain compiled binaries or demo code files.
- You have unlimited submissions until the due date specified by the syllabus.

1.5 Grading

Project 3a will be marked on five rubric items. The descriptions for these tests are in the **The Autograder** section.

There is no peer review for this assignment. You will not be marked on code style.

The score at the due date is final. Do note that they do not sync with Canvas immediately; the course staff must manually release them.

1. **Proxy Test** : 20 points
2. **Load Test** : 20 points
3. **Sequential Clients Test** : 30 points
4. **Client Error Checking Test** : 15 points
5. **Server Error Checking Test** : 15 points

Total : 100 points

2 Project 3b: Multi-threaded Server

Overview

In this part, you will enhance your TCP server in Part A (`tcpserver`), such that the new server (`multi-tcpserver`) can handle multiple concurrent client requests using *multi-threading*. In the server, when a client connection is accepted (via the `accept` API), the server should create a separate thread to handle the Handshake Protocol 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 learned threads.

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

2.1 Requirements

2.1.1 Source files and arguments

All of the requirements from Part A remain the same, with the following changes:

- You **MUST** implement the server in `multi-tcpserver.c` instead of `tcpserver.c`
- The makefile **MUST** create the executable `multi-tcpserver` instead of `tcpserver`
- You **MUST** link the pthread library (you will get a compile error if you forget!).

2.1.2 Handshake Protocol

All of the requirements from Part A apply to Part B.

2.1.3 End of Line characters

All of the requirements from Part A apply to Part B.

2.1.4 Multi-threading

- You **MUST** use multi-threading for this part. The autograder will do a static analysis to ensure you call the appropriate functions. See the **The Autograder** section below for details on this analysis.
- Your multi-thread **MUST** handle interleaving correctly. This means that clients may send messages out of order, e.g. Client A may send `HELLO X` followed by Client B sending `HELLO Y`. See the **Testing Interleaving** section below for how to do this locally.
- Threads should run as detached threads. Calling `pthread_join` can lead to non-concurrent operations and **MUST NOT** be used.

2.1.5 Miscellaneous

- You **SHOULD** reuse the client from Part A. You do not need to make any changes to the client for this part (unless there are still bugs).
- You **MAY** assume that no more than 100 concurrent client connections will attempt to connect to the server.
- All of the requirements from Part A apply to Part B.

2.2 Suggested Approach

This is a suggested approach. You are free to implement the solution in any manner you want.

1. Start by copying your Part A server code into the new source file.
2. If you haven't already done so, refactor your code such that all of the handshake functionality is in a new helper function. What should be left is a small loop that accepts connections and then calls the helper function for the handshakes. Your server should still work as a sequential server at this point.
3. You'll want to create an array of thread IDs, this will allow you to keep track of threads as they are created and destroyed. You only need to do this once.
4. Replace the call to the helper function with a call to `pthread_create`, with the helper function as an argument. Be sure to read the manual pages to ensure you setup the call correctly. Store the thread ID into the thread ID array.
5. Test your multi-threaded server by running a client. You should see all the handshakes print as expected.
6. Further test your multi-threaded server by running several clients back to back, sequentially. You should see each set of clients print the appropriate messages, in order.
7. Stress test your multi-threaded server by running several clients concurrently. The messages may print out of order due to interleaving. See the **Testing Interleaving** section below.

2.3 Submission

- Your solution **MUST** compile when running `make clean` followed by `make`. Please do check this before submitting. Submissions that fail to compile will receive a 0.
- You **MUST** submit all source and header files, and the makefile, in a tarball (`.tar.gz`) to the Gradescope assignment CIT 5950 Project 3b.
- Your tarball **SHOULD NOT** contain compiled binaries or demo code files.
- You have unlimited submissions until the due date specified by the syllabus.

2.4 Grading

Project 3b will be marked on five rubric items. The descriptions for these tests are in the **The Autograder** section.

There is no peer review for this assignment. You will not be marked on code style.

The score at the due date is final. Do note that they do not sync with Canvas immediately; the course staff must manually release them.

1. **Static Analysis** : 0 points
2. **Proxy Test** : 20 points
3. **Load Test** : 30 points
4. **Interleaving Clients Test** : 30 points
5. **Server Error Checking Test** : 20 points

Total : 100 points

3 Project 3c: Event-driven Server

Overview

In Part B, you had implemented a multi-threaded TCP server that supports the 3-way handshake protocol. In this part, you are required to write a single-threaded server (`async-tcpserver`) that monitors multiple sockets for new connections using an event-driven approach, and perform the same 3-way handshake with many concurrent clients.

Instead of using threads, you will use the `select` API to monitor multiple sockets and an array to maintain state information for different clients. Along with the manual pages, 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.

3.1 Requirements

3.1.1 Source files and arguments

All of the requirements from Part A remain the same, with the following changes:

- You **MUST** implement the server in `async-tcpserver.c` instead of `tcpserver.c`
- The makefile **MUST** create the executable `async-tcpserver` instead of `tcpserver`

3.1.2 Handshake Protocol

All of the requirements from Part A apply to Part C.

3.1.3 End of Line characters

All of the requirements from Part A apply to Part C.

3.1.4 Miscellaneous

- You **MAY** use the `epoll` family of functions to handle the asynchronous portion of the assignment instead of `select`.
- You **SHOULD** read the manual pages in detail.
- You **SHOULD** read the manual pages in detail, again.
- The server **MUST** 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`). This requirement refers to the Handshake Protocol above.
- You **MUST** place the code for event handling logic into two functions named `handle_first_shake` and `handle_second_shake`.

- You **MAY** assume that no more than 100 concurrent client connections will attempt to connect to the server.
- You **SHOULD** reuse the client from Part A. You do not need to make any changes to the client for this part (unless there are still bugs).
- All of the requirements from Part A apply to Part C.

3.2 Suggested Approach

This is a suggested approach. You are free to implement the solution in any manner you want.

1. Start by copying your Part A code into the new source file.
2. Instead of a single helper function to do the entire handshake protocol, refactor your code such that each individual handshake is handled by a separate event handler function. Your server should still work as a sequential server at this point.
3. Create two file descriptor sets: one to track *all* of the file descriptors you are interested in, another to track those that are ready to be *read*. Be sure to initalise these correctly (by reading the manual pages).
4. Create some kind of struct to track the state of a single client (e.g. `client_state`). Because the server will handle each individual handshake asynchronously, the `client_state` needs to remember what has already happened in order to perform the correct next step. Be sure to initialise it to a known default state.
5. Since there are potentially multiple clients communicating with the server, you'll need to keep track of multiple clients, possibly in some kind of array (e.g. a `client_state_array`). Setup this data structure and properly initialise it as well.
6. Now for the asynchronous implementation. You'll have to do some prep work before calling `select`:
   - Note the arguments to `select` (from the manual pages). The first argument is the highest file descriptor you are interested in (plus one). Since there are no clients connected right now, the only file descriptor that might have data to read is the `listen`ing socket. Be sure to add that socket to the "all" file descriptor set. Additionally, the highest file descriptor may change during the execution of the program (e.g. when you `accept` a new client), so you'll need to track this value.
   - The second argument is a set of file descriptors that you want to read from. You could use the "all" file descriptor set, but unfortunately `select` modifies the set in-place; `select` essentially filters out file descriptors that aren't ready to be read. This is why you have two file descriptor sets. The "all" set keep a record of all file descriptors, while the "read" set can safely be filtered.

- Clear out whatever happens to be in the "read" set, then copy the "all" set to the "read" set. Now when `select` filters out non-ready file descriptors, the "read" set only has file descriptors that are ready to be read, and the "all" set still has a record of all the file descriptors of interest.
- To make everything properly asynchronous, you need to disable blocking for the listening socket. Use `fcntl` to do this.

7. Call `select` in a loop after all this setup is done. If everything is done correctly, you should be able to see your program exit the `select` call without error. If you check the contents of the "read", you should see that the listening socket is still there. This is easier to do in gdb than with print statements (you should be using gdb anyway).

8. Now you can start handling file descriptors. After `select` returns, scan through all the possible file descriptors and check if it is set in the "read" set. If so, then there is an additional check: is it the `listen`ing socket, or is it a client socket?

9. If it is the `listen`ing socket, `accept` the incomming connection and store the new socket file descriptor into an unused `client_state`. Don't forget to set this socket to non-blocking, add it to the "all" set, and update the highest file descriptor number you have seen so far. Now you can return to the start of the loop and `select` will filter-in this socket.

10. If it is not the `listen`ing socket, then it must be one of the clients. Find the `client_state` for this socket in your client state data structure, and check which handshake to process, calling the appropriate handling function.

11. Be sure to cleanup and reset everything after the second handshake.

12. Test your asynchronous server by running a client. You should see all the handshakes print as expected.

13. Further test your asynchronous server by running several clients back to back, sequentially. You should see each set of clients print the appropriate messages, in order.

14. Stress test your asynchronous server by running several clients concurrently. The messages may print out of order due to interleaving. See the **Testing Interleaving** section below.

3.3 Submission

- Your solution **MUST** compile when running `make clean` followed by `make`. Please do check this before submitting. Submissions that fail to compile will receive a 0.
- You **MUST** submit all source and header files, and the makefile, in a tarball (`.tar.gz`) to the Gradescope assignment CIT 5950 Project 3c.
- Your tarball **SHOULD NOT** contain compiled binaries or demo code files.
- You have unlimited submissions until the due date specified by the syllabus.

3.4 Grading

Project 3c will be marked on five rubric items. The descriptions for these tests are in the **The Autograder** section.

There is no peer review for this assignment. You will not be marked on code style.

The score at the due date is final. Do note that they do not sync with Canvas immediately; the course staff must manually release them.

1. **Static Analysis** : 0 points
2. **Proxy Test** : 20 points
3. **Load Test** : 30 points
4. **Interleaving Clients Test** : 30 points
5. **Server Error Checking Test** : 20 points

Total : 100 points

4 Testing Interleaving

One of the challenges with 3b and 3c is that your code needs to handle client interleaving. For example, the server may get the first message from ClientA, followed by ClientB and then ClientC. However, the second message from ClientB might arrive at the server before ClientA or ClientC. The autograder will create this interleaving by using a custom `tcpclient`. 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.

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 with thread handling (3b) or file descriptor management (3c). 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 handshake and print out the result. If both numbers are 100, your code is basically working and then you can test the interleaving.

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. You can do something similar to add a random delay between the `connect` and first message. 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. The script should exit without errors, the server should still be running, and the number of first and second handshakes should be 100. If you have questions about this testing, post on the course discussion forum or attend an office hour.

   When you are ready to submit your code to the autograder, remove any extra print statements (such as printing out the number of handshake calls).

3. Client Delay Code

   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 used 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...
```

5 The Autograder

For each part of this assignment, the autograder runs several tests: Static Analysis (for 3b and 3c only), Proxy Test, Load Test, Client Error Checking Test (for 3a only), Server Error Checking Test, and Sequential Clients Test (for 3a only) or Interleaving Clients Test (for 3b and 3c only). Passing the static and proxy tests are mandatory: your score will be 0 if you don't pass both of them.

5.1 Static analysis

Value : 0 points

This test checks for whether or not your server code uses the correct architecture. Below are the static analysis checks that are performed by the autograder:

Project	Static analysis checks performed by the auto-grader
3a	1. No check (not performed)
3b	1. Use of `pthread` library functions 2. `recv` and `send` (or `read` and `write`) calls are made, and only made, inside of created threads 3. No call to `pthread_join` (this is unneeded, and may even prevent your implementation from handling interleaving client messages)
3c	1. Use of a `pselect`, `select`, `poll`, or `epoll` call 2. Implementation of a `handle_first_shake` function o which **MUST** include `recv` and `send` (or `read` and `write`) calls 3. Implementation of a `handle_second_shake` function o which **MUST** include a `recv` (or `read`) call 4. Whether `handle_first_shake` and `handle_second_shake` is invoked based on the *client state* 5. No use of `pthread` library functions (i.e. must be single-threaded)

Note: while it is fine to use helper function in additional source code files, the static analyzer will only be checking `multi-tcpserver.c` or `async-tcpserver.c` for 3b and 3c, respectively. Thus, the above functionality **MUST** be included in these files.

By the way: Static analysis is also one of the many stages that code goes through when being compiled. If you are curious to learn more about how static analysis works, you should take CIS 5470 as an elective!

5.2 Proxy Test

Value : 20 points

The proxy test spawns one server instance and one 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 server starting with the correct arguments

(2) the existence of message `X`, `Y` and `Z` through socket communication

(3) the correctness of the `X`, `Y` and `Z` values, which reflects the correctness of your handshake mechanism

5.3 Load Test

Value : 20 points for 3a; 30 points for 3b & 3c

The load test spawns one 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 printed statements generated by each client and the server — each of the 20 clients should print `HELLO Y`, and the server should print 20 lines of `HELLO X` (with different `X` values), and 20 lines of `HELLO Z`. Points are deducted for extraneous print statements in the server.

5.4 Client Error Checking Test

3a only

Value : 15 points

This test checks if the client correctly handles receiving an invalid sequence number in the `HELLO Y` message from the server. In this test, we take the message from the server and change the sequence number that subsequently gets sent to the client. To pass this test, your client code must follow the specifications according to the handshake protocol.

Server Error Checking Test

Value : 15 points for 3a; 20 points for 3b & 3c

This test checks if the server correctly handles receiving an invalid sequence number in the `HELLO Z` message from the client. In this test, we take the message from the client and change the sequence number that subsequently gets sent to the server. To pass this test, your server code must follow the specifications according to the handshake protocol.

5.5 Sequential Clients Test

3a only

Value : 30 points

This test executes handshakes with four clients (A, B, C and D) sequentially. Unlike the load test, these clients are created by a single program that explicitly checks if your server code is sending extraneous data or messages, or is printing extra lines to standard output. To avoid some common errors, make sure:

- You have removed any print statements used for debugging. INCORRECT (#2) or (#8)
- Your server's `send` call does not send more bytes than needed, i.e. provide the correct argument to the `len` parameter. See *man 2 send*. Otherwise may get INCORRECT (#10)
- Your server clears the buffer before calling `recv`. Otherwise you may print part of a previous message and get INCORRECT (#2), (#3), (#8) or (#9). These are detailed in the INCORRECT Message Reasons section.
- Your server does not send a response to `HELLO Z`. Otherwise may get INCORRECT (#10)

5.6 Interleaving Clients Test

3b and 3c only

Value : 30 points

This test is like the Sequential Clients Test but it interleaves the four clients' first and second handshakes, where the "first handshake" means sending `HELLO X` and waiting for `HELLO Y`, and the "second handshake" means sending `HELLO Z`. Additionally, all four clients will connect to the server before any client sends a message.

This is done to ensure that your `multi-tcpserver` and `async-tcpserver` implementations are handling client messages as they are received. In addition to the reminders for 3a listed above, make sure:

- Your server processes clients' first and second handshake messages in the order they are received. For 3c, this means that `async-tcpserver` does not call `recv` on a socket file descriptor unless there is data ready to be read from it. Otherwise you may get INCORRECT (#1), (#4) or (#7).

INCORRECT Message Reasons

#1	No print of first handshake
#2	Unparsable print of first handshake
#3	Wrong sequence number (`X`) in print of first handshake
#4	No reply to first handshake
#5	Unparsable reply to first handshake
#6	Wrong sequence number (`Y`) in reply to first handshake
#7	No print of second handshake
#8	Unparsable print of second handshake
#9	Wrong sequence number (`Z`) in print of second handshake
#10	Reply to second handshake

Good Luck!