CSD2161/CSD2160/CS260/MET3302 2025 Spring *Assignment 2*

Program RTIS/METS Name _____

DigiPen Institute of Technology

Due Date:	23rd February 2025
Topics covered:	Multi-threaded Winsock Programming using TCP sockets
Deliverable:	Moodle submission: submit two C++ files `server.cpp` and `client.cpp` to the Moodle. Naming convention: nil, all students use the same file names.

1 Design Requirements

A multi-threading **echo server** and a multi-threading **echo client** are to be designed in this assignment. The client program should be run on multiple machines to simulate multiple clients. Each client should be able to connect to the server, and request an echo service from the destination client via the server.
The detailed requirements are:

1. The client program can work in one of the following two modes to connect to the server:

   (a) Script Mode: When the server/client program is executed in Windows Command Prompt and a script file is provided, shown in the Figure 1, the server/client will accept the redirect input from the script instead of stdin.
   Please note:
   - *example script files are provided.*
   - *grading is based on Script Mode.*

   ```
   server.exe < server-in.txt
   ```

   Figure 1: Client Program Execution

   (b) Manual Mode: Otherwise the server/client program should wait for the user to type.

2. Server will prompt the user for a port and will then display the server's IP address and the specified port, and start listening on this IP address and port pair. Server will wait for an incoming connection.

3. Client will start and prompt user for an IP address/port number. Client will then have a means to connect to the server.

4. When the user/client is connected to the server, and if the user types a command and presses enter, the client will send a message with a defined format to the server.

   The message format includes a mandatory Command ID filed and several optional fields which are decided by the associated Command ID.
   The Command ID filed is 1 byte long, and indicates different commands, detailed in Listing 1.

```
enum CMDID {
UNKNOWN      = (unsigned char)0x0,
REQ_QUIT     = (unsigned char)0x1,
REQ_ECHO     = (unsigned char)0x2,
RSP_ECHO     = (unsigned char)0x3,
REQ_LISTUSERS = (unsigned char)0x4,
RSP_LISTUSERS = (unsigned char)0x5,
CMD_TEST     = (unsigned char)0x20,
ECHO_ERROR   = (unsigned char)0x30
};
```

Listing 1: Command ID definition

The remaining fields can be in one of the following three variations according to the Command ID in the message.

(a) `REQ_ECHO` and `RSP_ECHO` Command IDs: There are four fields after the Command ID field, illustrated in Figure 3.

i. IP Address: 4 bytes, in network byte order.

A. When the message is sent to the server, the destination client's IP address should be in this field.

B. When the message is sent from the server, the source client's IP address should be in this field.
Figure 2 illustrates the IP and port for different message directions.

ii. Port Number: 2 bytes, in network byte order

A. When the message is sent to the server, this field means the destination client's TCP port number.

B. When the message is sent from the server, this field means the source client's TCP port number.

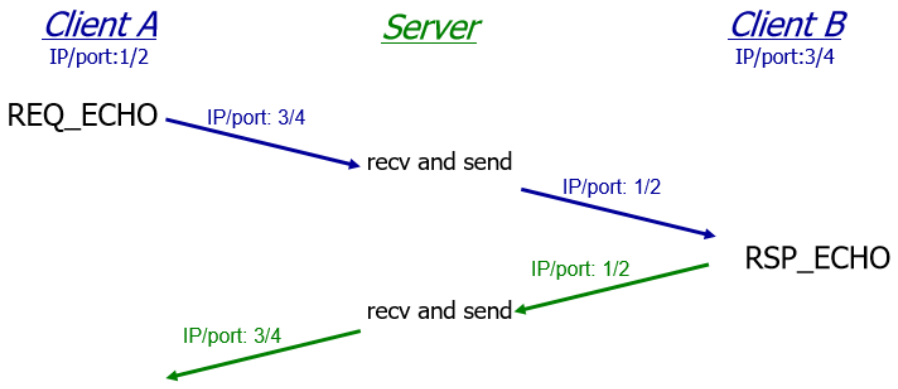

Figure 2: Echo Transaction with IP/Port info

iii. Text Length: 4 bytes, used to indicate the length of the sent text message (i.e., payload) in bytes, and in network byte order.

iv. Text: the echo text.

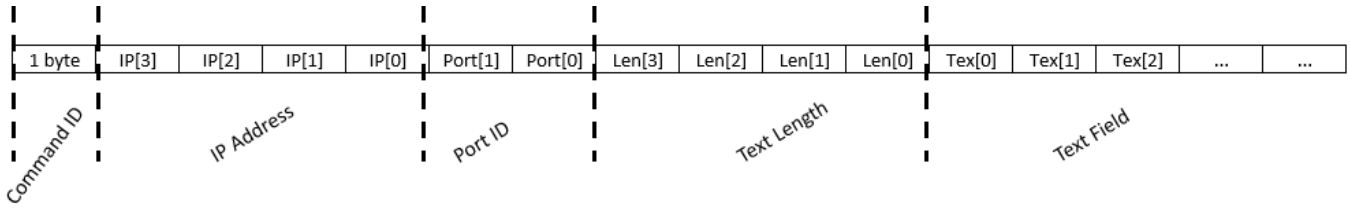

Figure 3: REQ_ECHO and RSP_ECHO Message Format

(b) `RSP_LISTUSERS` Command ID:

i. Number of Users: 2 bytes, in network byte order, used to indicate the number of users listed in this message; If the number of users in the list is N, then there are N pair of the following fields.

ii. IP Address: 4 bytes, IP address for a user (socket), in network byte order.

iii. Port Number: 2 bytes, TCP port number for a user (socket), in network byte order.

An N users `RSP_LISTUSERS` message format is shown in Figure 4.

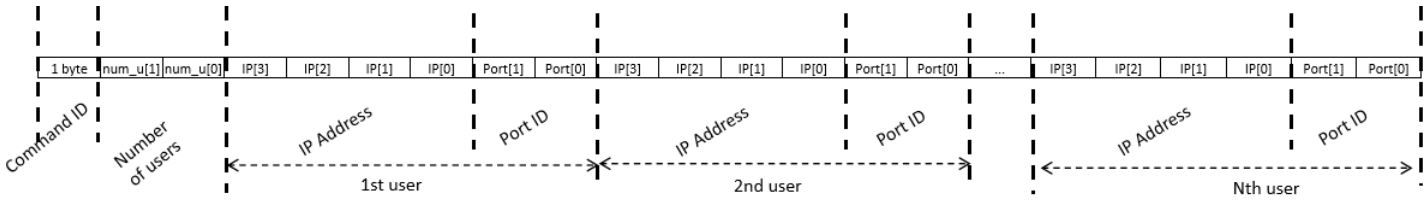

Figure 4: RSP_LISTUSERS Message Format:

(c) `REQ_QUIT`, `REQ_LISTUSERS` or `ECHO_ERROR` Command ID:
No other fields but Command Id is required.

In order to set and get the value in network byte order, you can use **htonl** or **htons** to convert the host byte order to network byte order at the sender side and use **ntohl** or **ntohs** to convert the network byte order to host byte order at the receiver side.

5. When the client sends `REQ_ECHO` command message, the server will not echo this type of message. Instead, it helps identify the destination IP address and TCP port number from the message and search its connected clients/users.

(a) If the server finds no match for the given IP address and port number in the list of the TCP connections maintained by itself, it sends back `ECHO_ERROR`, following the given message format to the client and the client will only print "Echo error" to the screen.

(b) If there is a match, the server will replace the received message field of IP address and port number with the source user's of IP address and port number and forward to the matched user, shown in Figure 2.

(c) The matched user is the destination of the echo message. Once it receives, it should obtain the source user's IP address and TCP port number - as well as the text from the received message, and show on the screen. It also constructs a

message of `RSP_ECHO` type with the message format defined above, which is the same as that for `REQ_ECHO`, with the echo request user as the destination user in this `RSP_ECHO` message, which is sent to the server, shown in Figure 2.

(d) The server forwards the received `RSP_ECHO` message to the source user after replacing the IP address and port number with the response user, shown in Figure 2.

(e) The source receives `RSP_ECHO` message and display the text in the message.

6. There are three **directives** in the input text that may be typed by the user.

   (a) /t : same as the Assignment 1.
   For example, keying in "/t 02c0a80062426a00000005776f726c64" means the client would ignore "/t" and interpret the subsequent input as the hexadecimal values. The client should locally discard "/t" and pack all the hexadecimal values into the message that is sent with the following bytes: 02, c0, a8, 00, 62, 42, 6a, 00, 00, 00, 05, 77, 6f, 72, 6c, 64 respectively (in network byte order).

   (b) /q : same as the Assignment 1.
   NOTE: It is necessary for the server to be able to accept multiple clients at the same time. The server will not echo this type of message to the client and the client will not print to the screen.

   (c) /e : Echo request.
   For example "/e 192.168.0.98:10000 hello world", the client should identify the directive "/e" and the destination IP address 192.168.0.98 and port number 10000, and echo the message "hello world".

   (d) /l : request to list all uses, which is equivalent to the `REQ_LISTUSERS` command.

7. There are two echo transation examples here:

   (a) The sequence diagram shown in Figure 5 illustrates the example of messaging between the server and two clients (client1 and client2).

      i. Client 1 types in e.g. "/e 192.168.0.98:10000 hello world", the Client 1's program will take the destination (Client 2) IP address 192.168.0.98 and port number 10000 that will echo the message "hello world" to Client 1. Both Client 1 and Client 2 send the message to the server with the TCP connection.

      ii. The server helps forward the message from Client 1 to Client 2 after changing the IP address and TCP port number in the message.

      iii. Client 2 will print that text to the screen (e.g. "hello world", and then using `RSP_ECHO` command, sends it to the server.

      iv. The server helps forward the `RSP_ECHO` command message to Client 1 after changing the IP address and TCP port number in the message.

      v. At the end, the `RSP_ECHO` command message is sent to Client 1. Client 1 will also print the text to the screen.

   (b) The sequence diagram shown in Figure 6 illustrates the example of messaging between the server and two clients (client1 and client2). But the message fails to echo back due to client2 disconnection.

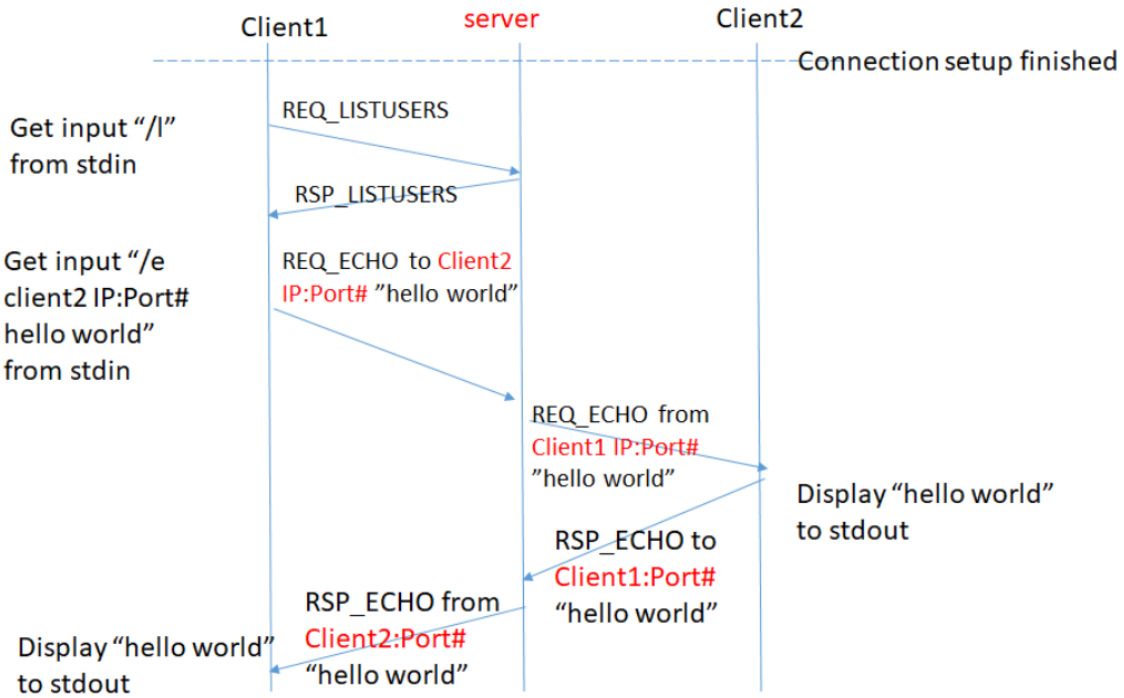

Figure 5: Communication transaction sequence 1

i. Client 1 types in e.g. "/e 192.168.0.98:10000 hello world",
ii. Client 2 disconnects before server receives the `REQ_ECHO` message.
iii. The server searches the list information for client2 and finds no match. It sends `ECHO_ERROR` command message to Client 1.
iv. Client 1 receives `ECHO_ERROR` command message and displays "Echo error" on the screen.

8. Please note that the receiver may process a few times to receive a complete TCP packet when it uses a fixed size buffer to receive a TCP packet with a unknown size.

9. We assume you use taskqueue.h and taskqueue.hpp to provide multi-threading for server. You should use std::thread for your client.

   (a) The pre-threading model given by taskqueue.h and taskqueue.hpp can be used by the code for the server. It creates a pool of threads and queues the socket of the accepted TCP connection. The thread once obtains the socket from the queue and handles accordingly, which will take charge of the data communication with a client. In this thread function, the server will send and receive the message from the connecting client. If the server starts N threads, then there are maximum N clients that can connect to the server.

   (b) The client creates a thread with the function to handle the receiving of the message from the server. If the client needs to respond, it will send the message to the server in this thread function. The main thread accepts the input from the stdin. If the input is a command to send a message, the client will construct a message and send to the server. All the command message receiving will be

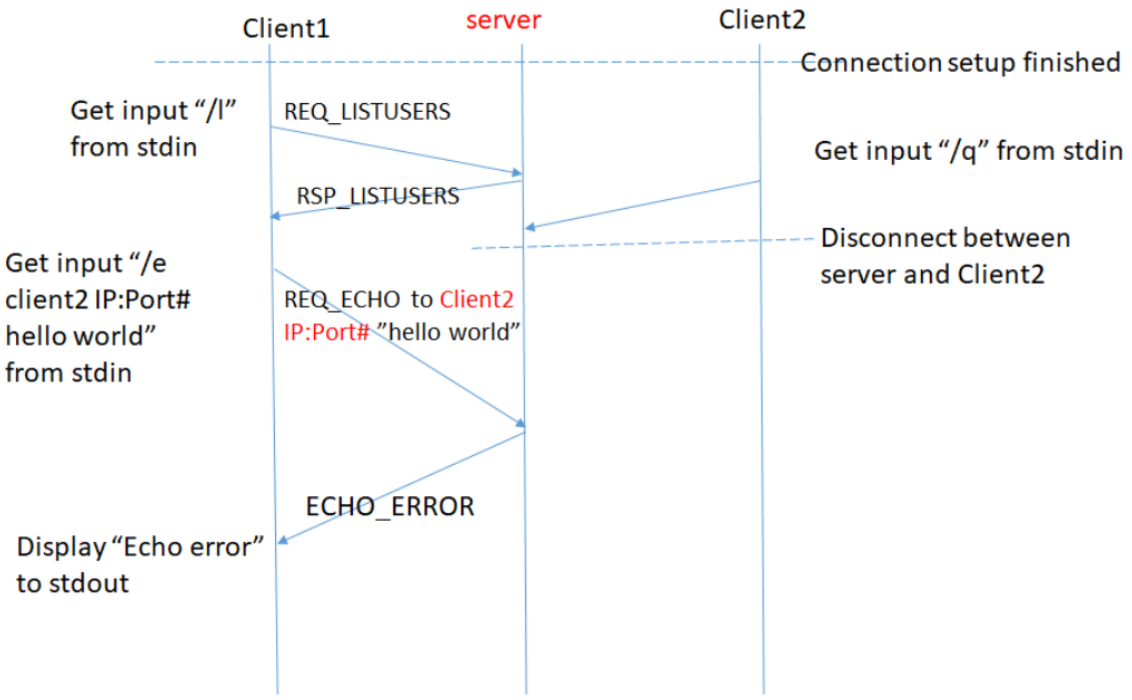

Figure 6: Communication transaction sequence 2

handled by the thread function. **Keying in the input from the stdin should not block the receiving and sending to the server and vice verse**.

10. To facilitate the testing using bash script (likely used by our grading system), the following code can be used, so that it could create some latency between two inputs from stdin or redirect input from the file.

```
1
2       #include <thread>
3       //Insert the following code
4       #ifdef DEBUG_ASSIGNTMENT2_TEST
5       using namespace std::chrono_literals;
6       std::this_thread::sleep_for(5000ms);
7       #endif
8
9       //after the stdin receives the input and sends the command message
    accordingly, i.e.,
10
11      ///call std:cin << ...
12
13      ...
14
15      ///call send()
16      #ifdef DEBUG_ASSIGNTMENT2_TEST
17      using namespace std::chrono_literals;
18      std::this_thread::sleep_for(5000ms);
19      #endif
20
21
```

11. Please refer to the sample executable file for print out format.

2 Rubrics

1. An improper submission (e.g., incorrect file names, missing files etc.) results in a penalty even after re-submission.
2. Your submitted C++ files will be compiled in release mode. Any warnings result in a penalty.
3. Comment & coding style, including:
   - Well-commented in your own words (e.g. using the given demo code's comments or reusing sentences from this PDF does not count.)
   - Indentation
   - Variable naming
   - No magic numbers
4. The server program should be able to prompt for the user to enter a port number.
5. The server program should be able to print the IP address and port number of the listening socket.
6. The client program should be able to prompt for the IP address and port number of the server to which the client will connect.
7. Once the server accepts the connection request, it should print the IP address and port number of the connecting client.
8. Message construction and command processing: The client should be able to construct the correct message following the given format. The command should be able to be processed accordingly and correctly by the server.
9. Echo message
   (a) Whatever is typed into at the client side should first be printed on the server and then printed on the client too.
   When the destination client echoes the message, the IP address and port number fields of the message should be changed. The modified message is then forwarded and echoed on the source client.
   (b) If nothing is typed into at the client side but there is something echoed or printed on the server, there will be a penalty.
10. Correctness and Robustness:
    (a) In our test case, it is expected that send() and recv() may be called multiple times to send/receive the message completely and process accordingly, due to the limited buffer size (of either OS or application). There will be a penalty for no such support.
    (b) You are expected to use non-blocking I/O on a socket. The example code is given in ServerNonBlocking.cpp. Please study carefully before use.
    (c) Command ID: If the server receives the command other than QUIT and ECHO, it should report the error.

11. The server should be able to receive multiple new client connections and handle the data communication on these connections concurrently/in parallel.

12. Please also test your programs on more than one machine.

13. To avoid the interleaving printout of the multiple threads, you should use a synchronization mechanism (please refer to taskqueue.hpp) to ensure a clean printout. You should use the mutex stdoutMutex to declare a lock, and use this before any print out.

```
1
2     std::lock_guard<std::mutex> usersLock{ _stdoutMutex };
3
4
```

14. The evaluation/grading on your submission will be done by cross-testing and/or self-testing:

    (a) your server.exe with the reference/sample client.exe

    (b) your client.exe with the reference/sample server.exe

    (c) your own client.exe with your own server.exe

The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow re-submission or submission after the deadline.