Objective of this assignment:

- Develop and implement a simple application using UDP and TCP sockets. The application using UDP sockets was developed for *Programming Assignment 2*. The application using TCP sockets is due for **this** *Programming Assignment 3*.

What you need to do:

1. Implement a simple UDP Client-Server application (Done for *Programming Assignment 2*)
2. Implement a simple TCP Client-Server application (**this** *Programming Assignment 3*)

Objective:

The objective is to implement a client-server application using a safe method: start from a simple **working** code for the client and the server. **You must slowly and carefully *bend* (modify) little by little the client and server alternatively until you achieve your ultimate goal. You must bend and expand each piece alternatively the way a black-smith forges iron. From time to time save your working client and server such that you can roll-back to the latest working code in case of problems**. Failing to follow this incremental approach may result in a ball of wax impossible to debug in case your program does not behave or work as expected.

For this programming assignment, you are advised to start from the *Friend* client and server application to implement the calculator server. You implemented the calculator server using UDP for *Programming Assignment 2*). Now, you must use TCP for **this** *Programming Assignment 3*.

Part A: **Datagram socket programming** (**Done** for *Programming Assignment 2*) **>>>> See Part B**

The objective is to design a ***Calculating Server (CS)***. This calculating server performs bitwise boolean and arithmetic computations requested by a client on 16-bit signed integers. Your server must offer the following operations: 1) addition (+), 2) subtraction (-), 3) bitwise ***OR*** (|), 4) bitwise ***AND*** (&), 5) Shift Right (>>), and 6) Shift Left (<<).

A **client *request*** will have the following format:

Field	TML	Request ID	Op Code	Number Operands	Operand 1	*Operand 2*
Size (bytes)	1	1	1	1	2	2

Where

1) **TML** is the Total Message Length (in bytes) including TML. It is an integer representing the **total** number of bytes in the message.
2) **Request ID** is the request ID. This number is generated by the client to differentiate requests. You may use a variable randomly initialized and incremented each time a request is sent.
3) **Op Code** is a number specifying the desired operation following this table

Operation	+	-	|	&	>>	<<	~
OpCode	0	1	2	3	4	5	6

4) **Number Operands** is the number of operands: 2 for (+, -, |, &) and shifts. It is 1 for ~ (NOT).
5) **Operand 1:** this number is the first or unique operand for all operations.
6) **Operand 2:** this number is the second operand for operations (+, -, |, &, <<, >>). It is the number of bits to shift by for the shift operations. This operand does NOT exist for the ~ (NOT) operation.

Operands are sent in the **network byte order** (i.e., big endian).

Hint: create a class object *Request* like "Friend", but with the information needed for a request.....

Below are two examples of requests

Request 1: suppose the Client requests to perform the operation 240 >> 4, i.e., shift the number 240 right by 4 bits (if this is the 7th request):

0x08	0x07	0x04	0x02	0x00	0xF0	0x00	0x04

Request 2: suppose the Client requests to perform the operation 240 – 160 (if this is the 9th request):

0x08	0x09	0x01	0x02	0x00	0xF0	0x00	0xA0

The **Server** will respond with a message with this format:

Total Message Length (TML)	**Request ID**	**Error Code**	**Result**
one byte	1 byte	1 byte	4 bytes

Where

1) **TML** is the Total Message Length (in bytes) including TML. It is an integer representing the **total** numbers of bytes in the message.
2) **Request ID** is the request ID. This number is the number that was sent as Request ID in the request sent by the client.
3) **Error Code** is **0** if the request was valid, and **127** if the request was invalid (Length not matching TML).
4) **Result** is the result of the requested operation.

In response to ***Request 1*** below

0x08	0x07	0x04	0x02	0x00	0xF0	0x00	0x04

the server will send back:

0x07	0x07	0x00	0x00	0x00	0x00	0x0F

In response to ***Request 2***,

0x08	0x09	0x01	0x02	0x00	0xF0	0x00	0xA0

the server would send back:

0x07	0x09	0x00	0x00	0x00	0x00	0x50

a) **Repetitive Server**: Write a datagram **Calculating Server (ServerUDP.java)** in **java**. This server must respond to requests as described above. The server must run on port (10010+*GID*) and could run on any machine on the Internet. ***GID*** is your Canvas team #. The server must accept a command line of the form: ***java ServerUDP portnumber*** where ***portnumber*** is the port where the server should be working. For example, if your Group ID (GID) is 13 then your server must listen on Port # 10023.

   Whenever a server gets a request, it must:

   i. print the request one byte at a time in hexadecimal (for debugging and grading purpose)
   ii. print out the request in a manner convenient for a typical Facebook user: the request ID and the request (operands and required operation)

b) Write a datagram **client** (**ClientUDP.java**) in java:

   i. Accepts a command line of the form: ***java* ClientUDP *servername PortNumber*** where ***servername*** is the server name and ***PortNumber*** is the port number of the server. Your program must prompt the user to ask for an ***Opcode,*** an ***Operand1*** and if needed an ***Operand2*** where ***OpCode*** is the opcode of the requested operation (See the opcode table). ***Operand1 and Operand2*** *(if applicable)* are the operands. For each entry from the user, your program must perform the following operations:

ii. form a message as described above
iii. send the message to the server and wait for a response
iv. print the message one byte at a time in hexadecimal (for debugging and grading purpose)
v. print out the response of the server in a manner convenient for a typical Facebook user: the request ID and the response
vi. print out the round trip time (time between the transmission of the request and the reception of the response)
vii. prompt the user for a new request.

Part B: **TCP socket programming (Due for THIS *Programming Assignment 3*)**

Repeat part A using TCP sockets to produce **(ServerTCP.java, ClientTCP.java)**.

How to get started?

1) Download all files (UDP sockets) to run the "Friend" application used in Module 2 to illustrate how any class object can be exchanged: Friend.java, FriendBinConst.java, FriendDecoder.java, FriendDecoderBin.java, SendUDP.java, and RecvUDP.java.

2) Compile these files and execute the UDP server and client. Make sure they work

3) Create a new folder called Request and duplicate inside it ALL files related to the Friend class object

4) Inside the Folder Request, change ALL occurrences of "Friend" with "Request" including the file names.

3) Adapt each file to your calculator application. Replace the fields used by Friend with the fields used by a request.

4) Aim to have the client send one request and have the server understand it (just like what we did with a friend object).

5) When your server will receive and print out correctly a request, then you need to send back a response...

6) Create a class object Response....

Report

- Write a report. The report should not exceed 2 pages.
- Your report state: whether your programs work or not (this must be just ONE sentence). If your program does not work, explain the obstacles encoutered.

What you need to turn in:

- Electronic copy of all your source programs (standalone). **In addition**, put all the source programs in a folder that you name with your concatenated last name and first name. Zip the folder and submit it **TOO**. The grader should see on Canvas all your source programs separately **AND** a zip folder containing all the source programs needed to compile and execute your program.
- Electronic copy of the report (including your answers) (standalone). Submit the file as a Microsoft Word or PDF file.

Grading

1) UDP/TCP **client** is worth 40% if it works well: communicates with YOUR server. Furthermore, screenshots of your client and server running on Tux machines must be provided. The absence of screenshots or screenshots on machines other than the Tux machines will incur a 15% penalty.

2) UDP/TCP **client** is worth 10% extra if it works well with a working server from any of your classmates.

1) UDP/TCP **server** is worth 40% if it works well: communicates with YOUR client. Furthermore, screenshots of your client and server running on Tux machines must be provided. The absence of screenshots or screenshots on machines other than the Tux machines will incur a 15% penalty.

2) UDP/TCP **server** is worth 10% extra if it works well with a working client from any of your classmates.