Objective of this assignment:

- Develop and implement a simple application using UDP and TCP sockets. The application using TCP sockets will be due in Programming Assignment 3.
- Understand the issue of "*data representation*".

What you need to do:

1. Implement a simple UDP Client-Server application (Programming Assignment 2)
2. Implement a simple TCP Client-Server application (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.

For this programming assignment, you may start with the provided *Friend* client and server application to implement a "*Compressing*" server. You may use *Friend* if you are using Java. You can use any programming language to implement this programming assignment as long as the programming language is already available on the Engineering Tux Machines. You will first implement the "*Compressing*" server using UDP (Programming Assignment 2), and then TCP (Programming Assignment 3).

Part A: **Datagram socket programming** (Programming Assignment 2)

The objective is to design a ***"Compressing" Server (CS)***. This *compressing* server will receive a string of characters representing an integer n and will echo n using its machine representation. We assume that a string is represented using **UTF-16BE** encoding scheme. For example, suppose the server receives the string "567327" representing the number 567327. If the string "567327" was encoded using **UTF-16BE**, this means the server receives these bytes representing the ASCII codes of the characters '5', '6', '7', '3', '2', and '7' in hexadecimal:

Byte 1	Byte 2	Byte3	Byte 4	Byte 5	Byte 6
0x35	0x36	0x37	0x33	0x32	0x37

.

The server will *compress* the string "567327" into its machine representation, i.e. 0x8A81F. Therefore, for the string "567327", the server will echo the following bytes that are the representation of the number in hexadecimal

Byte 1	Byte 2	Byte 3
0x08	0xA8	0x1F

Hint: *Observe that the "compression" is simply the conversion of the string s into an integer n. The server needs simply to send back the integer n.*

a) **Repetitive Server**: Write a datagram **Compressing Server (ServerUDP.xxx)** in any language you prefer as long as it is already available on Engineering Tux machines. 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 group ID. The server must accept a command line of the form:

ServerUDP portnumber where ***portnumber*** is the port where the server should bind. For example, if your Group ID (GID) is 13 then your server must bind to Port # 10023.

b) Write a datagram **client** (**ClientUDP.xxx**) in any language you prefer as long as it is already available on Engineering Tux machines:

i. Accepts a command line of the form: **ClientUDP *servername PortNumber*** where ***servername*** is the server name and ***PortNumber*** is the port number of the server. Your program must repetively prompt the user to ask for a number *n* until the user enter 0 in which case the program terminates. For each entry n from the user, your program must perform the following operations:
ii. convert the number *n* into a string s.
iii. send the string s to the server using **UTF-16BE** encoding scheme and wait for a response
iv. print string s one byte at a time in hexadecimal (for debugging purpose)
v. print out the response of the server (normally the number n) in decimal
vi. print the response n in hexadecimal
vii. prompt the user for a new number n.
viii.

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

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

How to get started? (if you use Java and want to use the Friend template)

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 *Compressing* 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 half a page.
- Your report must 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 EACHyour source program separately (standalone). **In addition**, put all the source programs in a folder that you name with your group ID. Zip the folder and submit it TOO.
- 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.
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.
2) UDP/TCP server is worth 10% extra if it works well with a working client from any of your classmates.