- # General Description

Please read the WHOLE assignment description before start implementing.
We all know how bad the university's "Course Registration System" is, now you may have a chance to change it :-)
In this assignment you will implement a "Course Registration System" server and client. The communication between the server and the client(s) will be performed using a binary communication protocol.

The implementation of the server is based on the **Thread-Per-Client (TPC)** and **Reactor** servers taught in class.

You are required to implement the BGRS (Ben Gurion Registration System) protocol, which emulates a simple course registration system.
The messages in this protocol are binary numbers, composed of an **opcode** (*short* number of two bytes) which indicates the command, and the data needed for this command (in various lengths).
In the following sections, we define the specifications of the commands supported by the BGRS protocol.

Unlike real course registration systems, the courses are specified in one file, according to a specific format (shown below). The data which the server and clients get during their running are saved in the RAM (in data structures, *e.g.*, arrays, lists, etc…)

- ## Establishing a client/server connection

Upon connecting, a client must identify himself to the service. A new client will issue a Register command with the requested user name and password. A registered client can then login using the Login command. Once the command is sent, the server will acknowledge the validity of the username and password. Once a user is logged in successfully, she can submit other commands. The register and login commands are stated in the following section. Note that the register command will not perform automatic login (you will need to call login after it).

1.2 Courses File

The data about the courses (for the server use) are given by a text file which you will define beforehand. The file name **MUST** be **Courses.txt**, and it should be located in the main folder (the project folder). The file consists of lines, where every line refers to a specific course. The format of a single line will be as follows:
courseNum|courseName|KdamCoursesList|numOfMaxStudents
courseNum: the number of the course (100>= int >= 0)
courseName: the course name (non-empty string)
KdamCoursesList: the list of the Kdam courses. (format: **[course1Num, course2Num,...]**)
numOfMaxStudents: the maximum number of students allowed to register to this course(int >= 5)

No spaces before and after the **'|'** character - you don't need to check that.
The course name doesn't contain the **'|'** character - you also don't need to check that.
You can assume that the courses file is formatted well (means, all courses lines are written in the correct format), so no need to check that.
Example:
42|How to Train Your Dragon|[43,2,32,39]|25

1.3 Supported Commands

The BGRS protocol is composed of two types of commands, Server-to-Client and Client-to-Server. The commands begin with 2 bytes (short) to describe the **opcode** . The rest of the message will be defined specifically for each command as such:

2 bytes	Length defined by command
Opcode	…

We supplied functions that encode \ decode between 2 bytes and short for both java and C++ in the assignment page.
The BGRS protocol supports 11 types of messages:

- 1)Client-to-Server messages
- 2)Server-to-Client messages

Opcode	Operation
1	Admin register (ADMINREG)
2	Student register (STUDENTREG)
3	Login request (LOGIN)
4	Logout request (LOGOUT)
5	Register to course (COURSEREG)
6	Check Kdam course (KDAMCHECK)
7	(Admin)Print course status (COURSESTAT)
8	(Admin)Print student status (STUDENTSTAT)
9	check if registered (ISREGISTERED)
10	Unregister to course (UNREGISTER)

11	Check my current courses (MYCOURSES)
12	Acknowledgement (ACK)
13	Error (ERR)

ADMINREG Messages:

Messages have the following format:

2 bytes	string	1 byte	string	1 byte
Opcode	Username	0	Password	0

Messages that appear only in a Client-to-Server communication.
An ADMINREG message is used to register an admin in the service. If the username is already registered in the server, an ERROR message is returned. If successful an ACK message will be sent in return. Both string parameters are a sequence of bytes in UTF-8 terminated by a zero byte (also known as the '\0' char).

Parameters:

- Opcode: 1.
- Username: The username to register in the server.
- Password: The password for the current username (used to log in to the server).

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **ADMINREG <Username> <Password>**

STUDENTREG Messages:

Messages have the following format:

2 bytes	string	1 byte	string	1 byte
Opcode	Username	0	Password	0

Messages that appear only in a Client-to-Server communication.
A STUDENTREG message is used to register a student in the service. If the username is already registered in the server, an ERROR message is returned. If successful an ACK message will be sent in return. Both string parameters are a sequence of bytes in UTF-8 terminated by a zero byte (also known as the '\0' char).

Parameters:

- Opcode: 2.
- Username**:** The username to register in the server.
- Password: The password for the current username (used to log in to the server).

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **STUDENTREG <Username> <Password>**

LOGIN Messages:

Messages have the following format:

2 bytes	string	1 byte	string	1 byte
Opcode	Username	0	Password	0

Messages that appear only in a Client-to-Server communication.
A LOGIN message is used to login a user into the server. If the user doesn't exist or the password doesn't match the one entered for the username, sends an ERROR message. An ERROR message should also appear if the current client has already successfully logged in.
Both string parameters are a sequence of bytes in UTF-8 terminated by a zero byte.

Parameters:

- Opcode: 3.
- Username**:** The username to log in the server.
- Password: The password for the current username (used to log in to the server)

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **LOGIN <Username> <Password>**

LOGOUT Messages:

Messages have the following format:

2 bytes

Opcode

Messages that appear only in a Client-to-Server communication. Informs the server on client disconnection. Client may terminate only after receiving an ACK message in replay. If no user is logged in, sends an ERROR message.

Parameters:

- Opcode: 4.

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **LOGOUT**
- Once the **ACK** command is received in the client, it must terminate itself.

COURSEREG Messages:

Messages have the following format:

2 bytes	2 bytes
Opcode	Course Number

Messages that appear only in a Client-to-Server communication. Inform the server about the course the student want to register to, if the registration done successfully, an ACK message will be sent back to the client, otherwise, (e.g. no such course is exist, no seats are available in this course, the student does not have all the Kdam courses, the student is not logged in) ERR message will be sent back. (Note: the admin can't register to courses, in case the admin sends a COURSEREG message, and ERR message will be sent back to the client).

Parameters

Opcode: 5

Course Number: the number of the course the student wants to register to.

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **COURSEREG <CourseNum>**

KDAMCHECK Messages:

Messages have the following format:

2 bytes	2 bytes
Opcode	Course Number

Messages that appear only in a Client-to-Server communication.

KDAMCHECK this message checks what are the KDAM courses of the specified course.
If student registered to a course successfully, we consider him having this course as KDAM

Parameters:

- Opcode: 6.
- Course Number: the number of the course the user needs to know its KDAM courses

When the server gets the message it returns the list of the KDAM courses, in the SAME ORDER as in the courses file (if there are now KDAM courses it returns empty string).

Command initiation:

- This command is initiated by entering the following texts in the client command line interface:

KDAMCHECK <CourseNumber>

COURSESTAT Messages:

Messages have the following format:
(Admin Message)

2 bytes	2 bytes
Opcode	Course Number

Messages that appear only in a Client-to-Server communication.
The admin sends this message to the server to get the state of a specific course.
the client should prints the state of the course as followed:
Course: (<courseNum>) <courseName>
Seats Available: <numOfSeatsAvailable> / <maxNumOfSeats>
Students Registered: <listOfStudents> //ordered alphabetically
Example:
Course: (42) How To Train Your Dragon
Seats Available: 22/25
Students Registered: [ahufferson, hhhaddock, thevast]
//if there are no students registered yet, simply print []

Parameters:

- Opcode: 7

Course Number: the number of the course we want the state of.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface:

COURSESTAT <courseNum>

STUDENTSTAT Messages:

(Admin Message)
Messages have the following format:

2 bytes	String	1 byte
Opcode	Student Username	0

Messages that appear only in a Client-to-Server
A STUDENTSTAT message is used to receive a status about a specific student.
the client should print the state of the course as followed:
Student: <studentUsername>
Courses: <listOfCoursesNumbersStudentRegisteredTo>
//ordered in the same order as in the courses file
Example:
Student: hhhaddock
Courses: [42] // if the student hasn't registered to any course yet, simply print []

Parameters:

- Opcode: 8.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **STUDENTSTAT <StudentUsername>**

ISREGISTERED Messages:

Messages have the following format:

2 bytes	2 bytes

Opcode	Course Number

Messages that appear only in a Client-to-Server communication.
An ISREGISTERED message is used to know if the student is registered to the specified course.
The server send back “REGISTERED” if the student is already registered to the course,
otherwise, it sends back “NOT REGISTERED”.

Parameters:

- Opcode: 9.
- Course Number**:** The number of the course the student wants to check.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **ISREGISTERED <courseNum>**

UNREGISTER Messages:

Messages have the following format:

2 bytes	2 bytes
Opcode	Course Number

Messages that appear only in a Client-to-Server communication.
An UNREGISTER message is used to unregister to a specific course
The server sends back an ACK message if the registration process successfully done, otherwise, it sends back an ERR message.

Parameters:

- Opcode: 10.
- Course Number**:** The number of the course the student wants to unregister to.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **UNREGISTER <courseNum>**

MYCOURSES Messages:

Messages have the following format:

2 bytes

Opcode

Messages that appear only in a Client-to-Server communication.
A MYCOURSES message is used to know the courses the student has registered to.
The server sends back a list of the courses number(in the format:**[<coursenum1>,<coursenum2>]**) that the student has registered to (could be empty **[]**).

Parameters:

- Opcode: 11.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **MYCOURSES**

ACK Messages:

Messages have the following format:

2 bytes	2 bytes	-	1 byte
Opcode	Message Opcode	String to be printed at the client side (as bytes)	0

Messages that appear only in a Server-to-Client communication.
ACK Messages are used to acknowledge different Messages. Each ACK contains the message number for which the ack was sent. In the optional section there will be additional data for some of the Messages (if a message uses the optional section it will be specified under the message description).
All Messages that appear in a Client-to-Server communication require an ack/error message in response.

Parameters:

- Opcode: 12.
- Message Opcode: The message opcode the ACK was sent for.
- Optional: changes for each message.

Client screen output:

- Any ACK message received in **client** should be written to the screen in the following manner:

ACK <Message Opcode>

<Optional>

- Printing of the optional part: split between optional parameters by space.

ERROR Messages:

Messages have the following format:

2 bytes	2 bytes
Opcode	Message Opcode

Messages that appear only in a Server-to-Client communication.
An ERROR message may be the acknowledgment of any other type of message.
In case of error, an error message should be sent.

Parameters:

- Opcode: 13.
- Message Opcode: The message opcode the ERROR was sent for.

Error Notification:

- Any error message received in **client** should be written to screen:

ERROR <Message Opcode>

In any case, if the message cannot be processed successfully, or the user is not allowed to send this message (e.g. student sends an admin message), the server returns an ERROR message.

- # Implementation Details

- ## General Guidelines

  - The server should be written in Java. The client should be written in C++ with BOOST. Both should be tested on Linux installed at CS computer labs or the provided VM.

  - You must use maven as your build tool for the server and MakeFile for the c++ client.

  - The same coding standards expected in the course and previous assignments are expected here.

- **Server**

You will have to implement a single protocol, supporting both the **Thread-Per-Client** and ***Reactor*** server patterns presented in class. Code seen in class for both servers is included in the assignment wiki page. You are also provided with a new class:

- **Database** – This is a singleton class that should be used in order to manage the Courses and Users. You must only use this class in order to do that. Notice that this class should be thread-safe.
-
    - **(UserName)** – not required) **String getUserPassword(String**

Left to you, are the following tasks:

1. Finish the *Database* class.
2. Implement the *MessagingProtocol* and *MessageEncoderDecoder* to support the BGRS protocol as described in section 1.2. **The implementations MUST not be specific for the protocol implementation.** You will also need to define messages(<T> in the interfaces). You may add more classes as necessary to implement the protocol (shared protocol data ect…).

Leading questions:

- Which classes and interfaces are part of the Server pattern and which are part of the Protocol implementation?
- When and how do I register a new user in the **Database** class.
- Which class(es) use the **Database** class, apart from the **initialize**?
- When do I call **initialize** to initiate the **Database**?

Tip:

- You can test tasks 1 by fixing one of the examples in the impl folder in the supplied spl-net.zip to work with your implementations (easiest is the echo example)

Testing run commands:

- Reactor server**:**

mvn exec:java -
Dexec.mainClass=”bgu.spl.net.impl.BGRSServer.ReactorMain” -

Dexec.args=”<port> <No of threads>”

- Thread per client server**:**

mvn exec:java -
Dexec.mainClass="bgu.spl.net.impl.BGRSServer.TPCMain" -

Dexec.args="<port>"

The **server** directory should contain a **pom.xml** file and the **src** directory.
Compilation will be done from the server folder using:
mvn compile

- ## Client

An echo client is provided, but its a single threaded client. While it is blocking on stdin (read from keyboard) it does not read messages from the socket. You should improve the client so that it will run 2 threads. One should read from the keyboard while the other should read from socket. The client should receive the server's IP and PORT as arguments. You may assume a network disconnection does not happen (like disconnecting the network cable). You may also assume legal input via keyboard.
The client should receive commands using the standard input. Commands are defined in section 1.2 under command initiation sub sections. You will need to translate from keyboard command to network messages and the other way around to fit the specifications.
Notice that the client should close itself upon reception of an **ACK** message in response to an outgoing **LOGOUT** command.
The **Client** directory should contain a src, include and bin subdirectories and a Makefile as shown in class. The output executable for the client is named **BGRSclient** and should reside in the bin folder after calling **make**.
<u>Testing run commands:</u> BGRSclient <ip> <port>

- # Submission instruction

  - Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
  - You must submit one .tar.gz file with all your code. The file should be named "ID#1_ID#2.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar,

  .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.

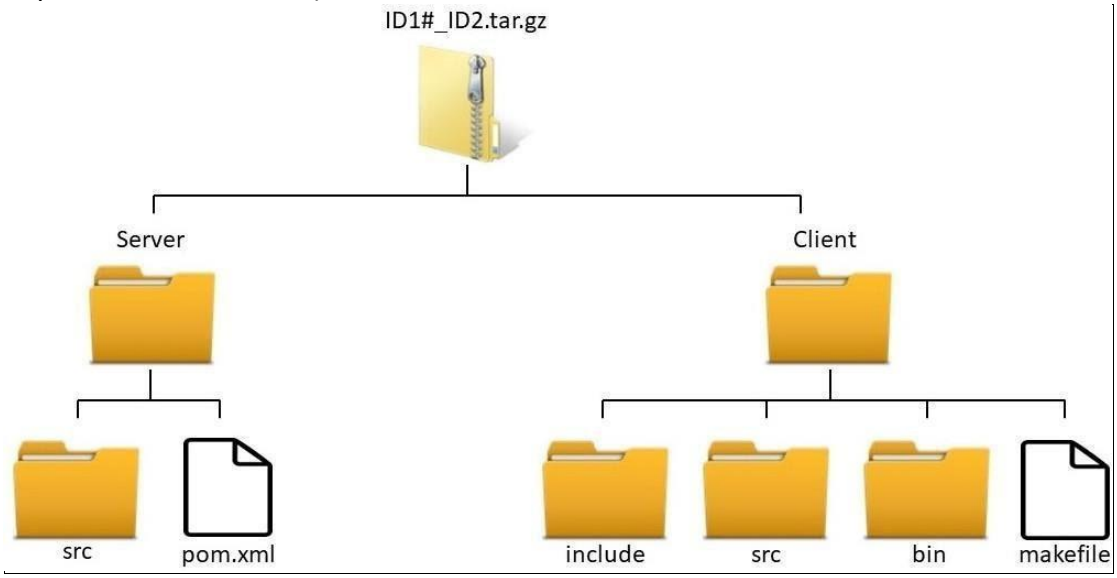
  - Extension requests are to be sent to Dolav. Your request email must include the following information:

    - Your name and your partners name.

- Your id and your partners id.
- Explanation regarding the reason of the extension request.
- Official certification for your illness

or army drafting. Requests without a compelling reason will not be accepted

- The submitted file should contain a **Client** directory and a **Server** directory (Their content was explained in the implementation section).



- ## Examples

The following section contains examples of commands running on clients. It assumes that the software opened a socket properly and a connection has been initiated.
We use “CLIENT#No<” and “CLIENT#No>” to annotate client #No terminal input (keyboard) \ output (screen print). The order of commands matches the order of reception on the server. Server and client actions are explained in between.
Note that the examples do not show the actual structure of the network messages, just the input \ output on the client terminal. The translation should be done according to specifications in section 1.2.

- ### Registration and login

Server assumptions for example:

- Server currently has 1 registered user named “Morty” with password “a123”

CLIENT#1< LOGIN Morty a321
CLIENT#1> ERROR 2
(Failed because of wrong password)
CLIENT#1< LOGIN Rick a123
CLIENT#1> ERROR 2
(Failed because username Rick isn't registered)
CLIENT#1< LOGIN Morty a123
CLIENT#1> ACK 2

CLIENT#2< LOGIN Morty a123
CLIENT#2> ERROR 2
(Failed because Morty is already logged-in)
CLIENT#2< LOGOUT
CLIENT#2> ERROR 7
(Failed because client #2 isn't logged in)
CLIENT#2< STUDENTREG Rick pain
CLIENT#2> ACK 1

CLIENT#1< LOGOUT
CLIENT#1> ACK 3
(client 1 closes)

CLIENT#2< LOGOUT
CLIENT#2> ERROR 3
(client 2 did not login)

- **Kdam and Unregistering from courses**

Server assumptions for example:

- No users are registered (via STUDENTREG).
- Courses file:

  82|Swordsmanship for Heros|[30,12]|2
  35|Swordsmanship: From Hero to King|[82,30,12]|1
  12|Advance Swordsmanship|[30]|5
  30|Intermediate Swordsmanship|[]|25

CLIENT#1> STUDENTREG Zoro roronoaZ
CLIENT#1< ACK 2
CLIENT#1> LOGIN Zoro roronoaZ
CLIENT#1< ACK 3
CLIENT#1> KDAMCHECK 35
CLIENT#1< ACK 7
CLIENT#1< [82,12,30]
CLIENT#1> COURSEREG 35
CLIENT#1< ERROR 5 (note that this message has sent because the student doesn't have all the Kdam courses)
CLIENT#1> KDAMCHECK 82
CLIENT#1< ACK 7
CLIENT#1< [12,30] (be careful, we print the courses in the order they listed in the file, not as they appear in the course 482 line)

CLIENT#1> COURSEREG 12
CLIENT#1< ERROR 10 (not registered to 30, which is a Kdam course for 12)
CLIENT#1> COURSEREG 30
CLIENT#1< ACK 5
CLIENT#1> COURSEREG 12
CLIENT#1< ACK 5
CLIENT#1> COURSEREG 82
CLIENT#1< ACK 5
CLIENT#1> ISREGISTERED 35
CLIENT#1< ACK 9
CLIENT#1< NOT REGISTERED
CLIENT#1> UNREGISTER 35
CLIENT#1< ERROR 10 (because the student is not registered to the course yet)
CLIENT#1> ISREGISTERED 82
CLIENT#1< ACK 9
CLIENT#1< REGISTERED
CLIENT#1> COURSEREG 35 (after been registered to all the Kdam courses of 35, he finally can register to 35)
CLIENT#1< ACK 5
CLIENT#1> UNREGISTER 12 (although the student has been registered to a course that 12 is a KDAM to, he can unregister himself from 12)
CLIENT#1< ACK 10

- ## Keyboard command to packet

In this section we will show a **few** keyboard commands and their matching network messages. Note that since network messages are just an array of bytes, we will print the hex values of those byte array.
Reminder: A byte can be represented by 2 hexadecimal values (0 to f), each representing 4 bits of the 8 in a single byte.

Keyboard command	**Message hex representation**
STUDENTREG Morty a123	00 02 4d 6f 72 74 79 00 61 31 32 33 00
LOGIN Morty a123	00 03 4d 6f 72 74 79 00 61 31 32 33 00
STUDENTSTAT Morty	00 08 4d 6f 72 74 79 00
COURSESTAT 32	00 07 00 20
LOGOUT	00 03