Goals: The goal of this assignment is to learn how to implement a multi-threaded server interacting with clients using two different kinds of communication models.

- In Part A, the clients and the server will communicate by explicitly exchanging messages over a TCP connection. You will send request and reply objects by serializing them and sending a sequence of bytes.
- In Part B, the clients will make service requests using RMI, and the server will be implementing the service as an RMI object, providing an RMI interface.

Useful Resources: Please study the following example programs on the course website.

1. Client-Server programming (TCP/IP)
   - EchoServer is a multi-threaded server, it simply sends back the data received from the client.
   - EchoClient connects to the server and sends a string on the TCP connection to the server and then receives data from the server.
2. Object Serialization and Communication
   - This example illustrates how an object is serialized and sent over a TCP connection by the SendObject program.
   - GetObject program reads two String objects, Date and String types, and prints them.
3. Java-RMI-Programming
   - Example of DateServer and DateClient.

PART A – TCP-Based Server (50 points): You will design and implement a multi-threaded server for a bank to store the account records for a set of people. Your server will maintain this data in the primary memory, in a hashtable. The Account objects stored in the hashtable will contain the following fields:

Class Account:
- UID: Integer
- balance: Integer

Each Account will have a unique UID, which will be used as a key for storing data in the hashtable of the server. The server will support remote invocations for the following four operations:

1. CreateAccount:
   - This will insert a new account object in the server’s hashtable. Initial balance will be zero.
   - input parameter: nothing
   - output parameter: UID for the new account will be returned by this parameter.

2. Deposit:
   - This will add a specified amount (positive integer) to the specified account:
   - input parameters: account UID and a positive Integer value for amount to be deposited.
   - output parameter: status (OK, or FAILED)
3. **GetBalance:**
   - input parameters: account UID
   - output parameter: Integer value for the current balance

4. **Transfer:**
   - This will transfer specified amount (positive integer) from the specified source account to a target account provided the resulting new balance of the source account does not become negative.
   - input parameters: account UID for source, account UID for target, a positive integer value for the amount to be transferred.
   - output parameter: status (OK or FAIL)

The client and the server will be using a TCP connection. To invoke one of the above operations on the server, the client will open a TCP connection to the server, and send a "Request" object to the server, in the serialized form. The Request object will contain the name of the requested operation and the parameters. The server will receive the object from the TCP connection stream, invoke the requested operation based on the parameters included in the Request object, prepare a Response object and send it to the client.

*(Hint: You may define Java class Request containing the name of the operation to be invoked. For each operation, you can then define a subclass of Request to contain the required parameters as its members. For example, you will define NewAccountRequest class as a subclass of Request. The server will then cast the received Request object to the appropriate class based on the operation name. The same kind of scheme can be used for Response objects.)*

**Server Design Requirements:** Your server implementation must meet the following requirements:

1. The server must be multi-threaded, so it will be able to handle multiple requests concurrently. You must make sure that concurrent updates to the records are properly synchronized.
2. The server will write to a log file (named `serverLogfile`) all requests, their parameter values, and the return status. This will help you in debugging and help us testing your program.

**Client Program Requirement:** You will be writing a multithreaded client program, called `BankClient` with the functionalities described below.

**BankClient:** This will be a multi-threaded client, in which each client thread will perform a sequence of operations as noted below. The number of client threads to be created will be specified as an argument to this program. Another parameter to this program will specify the number of operations each thread will execute in step 4 below. This client program will also log to a logfile (named `clientLogfile`) some of the operation response status.

*Example:* java BankClient serverHostname severPortnumber threadCount iterationCount

1. The main thread will first sequentially create 100 accounts on the server.
2. The main thread will the sequentially deposit 100 in each of these accounts.
3. It will now sequentially execute GetBalance on each account, and print sum of the total balance of all accounts. This value should be 10,000.
4. The main thread will now create the specified number of client threads, and each thread will perform the following sequence of operations for the specified number of iterations:
a. Randomly pick two accounts and transfer 10 from one to another. In case the operation fails due to insufficient balance, the client thread will write to the clientLogfile the response status and the IDs of the two accounts.

b. After performing the specified number of iterations of step (a) above, the client thread will terminate.

5. The main thread will wait for the completion of all of the client threads created in step 4 above.

6. The main thread will now sequentially execute GetBalance on each account, and print sum of the total balance of all accounts. This value should be again 10,000.

Submission requirements:
(a) Server program code.
(b) Programs for the client
(c) A Readme file containing the following information
   a. How to compile and run (in detail any configuration file or parameters).
   b. Any known errors (bugs) in your program.
   c. Any information TA has to know.

PART B – RMI-Based Server (50 points): *(If you want, you can do this part using Apache Thrift instead of Java RMI.)* You will implement the same server and the client programs using Java RMI protocol as the interface for the clients to invoke the above listed four operations. You will need to define an interface class for these operations to be provided through RMI. Your server code will implement these interface methods.

You may have to run your rmiregistry on a port other than 1099, because you may not have access to this port on CSE Lab machines. Therefore make sure that your client program and the server programs are properly written to connect with the registry running at some specified port.

The requirements for designing and implementing the server and the client programs are exactly the same as in Part A.

Submission Requirements: These requirements are the same as in Part A.
Submission Instructions:
- Please include names and student-IDs of all group members.
- Submit one UNIX tar file containing Parts A (TCP based) and Part B (RMI based) in separate directories.

Grading Criteria: For each part the grading will be based on the following allocation of the 50 points:

a. Multithreading of the client and server (5 points)
b. Correctly maintaining the log files for the client and the server (5 points)
c. Correctly creating accounts (10 points)
d. Correctly depositing amounts in step (2) of the client (10 points)
e. Correctly getting the balance of all accounts in step (3) of the client (5 points)
f. Correctly performing Transfer operations (10 points)
g. Correct verification in step (6) of the client for total balance of 10,000 (5 points)