Distributed Class Management System (DCMS) using Java RMI

Techniques used :

1) Multithreading :

Multitasking is when multiple processes share common processing resources such as a CPU.

Following are the stages of the life cycle −

- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

2) Socket Programming :

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

Here we are using Udp programming. It uses dtagram packets that provide functions to make a packet to transmit, which includes array of bytes containig message, Length of message, Internet address, port number. Dtagram sockets provides support to send receive UDP datagram packets. Inetaddress is used for server.

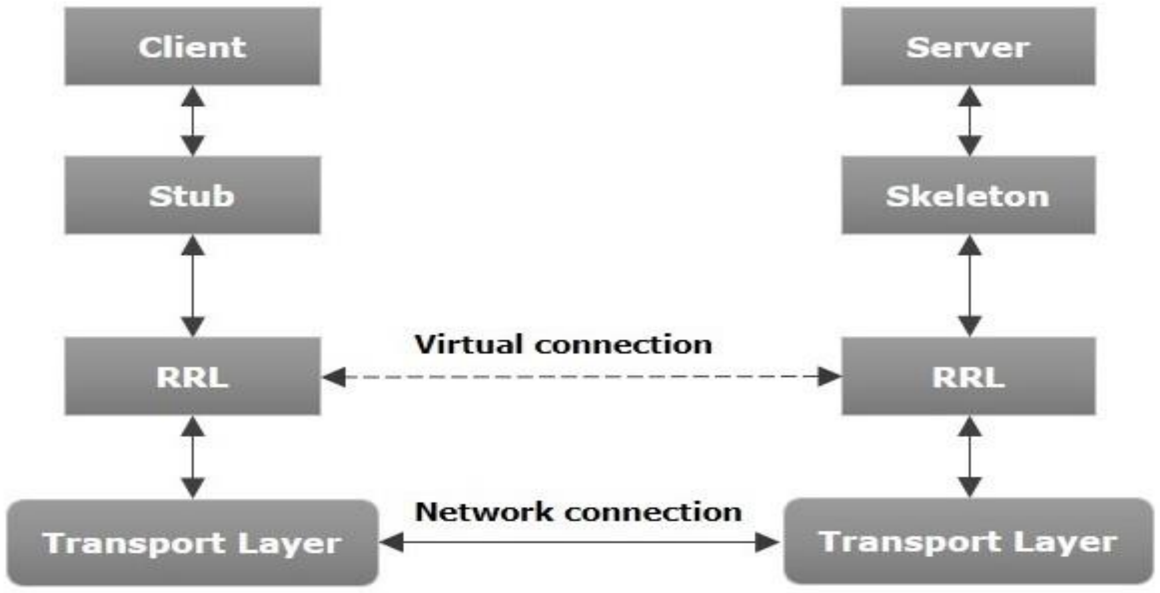
3) RMI :

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM. RMI is used to build distributed applications; it provides remote communication between Java programs.

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** − This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.

- **RRL(Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application

The following points summarize how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshalling.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

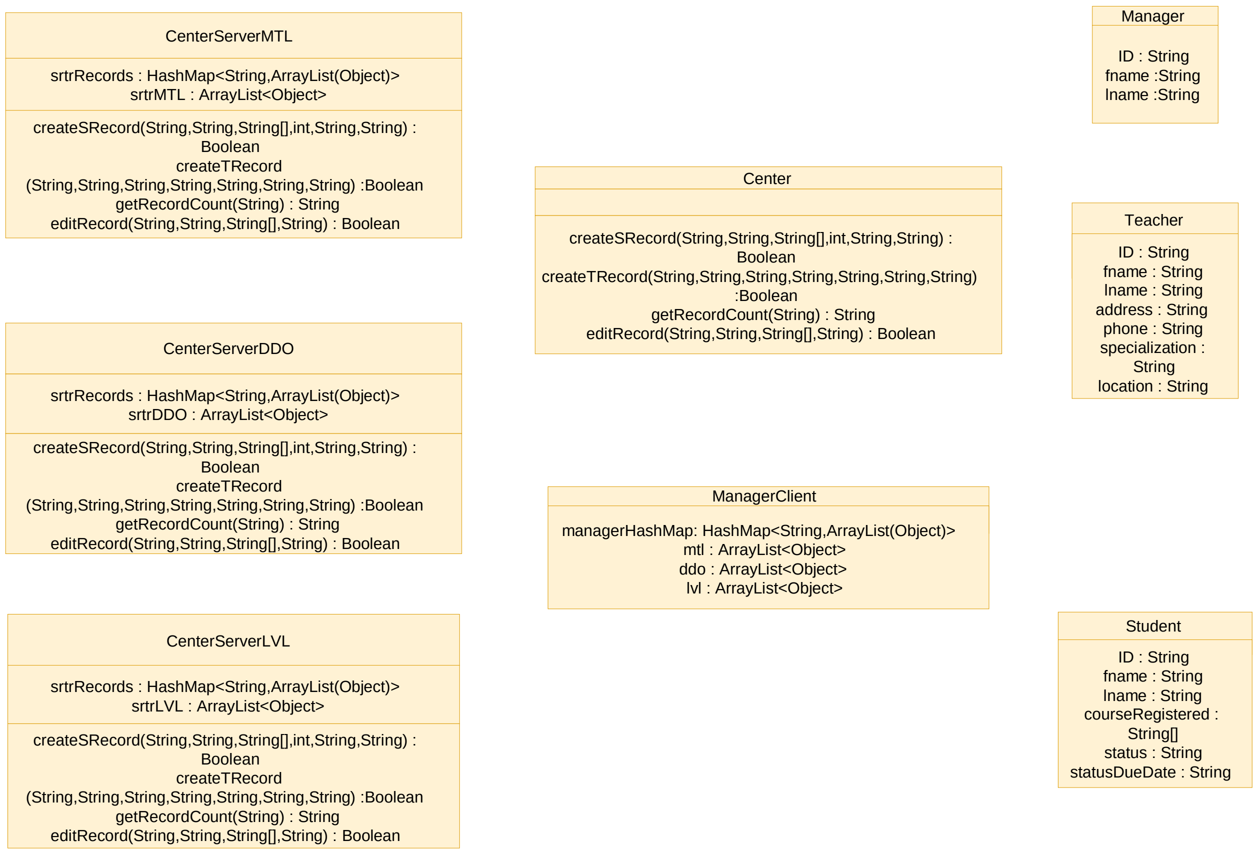
RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method).

Architecture :

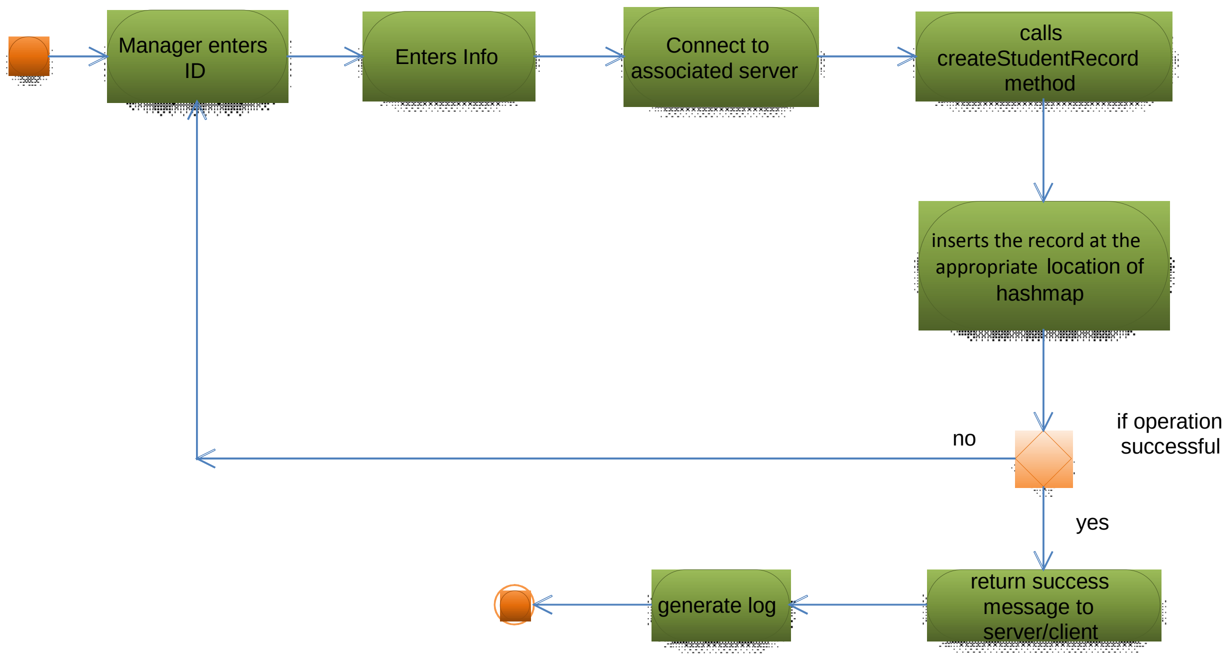
- **Interface(Center.java) :** It is java RMI Interface definition with four methods.
- **ManagerClient :** It is a client which invokes the center's server system to test the correct operation of the DCMS invoking multiple CenterServer. Initially there are some managers record.
- **CenterServer :** There are 3 center servers for Montreal, Laval and DDO. Each has their own hashmap to store the student and teacher record. Initially there are few records for student and teacher.
    - Hashmap maintains key and value pairs. Here we have used Hashmap<String, ArrayList<Object>> where String is the the String between A to Z. Arraylist strores the list of students and teachers according the first character of the last name. e.g. all the records whos last name starts with "A" will be stored in arraylist "a" and then put into hashmap with key "A".
- **LogHelper :** It to generate the log of the activities performed during the execution of the program.


CenterServerMTL
srtrRecords : HashMap<String,ArrayList(Object)>
srtrMTL : ArrayList<Object>
createSRecord(String,String,String[],int,String,String) : Boolean
createTRecord (String,String,String,String,String,String,String) :Boolean
getRecordCount(String) : String
editRecord(String,String,String[],String) : Boolean
Manager
ID : String
fname :String
lname :String
Center
createSRecord(String,String,String[],int,String,String) : Boolean
createTRecord(String,String,String,String,String,String,String) :Boolean
getRecordCount(String) : String
editRecord(String,String,String[],String) : Boolean
Teacher
ID : String
fname : String
lname : String
address : String
phone : String
specialization : String
location : String
CenterServerDDO
srtrRecords : HashMap<String,ArrayList(Object)>
srtrDDO : ArrayList<Object>
createSRecord(String,String,String[],int,String,String) : Boolean
createTRecord (String,String,String,String,String,String,String) :Boolean
getRecordCount(String) : String
editRecord(String,String,String[],String) : Boolean
ManagerClient
managerHashMap: HashMap<String,ArrayList(Object)>
mtl : ArrayList<Object>
ddo : ArrayList<Object>
lvl : ArrayList<Object>
CenterServerLVL
srtrRecords : HashMap<String,ArrayList(Object)>
srtrLVL : ArrayList<Object>
createSRecord(String,String,String[],int,String,String) : Boolean
createTRecord (String,String,String,String,String,String,String) :Boolean
getRecordCount(String) : String
editRecord(String,String,String[],String) : Boolean
Student
ID : String
fname : String
lname : String
courseRegistered : String[]
status : String
statusDueDate : String

Test Scenario :

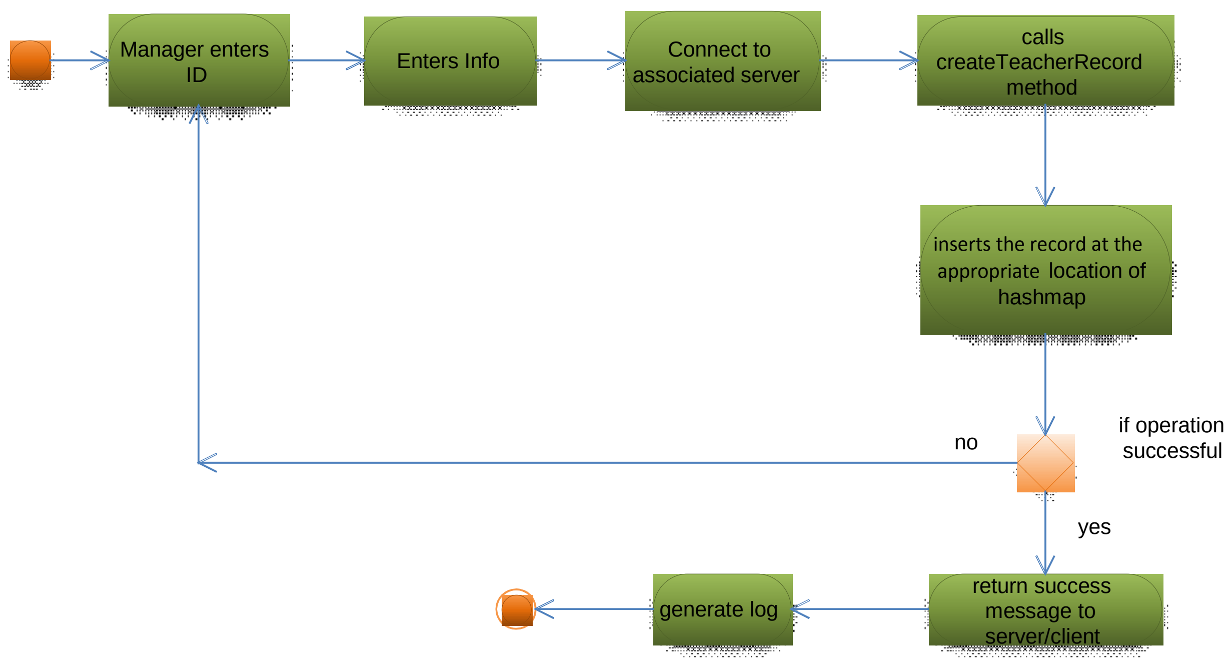
1) Create student

To test the method that is responsible to create the new student record. When a manager invokes this method from his/her center through a client program called *ManagerClient*, the server associated with this manager (determined by the unique *managerID* prefix) attempts to create a *TeacherRecord* with the information passed, assigns a unique *RecordID* and inserts the *Record* at the appropriate location in the hash map. The server returns information to the manager whether the operation was successful or not and both the server and the client store this information in their logs.


Manager enters ID
Enters Info
Connect to associated server
calls createStudentRecord method
inserts the record at the appropriate location of hashmap
if operation successful
no
yes
return success message to server/client
generate log

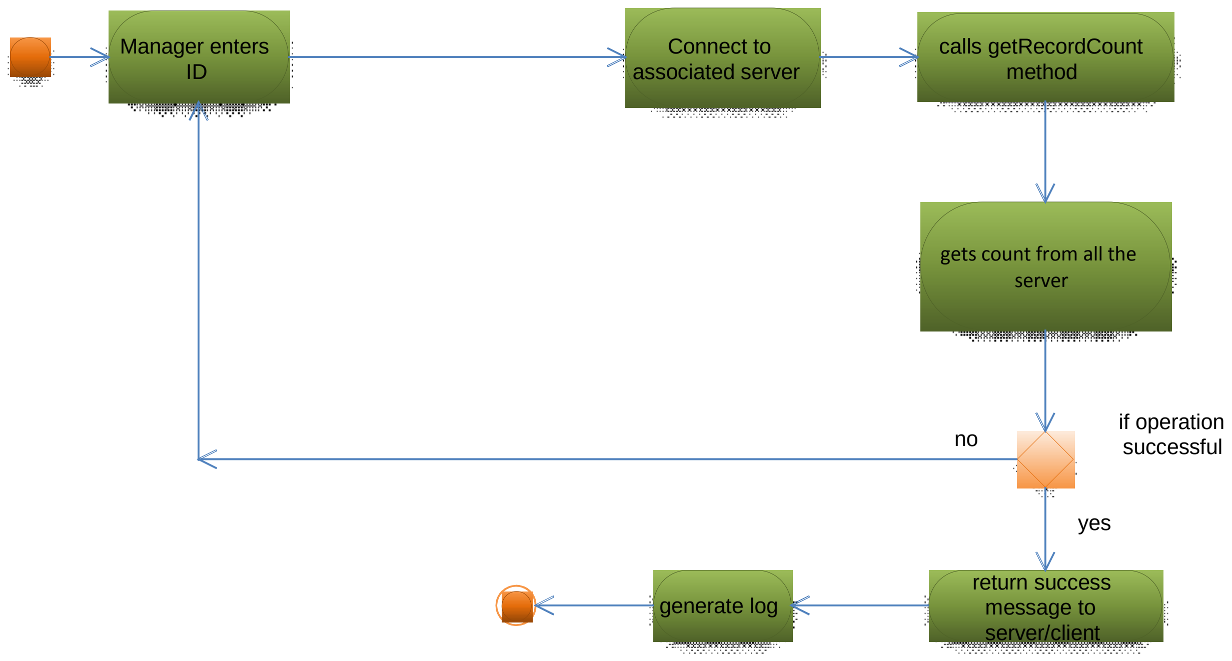
2) Create teacher

To test the method that is responsible to create the new teacher record. When a manager invokes this method from a *ManagerClient*, the server associated with this manager (determined by the unique *managerID* prefix) attempts to create a *StudentRecord* with the information passed, assigns a unique *RecordID* and inserts the *Record* at the appropriate location in the hash map. The server returns information to the manager whether the operation was successful or not and both the server and the client store this information in their logs.


Manager enters ID
Enters Info
Connect to associated server
calls createTeacherRecord method
inserts the record at the appropriate location of hashmap
if operation successful
no
yes
return success message to server/client
generate log

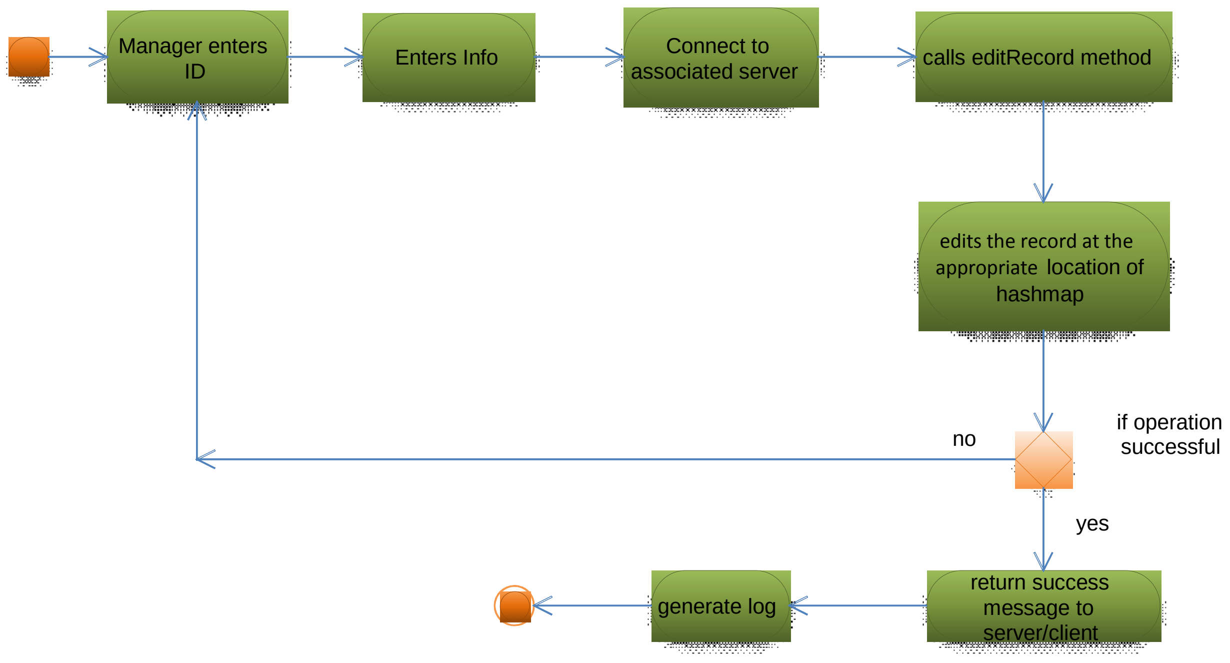
3) Get record count

To test the method that is responsible to return the number of records stored on all servers. A manager invokes this method from his/her *ManagerClient* and the server associated with that manager concurrently finds out the number of records (both TR and SR) in the other centers using UDP/IP sockets and returns the result to the manager. It only returns the record counts (a number) and not the records themselves. For example if MTL has 6 records, LVL has 7 and DDO had 8, it should return the following: MTL 6, LVL 7, DDO 8.


Manager enters ID
Connect to associated server
calls getRecordCount method
gets count from all the server
if operation successful
no
yes
return success message to server/client
generate log

4) Edit record

To test the method that is responsible to edit the record. When invoked by a manager, the server associated with this manager, (determined by the unique *managerID*) searches in the hash map to find the *recordID* and change the value of the field identified by "fieldname" to the *newValue,* if it is found. Upon success or failure it returns a message to the manager and the logs are updated with this information. If the new value of the fields such as location (Teacher), status (Student), does not match the type it is expecting, it is invalid. For example, if the found *Record* is a *TeacherRecord* and the field to change is *location* and *newValue* is other than mtl, lvl or ddo, the server shall return an error. The fields that should be allowed to change are *address, phone and location* (for TeacherRecord), and *course registered, status and status date* (for StudentRecord).


Manager enters ID
Enters Info
Connect to associated server
calls editRecord method
edits the record at the appropriate location of hashmap
no
if operation successful
yes
return success message to server/client
generate log