Concordia University

COMP6231- Distributed System Design

Assigment-2

Distributed Class Management System (DCMS) using Java CORBA

Date: 20th June 2018.

Recipient: Prof. Mohammed Taleb

Student Name: Sagar Vetal (40071979)
Khyatibahen Chaudhary (40071098)
Zankhanaben Ashish Patel (40067635)
Himanshu Kohli (40070839)

Contents

Sr. No.	Topic
1.	Introduction
2.	CORBA
3.	Design Architecture
4.	Class Diagram
5.	Class Description
6.	Data Structure
7.	Operations
8.	UDP
9.	Multi-threading, Synchronization and Concurrency
10.	Activity Logger
11.	Test Cases
12.	Difficulties faced
13.	Conclusion
14.	References

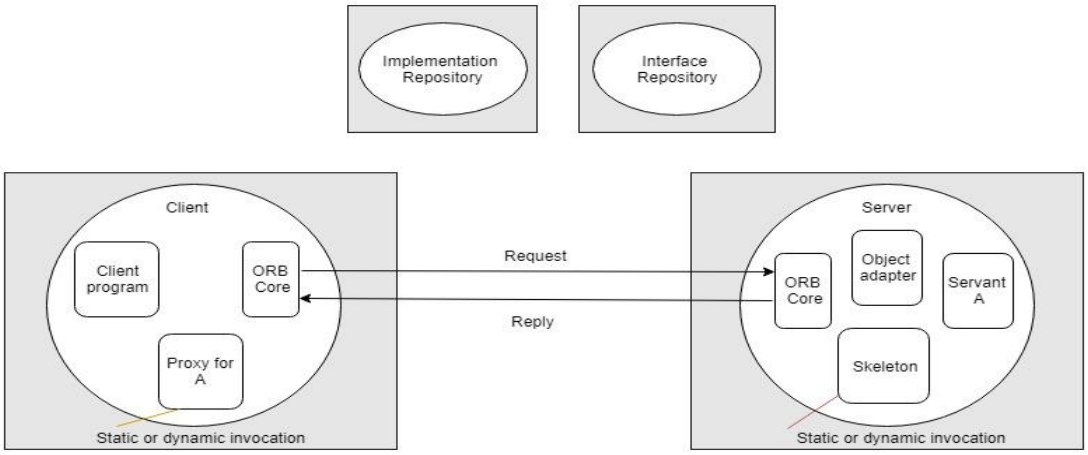
1. Introduction

The main objective is to design and implement a Distributed Class Management System which provides an infrastructure for teachers and student records which are shared across three data centres. Java CORBA offers simplicity to call methods on server machines and returning objects over network and provide network transparency**.** Use of multi-threading with proper synchronization to handle concurrent requests with ease. User Datagram Protocol is used by servers to interact between each other, sending and receiving requests to perform inter-server operations. UDP is highly reliable so as the connection between the server is not easily sabotaged.

2. CORBA

The Common Object Request Broker Architecture is a designed to allow distributed objects to interoperate in a heterogeneous environment, where objects can be implemented in different programming languages and/or deployed on different platforms. CORBA normalizes the method call semantics between application objects residing either in the same address-space or in remote address spaces.

3. Design Architecture



COBRA Architecture

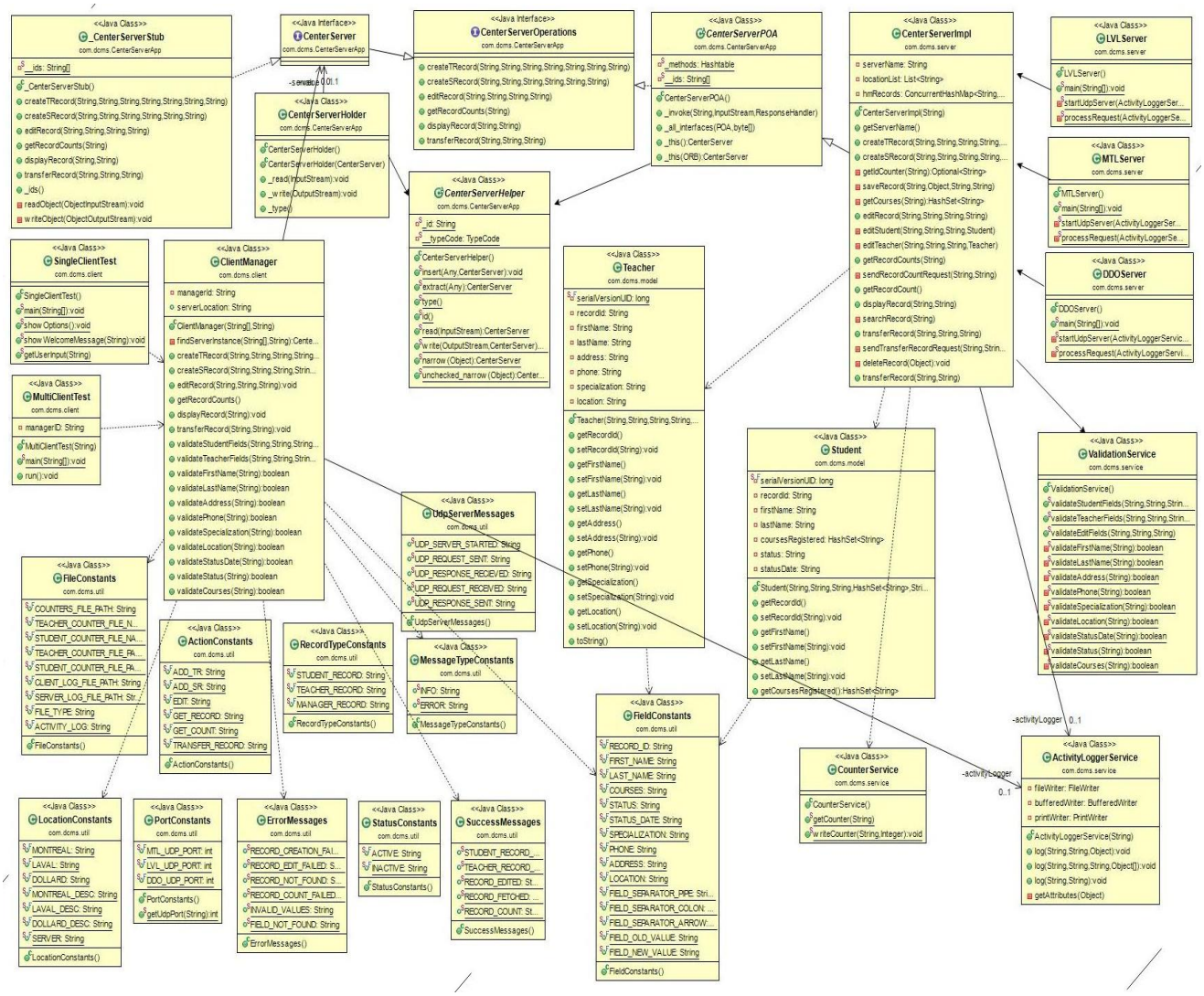
3.1. IDL

An interface definition language is used to describe software component API's. IDL's describe an interface in a language independent way, enabling communication between software components that do not share one language. IDL's are used in remote procedure calls, the machines either end of the link assorted computer using different operating system and computer languages.

```
module CenterServerApp
{
    interface CenterServer
    {
        string createTRecord(in string firstName, in string lastName, in string address, in string phone, in string specialization, in string location, in string managerId);
        string createSRecord(in string firstName, in string lastName, in string coursesRegistered, in string status, in string statusDate, in string managerId);
        string editRecord(in string recordID, in string fieldName, in string newValue, in string managerId);
        string getRecordCounts(in string managerId);
        string displayRecord(in string recordId, in string managerId);
        string transferRecord(in string managerID, in string recordID, in string remoteCenterServerName) ;
    };
};
```

IDL Code snippet

4. Class Diagram



5. Class Description

- **ClientManager.java**
  ClientManager communicate with CenterServer through CORBA. It can perform six different operations: createTRecord, createSRecord, editRecord, getRecordCounts, displayRecord and transferRecord. Clients send multiple requests, which are handled in parallel.

- **CenterServer.java**
  It is an interface which declares below methods that client can invoke on the service.
  1. *createTRecord (firstName, lastName, address, phone, specialization, location, managerId);*
  2. *createSRecord (firstName, lastName, coursesRegistered, status, statusDate, managerId);*
  3. *editRecord (recordId, fieldName, newValue, managerId);*
  4. *getRecordCounts (managerId);*
  5. *displayRecord (recordId, managerId);*
  6. *transferRecord (managerId, recordId, remoteCenterServerName);*

- **CenterServerImpl.java**
  This class implements methods for all operations declared in CenterServer interface. When a client needs a service from the DSMS, it sends a request over the CORBA to the CenterServer. This class will validate, process the request and send reply to client.

- **MTLServer.java, LVLServer.java and DDOServer.java**
  These classes start their respective UDP server and ORB.

- **CounterService.java**
  This class is serializable class used to generate the record id for student and teacher and it will help to keep record id unique among all servers. This class will read latest counter for record from one of below text files,
  StudentCounter.txt, *TeacherCounter.txt* and *RequestCounter.txt*

- **Student.java (Model)**
  This is a java bean class to hold following data of students,
  First Name, Last Name, Courses Registered, Status and Status Date.

- **Teacher.java (Model)**
  This is a java bean class to hold following data of teacher,
  First Name, Last Name, Specialization, Location, Phone and Address.

- **ValidationService.java**
  This class is used to perform following validations,
  1. First name, Last name and specialization fields should include only characters.
  2. Address field can be alphanumeric.
  3. Phone no field should be numeric of length 10.
  4. Location field should have either MTL, LVL or DDO only as a value.
  5. Status field should have either active or inactive as a value.
  6. Status Date format is dd-MM-yyyy.

6. Data Structure

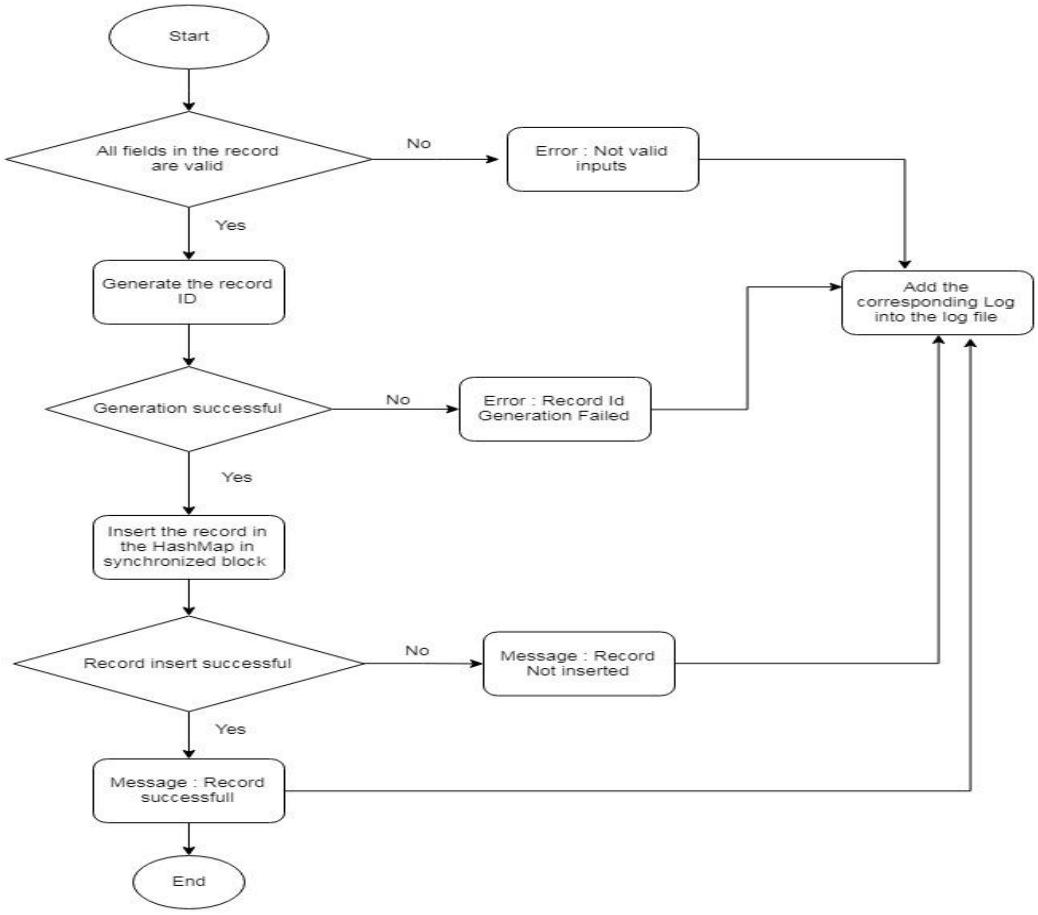
We have used **HashMap** like data structure for the storage of records which allows to perform adding, retrieval and updating operations on both type of records i.e. teacher and student.

Syntax:
HashMap<String, List<Object>> hmRecords = new HashMap<String, List<Object>>();

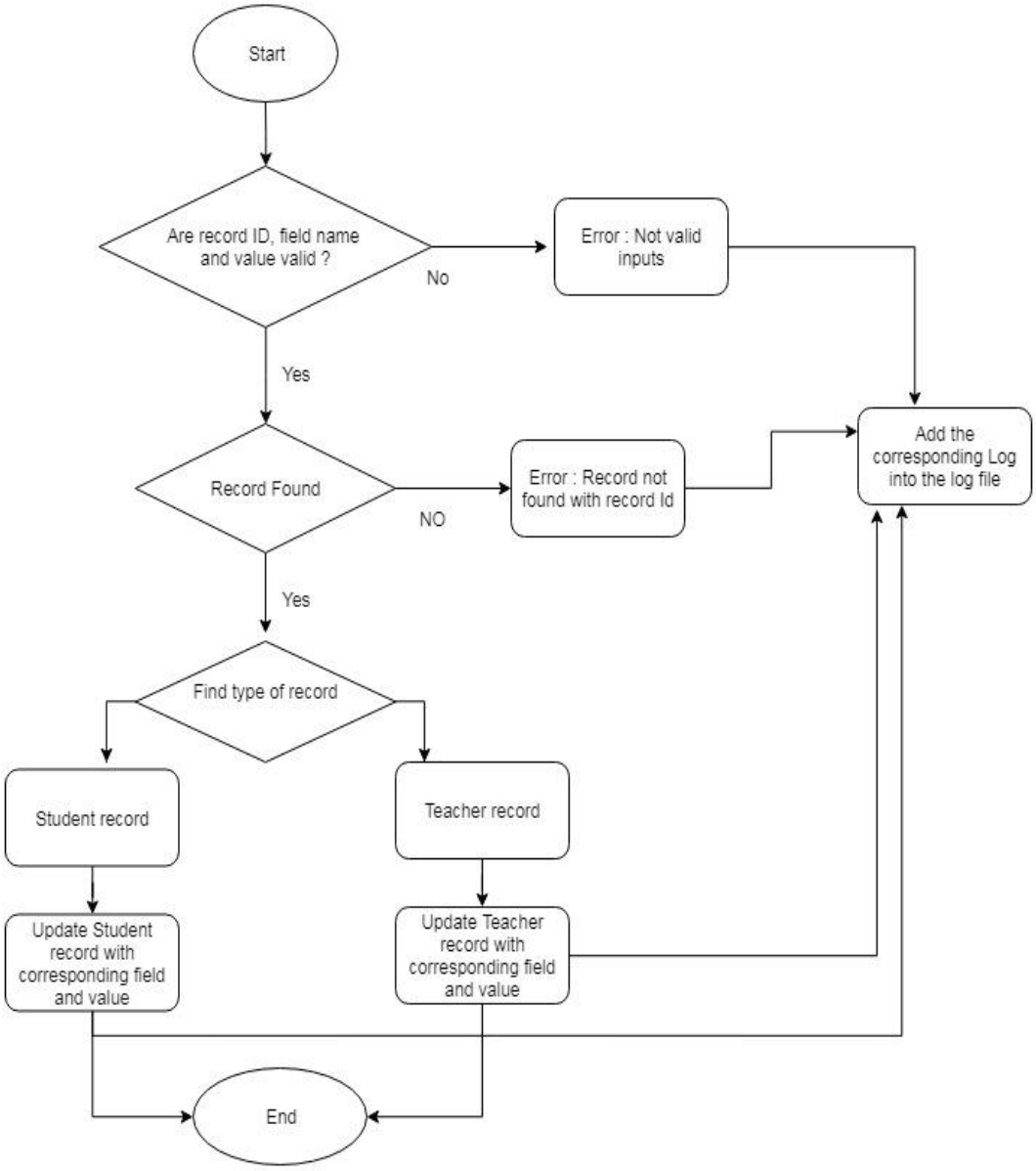
Here, key is first letter of the last name of student/teacher and value is an object which is the parent class which allow to choose object type i.e. Student or Teacher.

7. Operations:

- **Create Record:**
  - While creating new record first all the fields of the record are validated at server side by **ValidationService** class. Validations are as follow,
    1. First name, Last name and specialization fields should include only characters.
    2. Address field can be alphanumeric.
    3. Phone no field should be numeric of length 10.
    4. Location field should have either MTL, LVL or DDO only as a value.
    5. Status field should have either active or inactive as a value.
    6. Status Date format is dd-MM-yyyy.
  - If all fields are valid, record will be stored in hashmap using first letter of the last name of student/teacher as a key, otherwise corresponding error will be returned to client.
  - All success/error messages will be logged into respective log files.

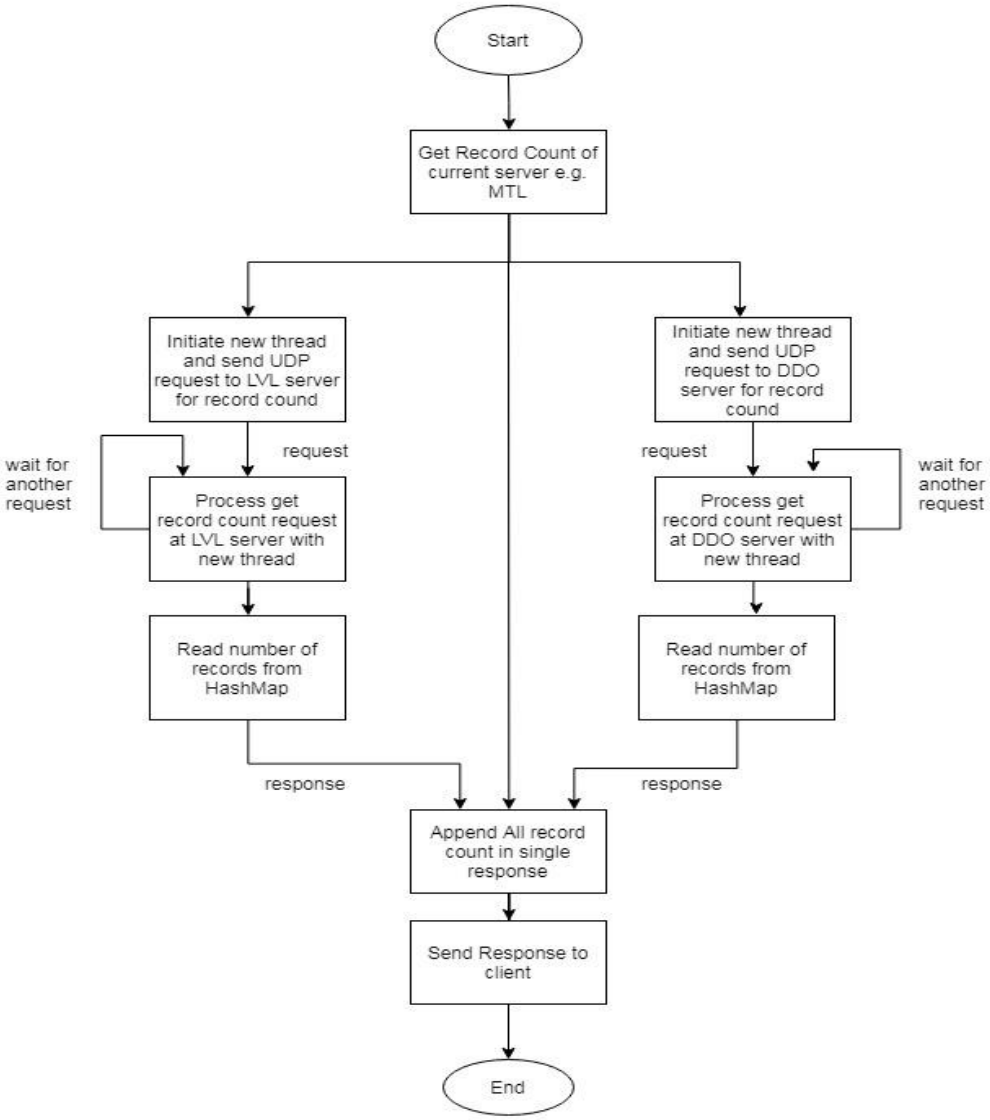


- **Edit Record:**
  - While editing records **ValidationService** class will take care of the validation of allowed fields to be modified and type data allowed for those fields.
  - If all fields are valid, the record will be searched in the hashmap using RecordId.
  - If record is found, entered field will get updated with new value.
  - Otherwise corresponding error will be returned to client.
  - All success/error messages will be logged into respective log files.

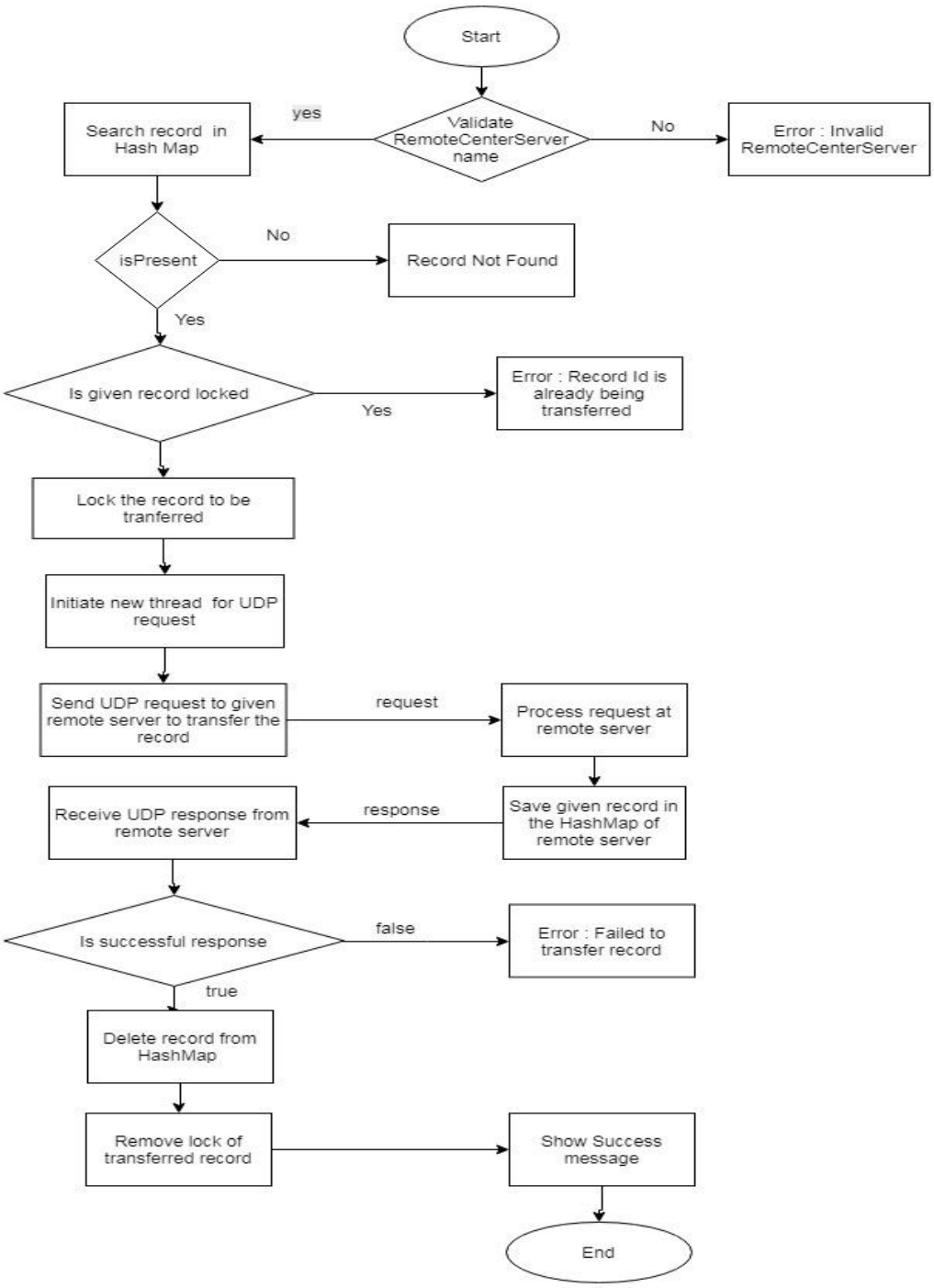


- **Get Record Count:**
  - First current server will fetch the number records from hashmap.
  - Then it will initiate separate thread for each UDP server and send request to fetch record count from each server.
  - Then each server will process get record count request in separate thread and return response to current server.
  - On successful response current server will append all record counts in single response and return it to client.
  - Otherwise corresponding error will be returned to client.

- All success/error messages will be logged into respective log files.



- **Transfer Record:**
  - First remote server name will be validated whether is correct or not. If it is wrong, server will returned error “Invalid remote server”.
  - If it is correct, the record will be searched in the hashmap using RecordId.
  - If record is found, current server will put lock on that record, so that other thread cannot do any operation on that thread. Then server will initiate new thread and send UDP request to remote server and remote server will process that request in separate thread and store that record and return response.
  - On successful response current server will release the lock and delete that record from Hashmap.
  - Otherwise corresponding error will be returned to client.
  - All success/error messages will be logged into respective log files.



8. UDP

We have implemented UDP for communication among three servers (MTL, LVL and DDO servers) to perform operations like getRecordCound() and transferRecord(). The Datagram Packet and Datagram Socket classes in the java.net package implement datagram communication using UDP.

In **getRecordCount()** the current server communicates with other servers using UDP to fetch the number of records available on corresponding server and returns server specific record count.

In **transferRecord()** server checks the HashMap for the required record. If found, the server put lock on that record and transfers the record to specified remote server and after successfully transferring the record to remote server it will remove the record from its storage and then server returns success message to client.

9. Multithreading, Synchronization and Concurrency:

We have implemented multithreading in UDP server communication and multiple client testing. In ***getRecordCounts() and transferRecord()*** methods, a separate thread is initiated to send UDP (getRecordCount and transferRecord respectively) requests to remaining servers. Also, above mentioned requests are processed in separate thread at each UDP server.

We have used the keyword **synchronized** to synchronize thread execution so that only one thread gets the data access and response from server at a time and others wait while execution complete. Synchronization is a procedure which is used to avoid interference with the other threads and memory consistent errors. Moreover, each thread has its own memory pool so there are no shard resources among other threads.

We have used following Synchronization in the project:

- **Block level Synchronization:**
  In the methods like **createSRecord()**, **createTRecord() and editRecord()**, we have made specific block of code synchronized. That code block contains actual insertion/updation into and retrieval from HashMap, and other part of the method don't need synchronization while performing the record addition.
- **Method level Synchronization:**
  Methods like **searchRecord() and getRecordCount()**, contains only retrieval code of HashMap, hence we have entirely made these methods synchronized. Also in order to avoid duplicate record id we have made **getCounter() and writeCounter()** method synchronized in **CounterService** class.

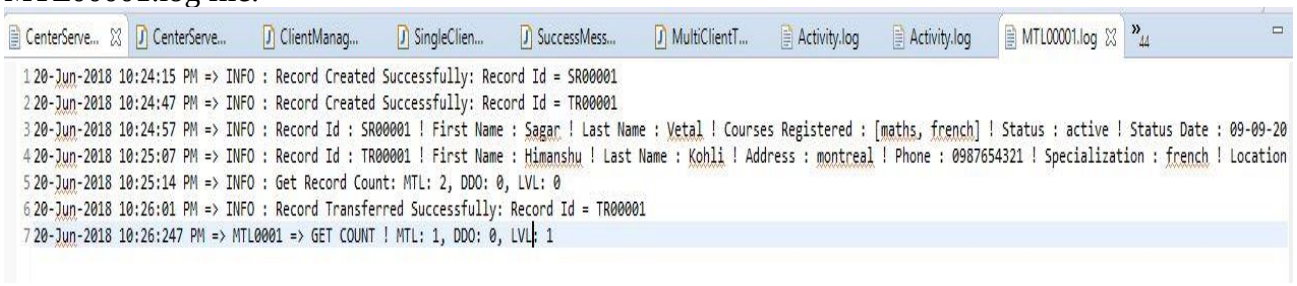
The Fine-grained locking is used to put intrinsic lock while doing parallel access to data resources so that data is accessed as required without infecting the data. This increases the efficiency and maximizes the concurrency to perform concurrent operations.

Concurrency is implemented by using the synchronized () {} block in java which puts temporary lock on the object or list passed to it as parameter in this implementation.

10. Activity Logger

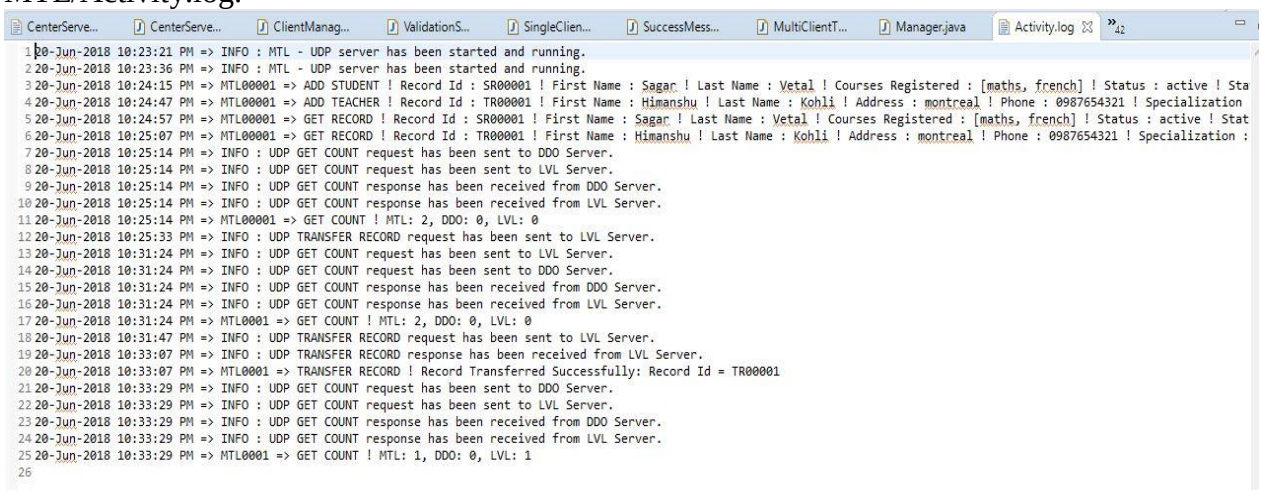
We have created our own customized logger service class named as "**ActivityLoggerService**" which has three overloaded methods to log the success messages and error messages in respective activity log file. There will be log files at both server and client side. One log file named as 'Activity.log' for each server and one log file (using mangerId as a name) for each manager at client side.

MTL00001.log file:



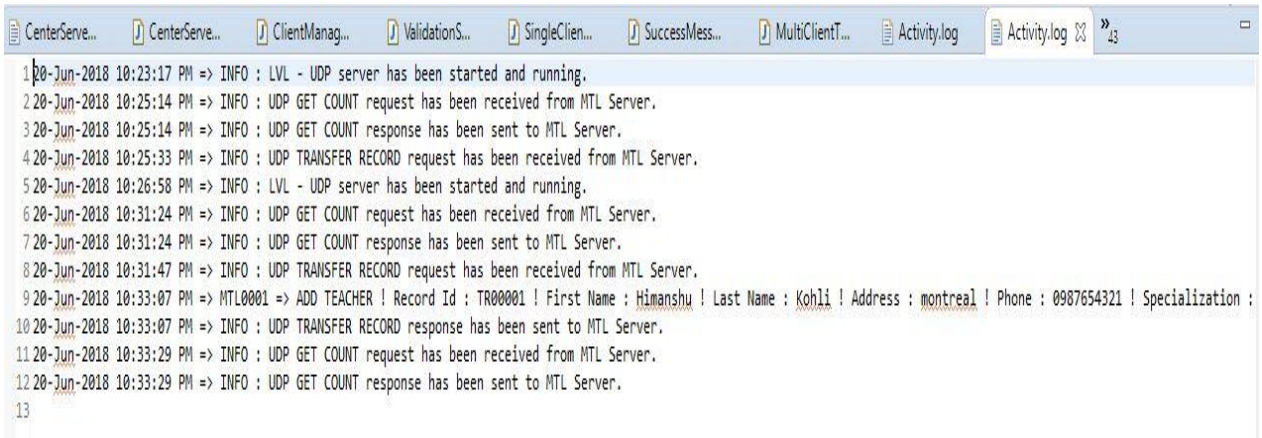
```
20-Jun-2018 10:24:15 PM => INFO : Record Created Successfully: Record Id = SR00001
20-Jun-2018 10:24:47 PM => INFO : Record Created Successfully: Record Id = TR00001
20-Jun-2018 10:24:57 PM => INFO : Record Id : SR00001 ! First Name : Sagar ! Last Name : Vetal ! Courses Registered : [maths, french] ! Status : active ! Status Date : 09-09-20
20-Jun-2018 10:25:07 PM => INFO : Record Id : TR00001 ! First Name : Himanshu ! Last Name : Kohli ! Address : montreal ! Phone : 0987654321 ! Specialization : french ! Location
20-Jun-2018 10:25:14 PM => INFO : Get Record Count: MTL: 2, DDO: 0, LVL: 0
20-Jun-2018 10:26:01 PM => INFO : Record Transferred Successfully: Record Id = TR00001
20-Jun-2018 10:26:247 PM => MTL0001 => GET COUNT ! MTL: 1, DDO: 0, LVL: 1
```

MTL/Activity.log:



```
20-Jun-2018 10:23:21 PM => INFO : MTL - UDP server has been started and running.
20-Jun-2018 10:23:36 PM => INFO : MTL - UDP server has been started and running.
20-Jun-2018 10:24:15 PM => MTL00001 => ADD STUDENT ! Record Id : SR00001 ! First Name : Sagar ! Last Name : Vetal ! Courses Registered : [maths, french] ! Status : active ! Sta
20-Jun-2018 10:24:47 PM => MTL00001 => ADD TEACHER ! Record Id : TR00001 ! First Name : Himanshu ! Last Name : Kohli ! Address : montreal ! Phone : 0987654321 ! Specialization
20-Jun-2018 10:24:57 PM => MTL00001 => GET RECORD ! Record Id : SR00001 ! First Name : Sagar ! Last Name : Vetal ! Courses Registered : [maths, french] ! Status : active ! Stat
20-Jun-2018 10:25:07 PM => MTL00001 => GET RECORD ! Record Id : TR00001 ! First Name : Himanshu ! Last Name : Kohli ! Address : montreal ! Phone : 0987654321 ! Specialization :
20-Jun-2018 10:25:14 PM => INFO : UDP GET COUNT request has been sent to DDO Server.
20-Jun-2018 10:25:14 PM => INFO : UDP GET COUNT request has been sent to LVL Server.
20-Jun-2018 10:25:14 PM => INFO : UDP GET COUNT response has been received from DDO Server.
20-Jun-2018 10:25:14 PM => INFO : UDP GET COUNT response has been received from LVL Server.
20-Jun-2018 10:25:14 PM => MTL00001 => GET COUNT ! MTL: 2, DDO: 0, LVL: 0
20-Jun-2018 10:25:33 PM => INFO : UDP TRANSFER RECORD request has been sent to LVL Server.
20-Jun-2018 10:31:24 PM => INFO : UDP GET COUNT request has been sent to LVL Server.
20-Jun-2018 10:31:24 PM => INFO : UDP GET COUNT request has been sent to DDO Server.
20-Jun-2018 10:31:24 PM => INFO : UDP GET COUNT response has been received from DDO Server.
20-Jun-2018 10:31:24 PM => INFO : UDP GET COUNT response has been received from LVL Server.
20-Jun-2018 10:31:24 PM => MTL0001 => GET COUNT ! MTL: 2, DDO: 0, LVL: 0
20-Jun-2018 10:31:47 PM => INFO : UDP TRANSFER RECORD request has been sent to LVL Server.
20-Jun-2018 10:33:07 PM => INFO : UDP TRANSFER RECORD response has been received from LVL Server.
20-Jun-2018 10:33:07 PM => MTL0001 => TRANSFER RECORD ! Record Transferred Successfully: Record Id = TR00001
20-Jun-2018 10:33:29 PM => INFO : UDP GET COUNT request has been sent to DDO Server.
20-Jun-2018 10:33:29 PM => INFO : UDP GET COUNT request has been sent to LVL Server.
20-Jun-2018 10:33:29 PM => INFO : UDP GET COUNT response has been received from DDO Server.
20-Jun-2018 10:33:29 PM => INFO : UDP GET COUNT response has been received from LVL Server.
20-Jun-2018 10:33:29 PM => MTL0001 => GET COUNT ! MTL: 1, DDO: 0, LVL: 1
```

LVL.Log



```
20-Jun-2018 10:23:17 PM => INFO : LVL - UDP server has been started and running.
20-Jun-2018 10:25:14 PM => INFO : UDP GET COUNT request has been received from MTL Server.
20-Jun-2018 10:25:14 PM => INFO : UDP GET COUNT response has been sent to MTL Server.
20-Jun-2018 10:25:33 PM => INFO : UDP TRANSFER RECORD request has been received from MTL Server.
20-Jun-2018 10:26:58 PM => INFO : LVL - UDP server has been started and running.
20-Jun-2018 10:31:24 PM => INFO : UDP GET COUNT request has been received from MTL Server.
20-Jun-2018 10:31:24 PM => INFO : UDP GET COUNT response has been sent to MTL Server.
20-Jun-2018 10:31:47 PM => INFO : UDP TRANSFER RECORD request has been received from MTL Server.
20-Jun-2018 10:33:07 PM => MTL0001 => ADD TEACHER ! Record Id : TR00001 ! First Name : Himanshu ! Last Name : Kohli ! Address : montreal ! Phone : 0987654321 ! Specialization :
20-Jun-2018 10:33:07 PM => INFO : UDP TRANSFER RECORD response has been sent to MTL Server.
20-Jun-2018 10:33:29 PM => INFO : UDP GET COUNT request has been received from MTL Server.
20-Jun-2018 10:33:29 PM => INFO : UDP GET COUNT response has been sent to MTL Server.
```

11. Test Cases

We have created two separate classes named **SingleClientTest** and **MultiClientTest** to test the functionality with single client and multiple clients respectively. We have created multiple threads in the **MultiClientTest** class for each server to test the functionality. Each thread performs actions on multiple records maintaining concurrency and are synchronized with each other. The threads do not access the same data resources at the same time.

Below excel sheet(***"TestCases.xlsx"***) includes multiple test cases for single client and multiple client test.


TestCases.xlsx

12. Difficulties Faced

1. Implementation of threads for UDP.
2. Implementing synchronization among shared data and avoiding deadlocks
3. UDP implementation of getRecordCount() and transferRecord()

13. Conclusion

The assignment has successfully implemented and demonstrates the use of CORBA to implement Distributed Class Management System with the help of UDP.

14. References

1. https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture
2. https://en.wikipedia.org/wiki/Interface_description_language
3. https://www.cse.wustl.edu/~schmidt/gifs/corba5.gif
4. https://image.slidesharecdn.com/se2ja11-java-networking42-audio-140225050525-phpapp01/95/network-protocols-and-java-programming-28-638.jpg?cb=1393305164
5. http://www.ejbtutorial.com/corba/tutorial-for-corba-hello-world-using-java