Concordia University

COMP6231- Distributed System Design

Assigment-1

Distributed Class Management System (DCMS) using Java RMI

Date: 03 June 2018.

Recipient: Prof. Mohammed Taleb

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

Contents

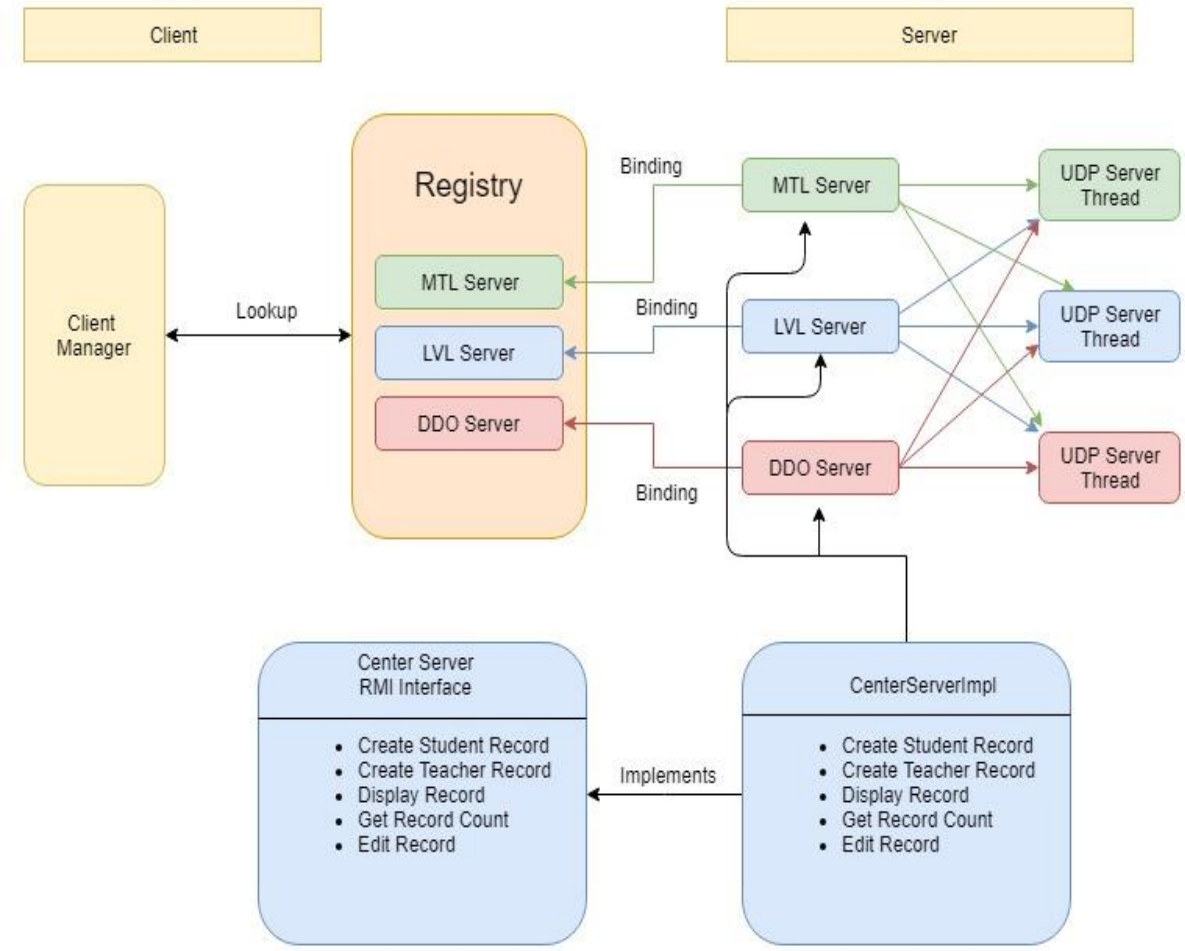
1. Introduction

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.

2. RMI

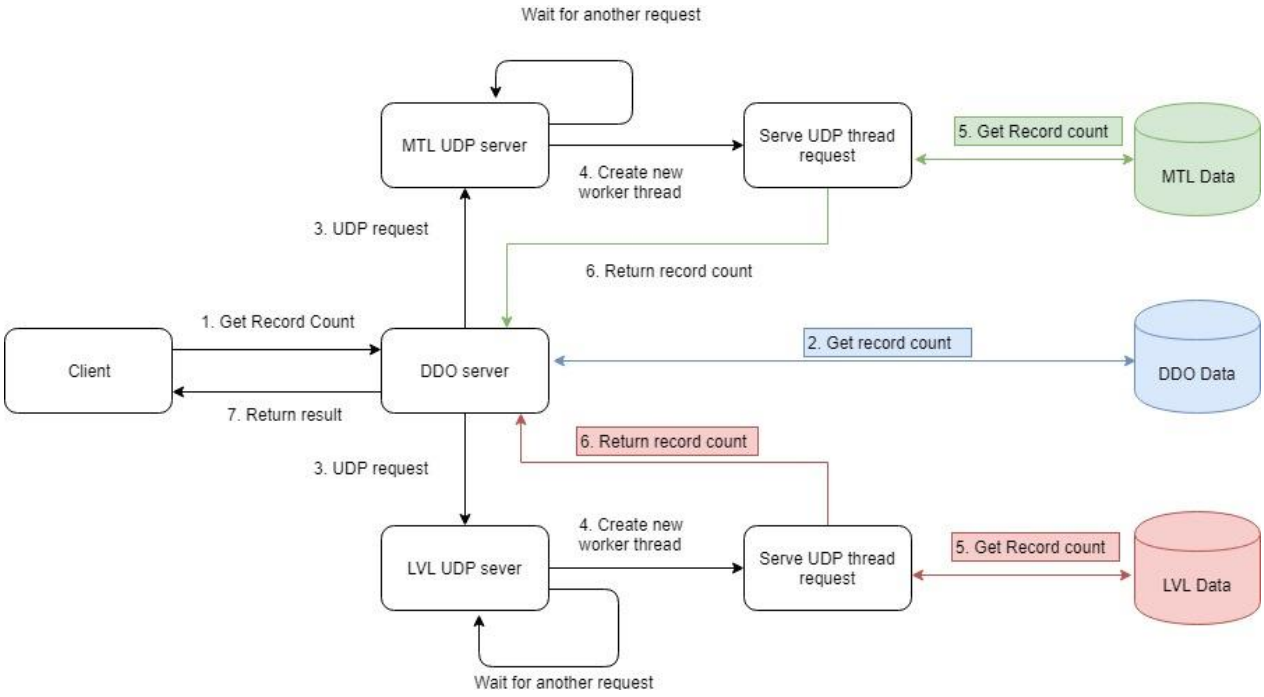
Java RMI offers simplicity to call methods on server machines and returning objects over network and provide network transparency. Use of multi-threading with proper synchronization to handle concurrent requests with ease. User Datagram Protocol is used by servers to interact between each other, sending and receiving requests to perform inter-server operations. UDP is highly reliable so as the connection between the server is not easily sabotaged.

3. Design Architecture



4. UDP Server Design

The communication between the servers is done by UDP to fetch the number of the records generated on each of the servers and display them. The get record requests each server will fork and get the count request thread to remaining servers. Communication among to all three servers take place using UDP/IP protocol.



We have implemented UDP for communication among three servers (MTL, LVL and DDO servers) to perform operation like getRecordCound(). 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.

5. Class Description

- **ClientManager.java**
  ClientManager communicate with CenterServer through RMI. 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);*

- **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 RMI 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 register its name in RMI registry.

- **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

- **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

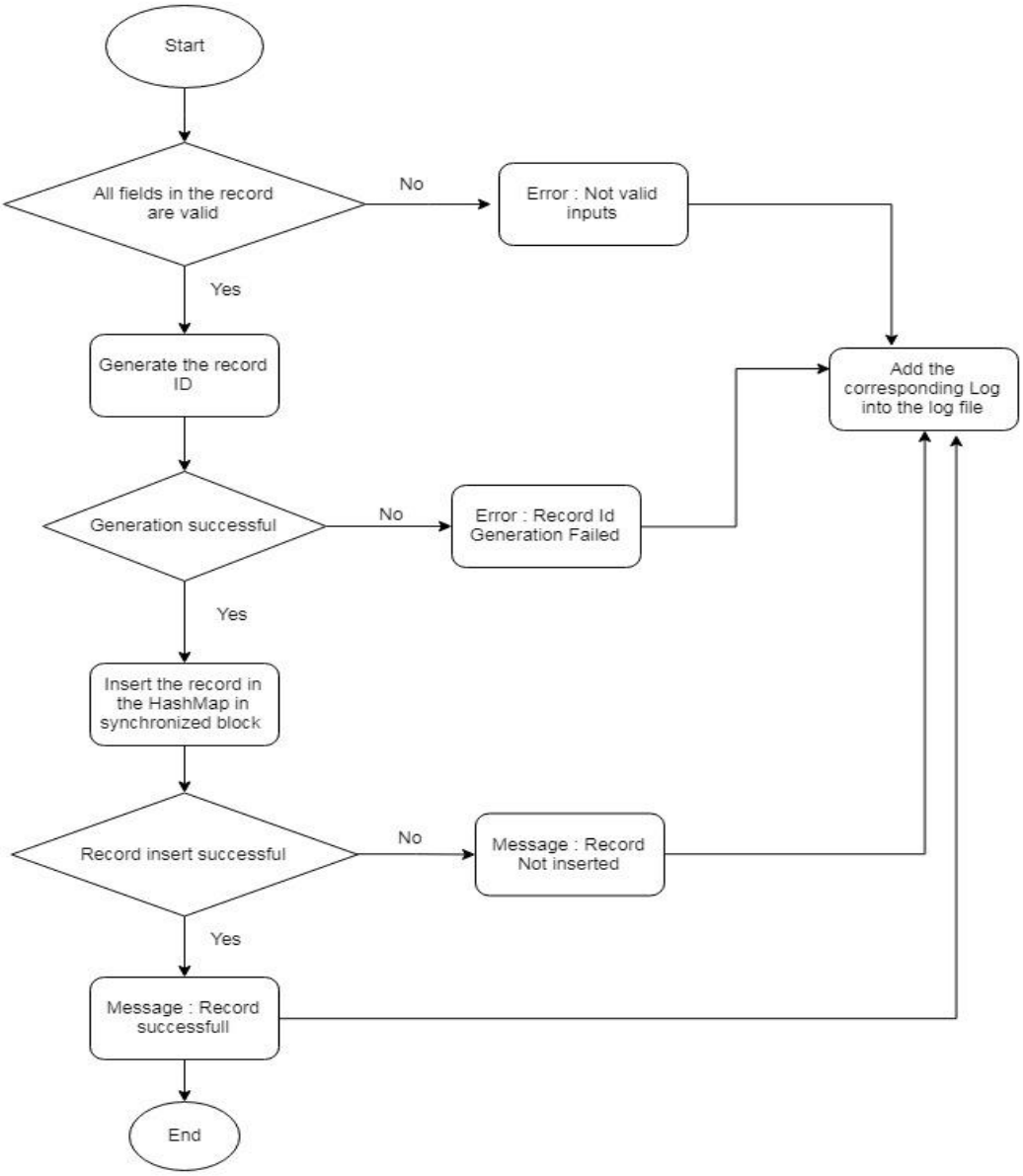
We have used **HashMap** like data structure for the storage of records which allows to perform adding, retrieval and updating operations on both type of records i.e. teacher and student.

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

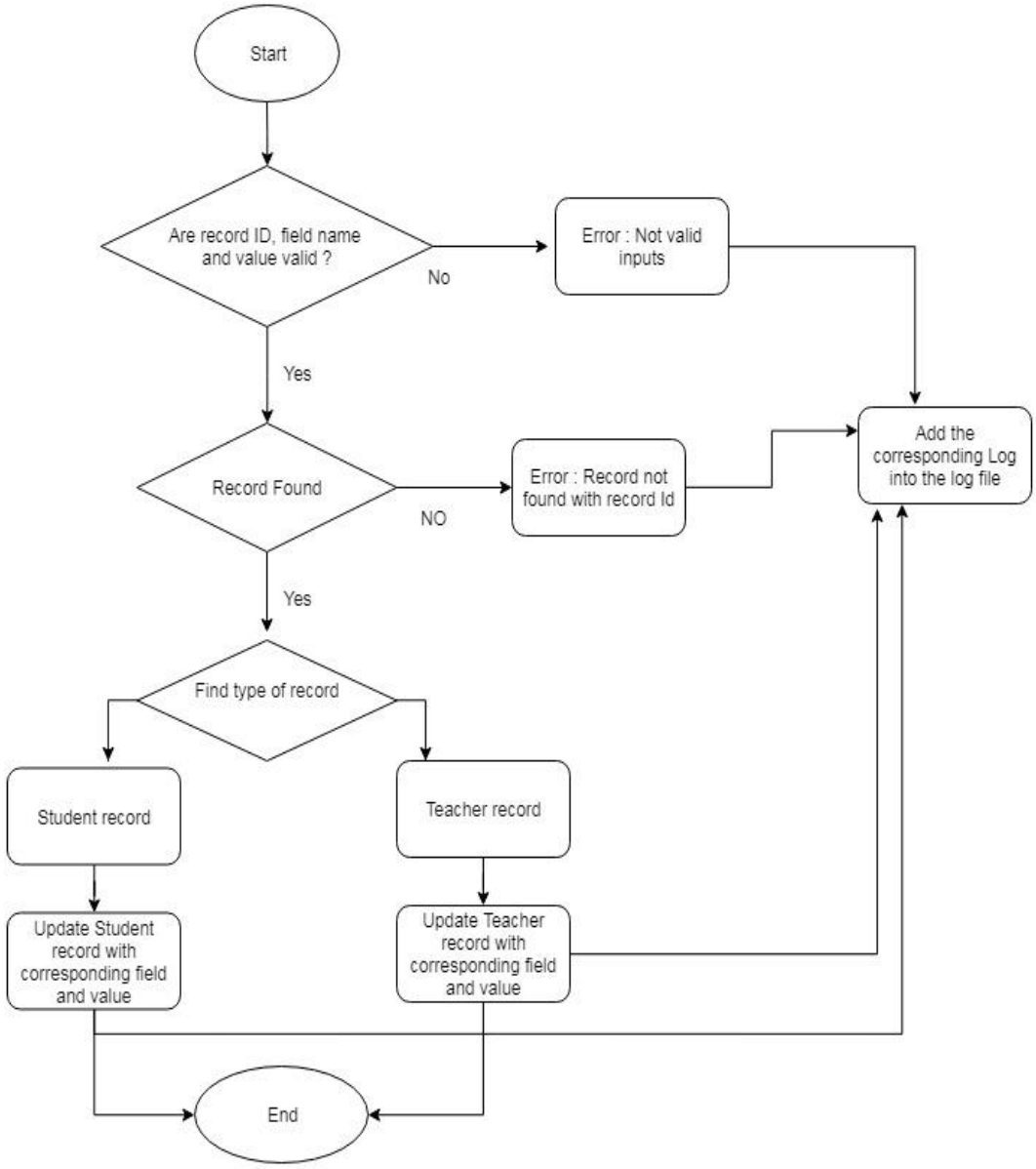
Here, key is first letter of the last name of student/teacher and value is an object which is the parent class which allow to choose object type i.e. Student or Teacher.

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.

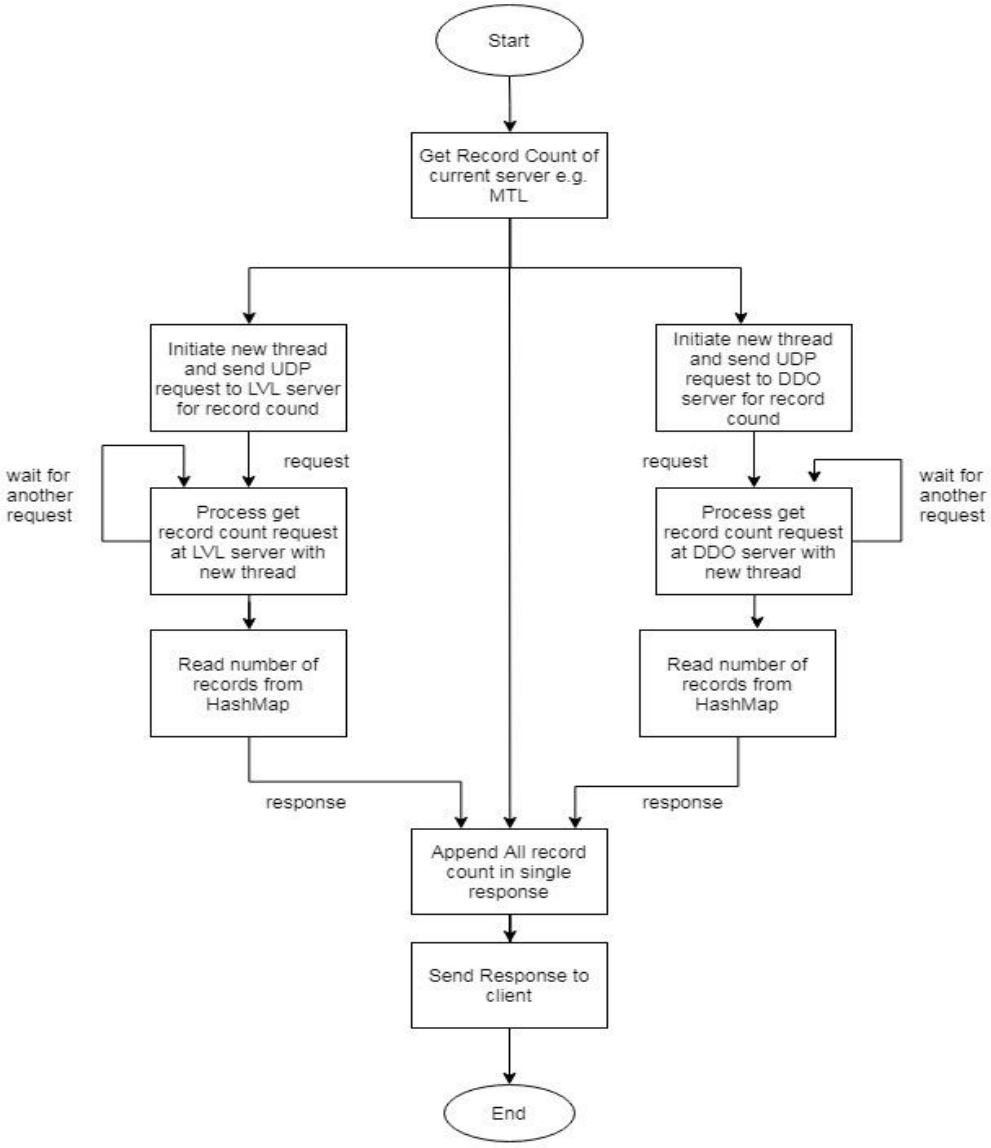


- **Edit Record:**
  - While editing records **ValidationService** class will take care of the validation of allowed fields to be modified and type data allowed for those fields.
  - If all fields are valid, the record will be searched in the hashmap using RecordId.
  - If record is found, entered field 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.



8. Multithreading, Synchronization and Concurrency:

We have implemented multithreading in UDP server communication and multiple client testing. In ***getRecordCounts()*** method, a separate thread is initiated to send UDP (getRecordCount) 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.

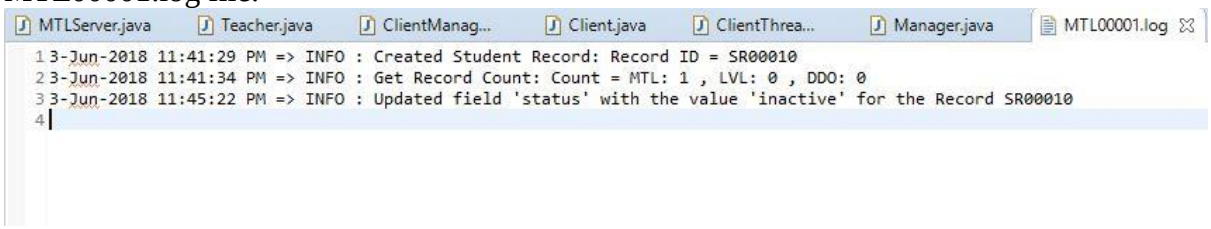
The Fine-grained locking is used to put intrinsic lock while doing parallel access to data resources so that data is accessed as required without infecting the data. This increases the efficiency and maximizes the concurrency to perform concurrent operations.

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.

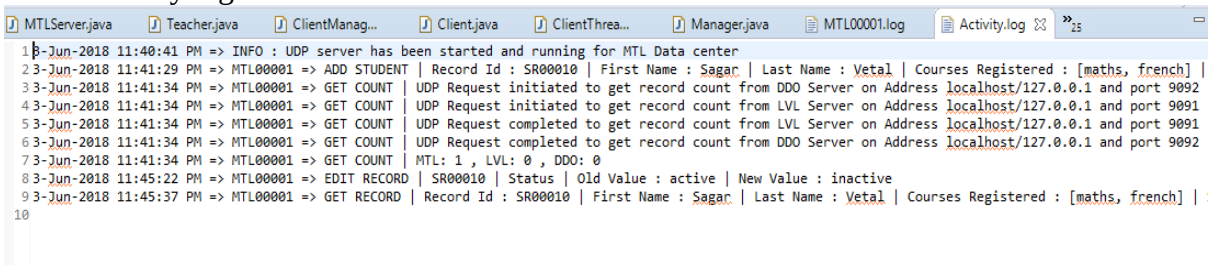
9. Activity Logger

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:



MTL/Activity.log:



10. 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

11. Difficulties Faced:

- Java RMI implementation
- Concurrency with multithreaded system
- Data modelling
- Fine grained locking and synchronization
- Java Reflection

12. Conclusion

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

13. References:

1. http://www.ejbtutorial.com/java-rmi/new-easy-tutorial-for-java-rmi-using-eclipse
2. https://www.mkyong.com/java/java-rmi-distributed-objects-example
3. https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/
4. https://stackoverflow.com/questions/2836646/javaserializableobjecttobytearray
5. https://moodle.concordia.ca/moodle/pluginfile.php/3157534/mod_resource/content/2/Tutorial_3_JAVA%20RMI2.pdf