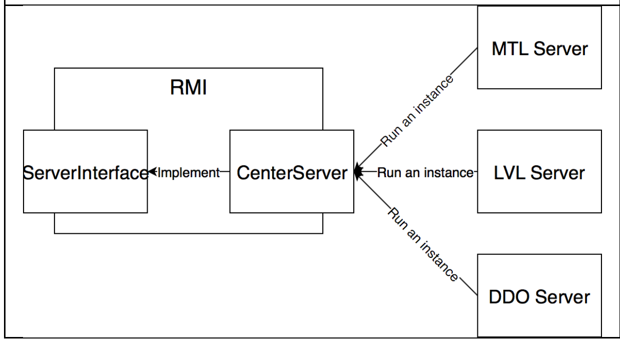
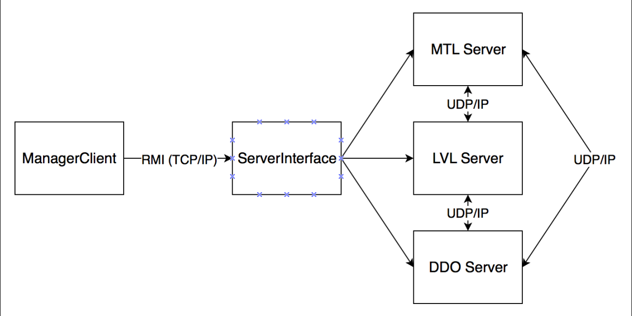
Assignment 01
Name: Quoc Minh Vu
ID: 40040500

The project is designed as follow

Servers model	**Communications model**
	

Why using UDP for server-to-server communication?

The communication between servers to request the number of records on each server uses UDP datagram because of some reasons

1. The communication is just requesting and replying, once, no connection needs to be maintained between any pair of servers.
2. A server may need to communicate with an unknown number of other servers. By using UDP, the server needs only one socket to send to and receive from many other servers.

ServerInterface.java

```
public interface ServerInterface extends Remote
```

- This is an RMI interface

```
String createTRecord(String firstName, String lastName, String address, String phone,
String specialization, String location) throws RemoteException,
ServerNotActiveException;
```

- createTRecord() function is a remote function invoked by clients to create new teacher record. It returns the new record's ID if successful, otherwise return a blank string

```
String createSRecord(String firstName, String lastName, String coursesRegistered, String
status) throws RemoteException, ServerNotActiveException;
```

- createSRecord() function is a remote function invoked by clients to create new student record. It returns the new record's ID if successful, otherwise return a blank string

```
String getRecordCounts() throws RemoteException;
```

- getRecordCounts() function is a remote function invoked by clients to get the current number of records in each and every server of the system. It returns a string containing the acronym of the servers followed by the number of records in that server. In this function, the invoked server shall send request to all other servers to get the number of records in those servers

```
boolean editRecord(String recordID, String fieldName, String newValue) throws
RemoteException;
```

- editRecord() function is a remote function invoked by clients to edit a record. It return true if successful, otherwise returns false

```
String printRecords() throws RemoteException;
```

- printRecords() is a helper, remote function returning a string that contains the content of each and every record in a server

```java
Record locateRecord(String recordID) throws RemoteException;
```

- locateRecord() function is a helper, remote function returning an object of a record given its recordID. If no record is found, it shall return null

CenterServer.java

```java
public class CenterServer extends UnicastRemoteObject implements ServerInterface
```

- This class implement the interface defined in ServerInterface.java
- This class extends UnicastRemoteObject because RMI uses TCP/IP protocol which only support unicast stream

```java
private Map<Character, ArrayList<Record>> recordsMap;
private int recordID;
private final Object LOCK = new Object();
private Server_ID serverID;
private int recordsCount;
private int rmiPort;
private int udpPort;
private static final Logger LOGGER = Logger.getLogger(CenterServer.class.getName());
```

- recordsMap is a HashMap acting like a server database, mapping records' lastname initial characters to an ArrayList storing record objects
- recordID is an integer variable helping to generate the last 5 digits in a record ID
- LOCK is an object to help synchronizing a block of code in case we do not want to lock another object or to lock a variable
- serverID is a value representing an unique ID of a server
- recordsCount is an integer variable counting the total number of records in a server. It increase by one each time a new record is added
- rmiPort variable stores the port number used to communicate with clients using RMI
- udpPort variable stores the port number used to communicate with other servers using UDP/IP protocol
- LOGGER is used to write log

XXXServer.java

```java
public class MTLServer
```

- This class has only one main() function which creates an instance of the CenterServer.java class, register a RMI port in the registry, bind the server instance to its name in the registry and finally, let the server instance starts a socket to listen to UDP requests

Record.java

```java
public class Record implements Serializable
```

- This class is the parent class of both kinds of record which are Teacher Record and Student Record
- It implements Serializable so it can be marshalled/unmarshalled by RMI to transmit between clients and servers

```java
public enum Record_Type {TEACHER, STUDENT}
private String recordID;
private String firstName;
private String lastName;
private Record_Type recordType;
```

- This class has all mutual properties among Teacher record and Student record
- This class has a public enum Record_Type defining two kinds of record, along with a recordType variable to store the actual kind of the instance of the record
- recordID is a string storing the record's ID

- firstName is a string storing record’s first name
- lastName is a string storing record’s last name

TeacherRecord.java

```
public class TeacherRecord extends Record
```

- This class is a subclass of the Record class

```
public enum Mutable_Fields {address, phone, location}
private String address;
private String phone;
private String specialization;
private String location;
```

- It has a public enum to declare all fields that are editable by clients
- All other properties have kind of string, storing a teacher record’s information

StudentRecord.java

```
public class StudentRecord extends Record
```

- This class is a subclass of the Record class

```
public enum Mutable_Fields {coursesRegistered, status, statusDate}
private String coursesRegistered;
private String status;
private Date statusDate;
```

- It has a public enum to declare all fiends that are editable by clients
- It has other fields to store a student record’s information like coursesRegistered, status and statusDate

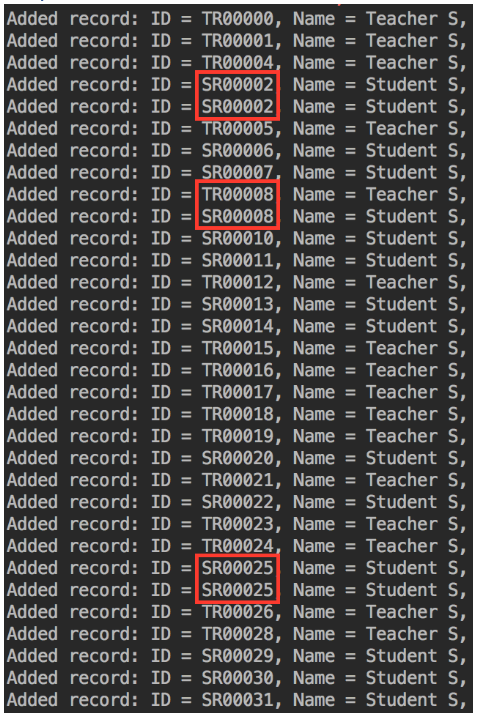
ManagerClient.java

```
public class ManagerClient implements Runnable
```

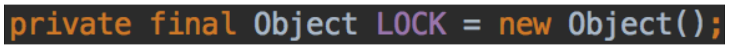
- This class is implemented to test the functionalities of the RMI distributed system
- It implements Runnable so it can create multiple instances as threads (each thread correspond to a manager client), simulating a real life situation where the system is used by multiple clients concurrently

Explaining multi-threading and synchronizing implementation

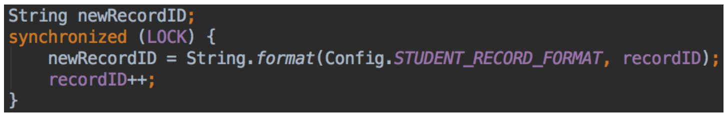
- **Problem**: Without locking new records' ID generating, it is possible to have records having same IDs when many threads trying to create new records and adding them into the same ArrayList, on the same server, concurrently
- **Test case**: running 32 threads trying to create 32 new records having the same LastName's initial character, on the same server, start them concurrently
- **Result**: records having the same ID
- **Solution**: synchronize the code where the program gets the current record ID and generates a new one



```
Added record: ID = TR00000, Name = Teacher S,
Added record: ID = TR00001, Name = Teacher S,
Added record: ID = TR00004, Name = Teacher S,
Added record: ID = SR00002, Name = Student S,
Added record: ID = SR00002, Name = Student S,
Added record: ID = TR00005, Name = Teacher S,
Added record: ID = SR00006, Name = Student S,
Added record: ID = SR00007, Name = Student S,
Added record: ID = TR00008, Name = Teacher S,
Added record: ID = SR00008, Name = Student S,
Added record: ID = SR00010, Name = Student S,
Added record: ID = SR00011, Name = Student S,
Added record: ID = TR00012, Name = Teacher S,
Added record: ID = SR00013, Name = Student S,
Added record: ID = SR00014, Name = Student S,
Added record: ID = TR00015, Name = Teacher S,
Added record: ID = TR00016, Name = Teacher S,
Added record: ID = TR00017, Name = Teacher S,
Added record: ID = TR00018, Name = Teacher S,
Added record: ID = TR00019, Name = Teacher S,
Added record: ID = SR00020, Name = Student S,
Added record: ID = TR00021, Name = Teacher S,
Added record: ID = SR00022, Name = Student S,
Added record: ID = TR00023, Name = Teacher S,
Added record: ID = TR00024, Name = Teacher S,
Added record: ID = SR00025, Name = Student S,
Added record: ID = SR00025, Name = Student S,
Added record: ID = TR00026, Name = Teacher S,
Added record: ID = TR00028, Name = Teacher S,
Added record: ID = SR00029, Name = Student S,
Added record: ID = SR00030, Name = Student S,
Added record: ID = SR00031, Name = Student S,
```

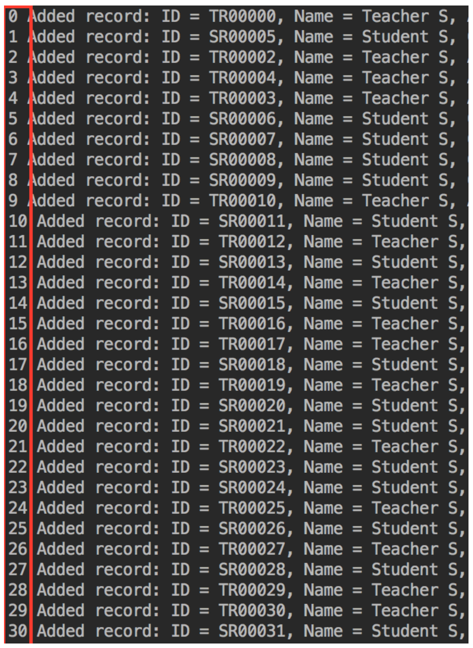


```java
private final Object LOCK = new Object();
```

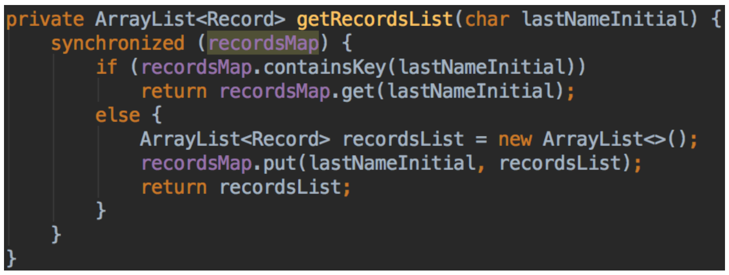


```java
String newRecordID;
synchronized (LOCK) {
    newRecordID = String.format(Config.STUDENT_RECORD_FORMAT, recordID);
    recordID++;
}
```

- **Problem**: When multiple threads trying to put new ArrayList into the same HashMap with the same key concurrently (a lot of new records having the same LastName's initial character), it is possible to lose some records
- **Test case**: running 32 threads trying to create 32 new records having the same LastName's initial character, on the same server, start them concurrently
- **Result**: sometimes, the final result could lose one or two records
- **Solution**: lock the HashMap before checking for the existence of an ArrayList corresponding to a lastname's initial character. Release the lock when finish creating or getting the ArrayList.

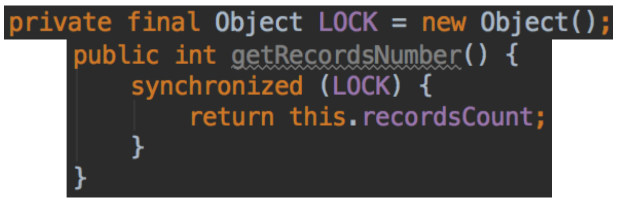


```
Added record: ID = TR00000, Name = Teacher S,
Added record: ID = SR00005, Name = Student S,
Added record: ID = TR00002, Name = Teacher S,
Added record: ID = TR00004, Name = Teacher S,
Added record: ID = TR00003, Name = Teacher S,
Added record: ID = SR00006, Name = Student S,
Added record: ID = SR00007, Name = Student S,
Added record: ID = SR00008, Name = Student S,
Added record: ID = SR00009, Name = Student S,
Added record: ID = TR00010, Name = Teacher S,
Added record: ID = SR00011, Name = Student S,
Added record: ID = TR00012, Name = Teacher S,
Added record: ID = SR00013, Name = Student S,
Added record: ID = TR00014, Name = Teacher S,
Added record: ID = SR00015, Name = Student S,
Added record: ID = TR00016, Name = Teacher S,
Added record: ID = TR00017, Name = Teacher S,
Added record: ID = SR00018, Name = Student S,
Added record: ID = TR00019, Name = Teacher S,
Added record: ID = SR00020, Name = Student S,
Added record: ID = SR00021, Name = Student S,
Added record: ID = TR00022, Name = Teacher S,
Added record: ID = SR00023, Name = Student S,
Added record: ID = SR00024, Name = Student S,
Added record: ID = TR00025, Name = Teacher S,
Added record: ID = SR00026, Name = Student S,
Added record: ID = TR00027, Name = Teacher S,
Added record: ID = SR00028, Name = Student S,
Added record: ID = TR00029, Name = Teacher S,
Added record: ID = TR00030, Name = Teacher S,
Added record: ID = SR00031, Name = Student S,
```



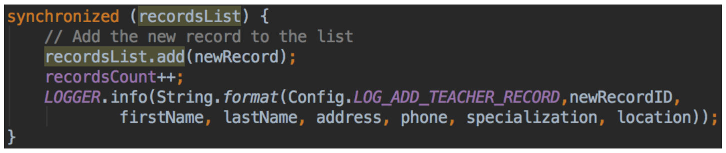
```java
private ArrayList<Record> getRecordsList(char lastNameInitial) {
    synchronized (recordsMap) {
        if (recordsMap.containsKey(lastNameInitial))
            return recordsMap.get(lastNameInitial);
        else {
            ArrayList<Record> recordsList = new ArrayList<>();
            recordsMap.put(lastNameInitial, recordsList);
            return recordsList;
        }
    }
}
```

This function returns the current number of records of the server. It can be invoked concurrently when having multiple servers request at the same time. Synchronization makes sure that only one thread can access the shared variable at a time.



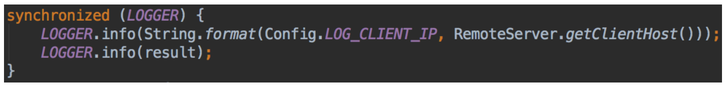
```java
private final Object LOCK = new Object();
public int getRecordsNumber() {
    synchronized (LOCK) {
        return this.recordsCount;
    }
}
```

Synchronization when adding a new record to an ArrayList prevents unpredictable behaviors of the ArrayList. Record counting variable increment is in the block to make sure this variable always tell the correct number of records of the server. Writing log is in the same block ensures log content is updated correctly and reliable.



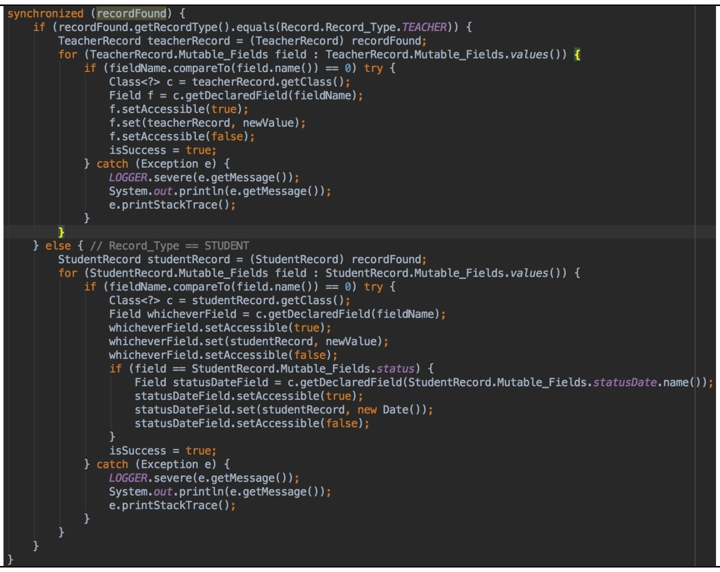
```java
synchronized (recordsList) {
    // Add the new record to the list
    recordsList.add(newRecord);
    recordsCount++;
    LOGGER.info(String.format(Config.LOG_ADD_TEACHER_RECORD,newRecordID,
            firstName, lastName, address, phone, specialization, location));
}
```

When servers need to write several logs regarding the same activities, the Logger should be synchronized so that log content related to each other can be written without disturbance.

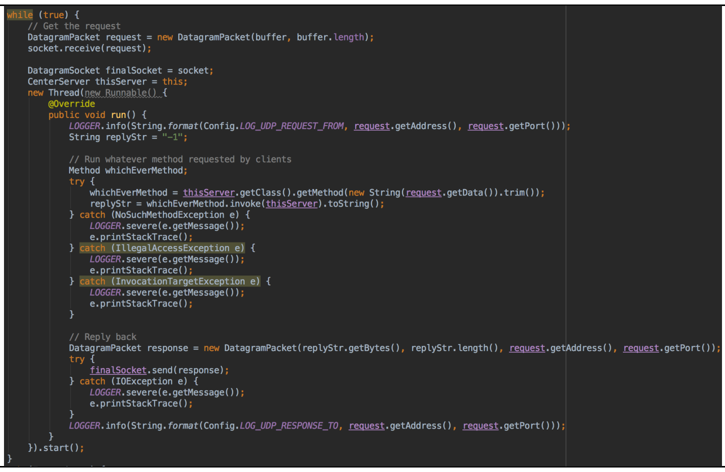


```java
synchronized (LOGGER) {
    LOGGER.info(String.format(Config.LOG_CLIENT_IP, RemoteServer.getClientHost()));
    LOGGER.info(result);
}
```

When edit a record, lock that particular record so that only one thread can edit it at a time



```java
synchronized (recordFound) {
    if (recordFound.getRecordType().equals(Record.Record_Type.TEACHER)) {
        TeacherRecord teacherRecord = (TeacherRecord) recordFound;
        for (TeacherRecord.Mutable_Fields field : TeacherRecord.Mutable_Fields.values()) {
            if (fieldName.compareTo(field.name()) == 0) try {
                Class<?> c = teacherRecord.getClass();
                Field f = c.getDeclaredField(fieldName);
                f.setAccessible(true);
                f.set(teacherRecord, newValue);
                f.setAccessible(false);
                isSuccess = true;
            } catch (Exception e) {
                LOGGER.severe(e.getMessage());
                System.out.println(e.getMessage());
                e.printStackTrace();
            }
        }
    } else { // Record_Type == STUDENT
        StudentRecord studentRecord = (StudentRecord) recordFound;
        for (StudentRecord.Mutable_Fields field : StudentRecord.Mutable_Fields.values()) {
            if (fieldName.compareTo(field.name()) == 0) try {
                Class<?> c = studentRecord.getClass();
                Field whicheverField = c.getDeclaredField(fieldName);
                whicheverField.setAccessible(true);
                whicheverField.set(studentRecord, newValue);
                whicheverField.setAccessible(false);
                if (field == StudentRecord.Mutable_Fields.status) {
                    Field statusDateField = c.getDeclaredField(StudentRecord.Mutable_Fields.statusDate.name());
                    statusDateField.setAccessible(true);
                    statusDateField.set(studentRecord, new Date());
                    statusDateField.setAccessible(false);
                }
                isSuccess = true;
            } catch (Exception e) {
                LOGGER.severe(e.getMessage());
                System.out.println(e.getMessage());
                e.printStackTrace();
            }
        }
    }
}
```

When getting a request via UDP from another server, the active server starts a new thread to handle that request. This implementation lets the server being able to handle multiple request concurrently, quickly come back to listening to avoid missing other requests.	

```java
while (true) {
    // Get the request
    DatagramPacket request = new DatagramPacket(buffer, buffer.length);
    socket.receive(request);

    DatagramSocket finalSocket = socket;
    CenterServer thisServer = this;
    new Thread(new Runnable() {
        @Override
        public void run() {
            LOGGER.info(String.format(Config.LOG_UDP_REQUEST_FROM, request.getAddress(), request.getPort()));
            String replyStr = "-1";

            // Run whatever method requested by clients
            Method whichEverMethod;
            try {
                whichEverMethod = thisServer.getClass().getMethod(new String(request.getData()).trim());
                replyStr = whichEverMethod.invoke(thisServer).toString();
            } catch (NoSuchMethodException e) {
                LOGGER.severe(e.getMessage());
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                LOGGER.severe(e.getMessage());
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                LOGGER.severe(e.getMessage());
                e.printStackTrace();
            }

            // Reply back
            DatagramPacket response = new DatagramPacket(replyStr.getBytes(), replyStr.length(), request.getAddress(), request.getPort());
            try {
                finalSocket.send(response);
            } catch (IOException e) {
                LOGGER.severe(e.getMessage());
                e.printStackTrace();
            }
            LOGGER.info(String.format(Config.LOG_UDP_RESPONSE_TO, request.getAddress(), request.getPort()));
        }
    }).start();
}
```