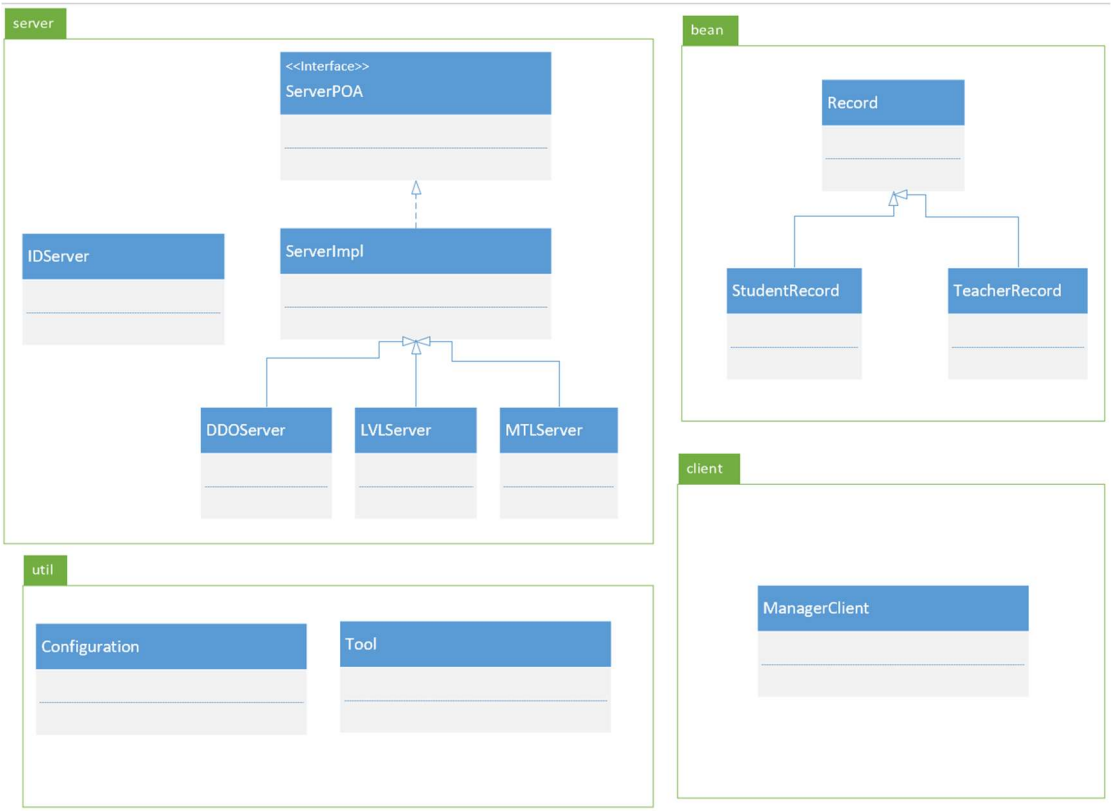
COMP 6231 Assignment 2 Report

Group Member:
Zexin Peng
Student number: 40166520

1. Architecture Design:

DDOServer, LVLServer and MTLServer are inherited from the same superclass “ServerImpl”, which implements the Interface “ServerPOA”. IDServer is responsible for assigning RecordID when creating new records. We should indicate the managerID and location of the client at first. The responsibility of “Configuration” is to store some configuration information like ports and directory names. “Tool” provides basic support to other classes.



2. Techniques:

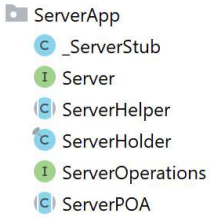
2.1 CORBA:

We need to define the interface in IDL language at first.

```
module ServerApp
{
    interface Server
    {
        string createTRecord(in string firstName, in string lastName, in string address, in string phone, in string

        string createSRecord(in string firstName, in string lastName, in string courseRegistered, in string status,
        /*
        if MTL has 6 records, LVL has 7 and DDO had 8, it should return the following: MTL 6, LVL 7, DDO 8.
         */
        string getRecordCounts(in string managerID);
        string editRecord(in string recordID, in string fieldName, in string newValue, in string managerID);
        string transferRecord(in string managerID, in string recordID, in string remoteCenterServerName);
    };
};
```

And then, we need to compile it and generate the according JAVA interface codes.



2.2 UDP/TCP Socket:

I use UDP socket to get the quantity of records in other servers.

The recordID should be unique in the system, so I create a new server named “IDServer” to assign recordID, and it uses TCP to communicate with other servers. Another usage of TCP programming in my assignment is in method transferRecord().

2.3 synchronized:

I use ‘synchronized’ in my code to avoid synchronization problem, because some shared data cannot be modified concurrently. By the way, to maximize concurrency, I do not maintain strong consistency in my assignment. In other word, if a server has 10 records, and a client will get 10 and will not be blocked when it wants to know the quantity of records in this server even if another client is creating new records in this server at the same time. If a system has strong consistency, the client will be blocked and get 11 after another client creates a new record, but it will reduce the concurrency.

3. Data Structure:

I encapsulate student records and teacher records as Bean classes with bunches of getter and setter method.

3.1 StudentRecord:

```
// maths/french/science, note that student could be registered for multiple courses
String[] coursesRegistered;
// active/inactive
String status;
// date when student became active (if status is active) or date when student became in
String statusDate;

public StudentRecord(String recordID, String firstName, String lastName, String[] cours
    this.recordID = recordID;
    this.firstName = firstName;
    this.lastName = lastName;
    this.coursesRegistered = coursesRegistered;
    this.status = status;
    this.statusDate = statusDate;
}
```

3.2 TeacherRecord:

```
private String address;
private String phone;
// french, maths, etc
private String specialization;
// mtl, lvl, ddo
private Location location;

public TeacherRecord(String recordID, String firstName, String lastName, String address, String phone,
    this.recordID = recordID;
    this.firstName = firstName;
    this.lastName = lastName;
    this.address = address;
    this.phone = phone;
    this.specialization = specialization;
    this.location = location;
}
```

3.3 RecordMap:

All records are stored in the RecordMap, whose type definition is HashMap<Character, List<Record>>. The key of this map is the first letter of the last name indicated in the records.

```
private static HashMap<Character, List<Record>> recordMap = new HashMap<>();
```

4. Test scenarios

Each server initially has a few records except scenario 4.1. We have two records in MTL Server, three records in LVL Server and four records in DDO Server. The correctness of methods createTeacherRecord(), createStudentRecord(), getRecordCounts() and editRecord() are already tested in the report of assignment 1, so I focus on concurrency of my system in this report. All codes can be found in package client named TestScenario1, TestScenario2 and so on.

4.1 Scenario 1:

Description: I start three threads representing three clients “LVL0001”, “DDO0001” and “MTL0001” respectively and create 300 student records and 300 teacher records concurrently to test the uniqueness of recordID and correctness of createRecords. You can see the code in client.TestScenario1.java.

Firstly, I create three threads and run them.

```
public static void main(String[] args) {
    Thread LVLThread = new Thread(new LVLThread());
    Thread MTLThread = new Thread(new MTLThread());
    Thread DDOThread = new Thread(new DDOThread());
    LVLThread.start();
    MTLThread.start();
    DDOThread.start();
}
```

In the IDServer, I got logs below.

```
reply to /127.0.0.1:65250 with teacherNum 297
reply to /127.0.0.1:65251 with teacherNum 298
reply to /127.0.0.1:65252 with teacherNum 299
reply to /127.0.0.1:65253 with teacherNum 300


reply to /127.0.0.1:65150 with studentNum 297
reply to /127.0.0.1:65151 with studentNum 298
reply to /127.0.0.1:65152 with studentNum 299
reply to /127.0.0.1:65153 with studentNum 300
reply to /127.0.0.1:65154 with teacherNum 201
reply to /127.0.0.1:65155 with teacherNum 202
```

Logs in three servers are below.

```
[SUCCESS] date: 2021-06-26 17:26:50, managerID: LVL0001, operation: createTRecord: {recordID: TR00195,
[SUCCESS] date: 2021-06-26 17:26:50, managerID: LVL0001, operation: createTRecord: {recordID: TR00196,
[SUCCESS] date: 2021-06-26 17:26:50, managerID: LVL0001, operation: createTRecord: {recordID: TR00198,

[SUCCESS] date: 2021-06-26 17:26:51, managerID: MTL0001, operation: createTRecord: {recordID: TR00298,
[SUCCESS] date: 2021-06-26 17:26:51, managerID: MTL0001, operation: createTRecord: {recordID: TR00299,
[SUCCESS] date: 2021-06-26 17:26:51, managerID: MTL0001, operation: createTRecord: {recordID: TR00300,

[SUCCESS] date: 2021-06-26 17:26:50, managerID: DDO0001, operation: createTRecord: {recordID: TR00197,
[SUCCESS] date: 2021-06-26 17:26:50, managerID: DDO0001, operation: createTRecord: {recordID: TR00199,
[SUCCESS] date: 2021-06-26 17:26:50, managerID: DDO0001, operation: createTRecord: {recordID: TR00200,
```

At last, I invoke the method getRecordCounts(), I got the result as below. The last student recordID is SR00300, and the last teacher recordID is TR00300. After analysis, we got a conclusion that all recordIDs are unique in the system.

```
MTL 200, LVL 200, DDO 200
```

4.2 Scenario 2:

Description: this test scenario focus on transferRecord() method in the server. I will edit all field in student record and teacher record by three managers “LVL0001”, “MTL0001” and “DDO0001”. By this test scenario, we will know the correctness of editRecordCounts method and log function.

Firstly, we will try some invalid input.

(a) The recordID that does not exist. We try it in LVL client.

```
serverImpl.transferRecord(client.managerID, recordID: "invalidID", Location.DDO.toString());
```

We will get log message as below, and messages are stored in the log files LVL0001 in the client side and LVL in the server side.

```
[ERROR] date: 2021-06-28 15:55:20, managerID: LVL0001, error message:  recordID [invalidID] does not exist.
```

(b) I try the invalid location name in MTL client.

```
serverImpl.transferRecord(client.managerID, recordID: "TR00003", remoteCenterServerName: "InvalidLocation");
```

We will get log message as below, and messages are stored in the log files MTL0001 and MTL.

```
[ERROR] date: 2021-06-28 16:01:00, managerID: MTL0001, error message: the location [InvalidLocation] is invalid
```

Secondly, we will test the correctness of the method, which focuses on atomicity and concurrency.

(c) I test the atomicity of the method in DDO Server. I start three threads with client managers “DDO0001”, “DDO0002” and “DDO0003”, and the record can only be transferred once.

```
public static void main(String[] args) {
    Client client = new Client();
    client.initialClient(new String[]{}, Location.LVL.toString(), clientID: 1);
    System.out.println(client.serverImpl.getRecordCounts(client.managerID));
    Thread a = new Thread(new Thread1());
    Thread b = new Thread(new Thread2());
    Thread c = new Thread(new Thread3());
    a.start();
    b.start();
    c.start();
    while (a.isAlive() || b.isAlive() || c.isAlive()) {

    }
    System.out.println(client.serverImpl.getRecordCounts(client.managerID));
}
```

We got log messages as below, and the quantity of records in the server is also consistent.

```
MTL 2, LVL 3, DDO 4
[SUCCESS] date: 2021-06-28 16:48:35, managerID: DDO0001, operation: transferred Record from server [DDO] into server [LVL]: {recordID: TR00004,
[ERROR] date: 2021-06-28 16:48:35, managerID: DDO0002, error message:  recordID [TR00004] does not exist.
[ERROR] date: 2021-06-28 16:48:35, managerID: DDO0003, error message:  recordID [TR00004] does not exist.
MTL 2, LVL 4, DDO 3
```

We can see that the record “TR0004” is only transferred once. Then we will see the correctness of log function. We got three new log files.

Contents in DDO0001, DDO0002 and DDO003 are showed in order as below, and the log messages are consistent with that in the client console.

```
managerID: DDO0001, operation: transferred Record from server [DDO] into server [LVL]: {recordID: TR00004, first
```

```
[ERROR] date: 2021-06-28 16:28:48, managerID: DDO0002, error message:  recordID [TR00004] does not exist.
```

```
[ERROR] date: 2021-06-28 16:28:48, managerID: DDO0003, error message:  recordID [TR00004] does not exist.
```

We can see the log message in LVL Server.

```
managerID: DDO0001, operation: transferred Record from server [DDO] into server [LVL]: {recordID: TR00004,
```

Contents in DDO Server.

```
[SUCCESS] date: 2021-06-28 16:28:48, managerID: DDO0001, operation: transferred Record from server [DDO] i
[ERROR] date: 2021-06-28 16:28:48, managerID: DDO0002, error message:  recordID [TR00004] does not exist.
[ERROR] date: 2021-06-28 16:28:48, managerID: DDO0003, error message:  recordID [TR00004] does not exist.
```

(d) This case tests the atomicity of the method in LVL Server. I start three threads to edit the field in the record, at the same time, I start a thread to transfer it. We can analysis the result whether is consistent through logs.

```
public static void main(String[] args) {
    Client client = new Client();
    client.initialClient(new String[]{}, Location.LVL.toString(), clientID: 4);
    System.out.println(client.serverImpl.getRecordCounts(client.managerID));
    Thread a = new Thread(new editThread1());
    Thread b = new Thread(new editThread2());
    client.generateLog(client.serverImpl.transferRecord(client.managerID, recordID: "TR00001",
    Thread c = new Thread(new editThread3());
    a.start();
    b.start();
    c.start();
    while (a.isAlive() || b.isAlive() || c.isAlive()) {

    }
    System.out.println(client.serverImpl.getRecordCounts(client.managerID));
}
```

We can see the log messages as below.

```
MTL 2, LVL 3, DDO 4
[SUCCESS] date: 2021-06-28 17:16:26, managerID: LVL0004, operation: transferred Record from server [LVL] into server [DDO]: {recordID: TR00001,
[ERROR] date: 2021-06-28 17:16:26, managerID: LVL0002, error message:  recordID [TR00001] does not exist.
[ERROR] date: 2021-06-28 17:16:26, managerID: LVL0003, error message:  recordID [TR00001] does not exist.
[ERROR] date: 2021-06-28 17:16:26, managerID: LVL0001, error message:  recordID [TR00001] does not exist.
MTL 2, LVL 2, DDO 5
```

Log messages in LVL Server:

```
[SUCCESS] date: 2021-06-28 17:16:26, managerID: LVL0004, operation: transferred Record from server [LVL] into server [DDO]: {recordID:
[ERROR] date: 2021-06-28 17:16:26, managerID: LVL0002, error message:  recordID [TR00001] does not exist.
[ERROR] date: 2021-06-28 17:16:26, managerID: LVL0003, error message:  recordID [TR00001] does not exist.
[ERROR] date: 2021-06-28 17:16:26, managerID: LVL0001, error message:  recordID [TR00001] does not exist.
```

Log messages in DDO Server:

```
[SUCCESS] date: 2021-06-28 17:16:26, managerID: LVL0004, operation: transferred Record from server [LVL] into server [DDO]: {recordID: TR00001,
```

The content is consistent in the system. In the next step, we change the sequence of threads as below.

```
public static void main(String[] args) {
    Client client = new Client();
    client.initialClient(new String[]{}, Location.LVL.toString(), clientID: 4);
    System.out.println(client.serverImpl.getRecordCounts(client.managerID));
    Thread a = new Thread(new editThread1());
    Thread b = new Thread(new editThread2());
    Thread c = new Thread(new editThread3());
    a.start();
    b.start();
    c.start();
    while (a.isAlive() || b.isAlive() || c.isAlive()) {

    }
    client.generateLog(client.serverImpl.transferRecord(client.managerID, recordID: "TR00001",
    System.out.println(client.serverImpl.getRecordCounts(client.managerID));
}
```

We can see the log messages as below.

```
MTL 2, LVL 3, DDO 4
[SUCCESS] date: 2021-06-28 17:25:04, managerID: LVL0001, operation: editValue: { recordID: TR00001, old value: mockAddress, new value: threadA }
[SUCCESS] date: 2021-06-28 17:25:04, managerID: LVL0002, operation: editValue: { recordID: TR00001, old value: threadA, new value: threadB }
[SUCCESS] date: 2021-06-28 17:25:04, managerID: LVL0003, operation: editValue: { recordID: TR00001, old value: threadB, new value: threadC }
[SUCCESS] date: 2021-06-28 17:25:04, managerID: LVL0004, operation: transferred Record from server [LVL] into server [DDO]: {recordID: TR00001, first name:
MTL 2, LVL 2, DDO 5
```

Log messages in LVL Server:

```
managerID: LVL0001, operation: editValue: { recordID: TR00001, old value: mockAddress, new value: threadA }
managerID: LVL0002, operation: editValue: { recordID: TR00001, old value: threadA, new value: threadB }
managerID: LVL0003, operation: editValue: { recordID: TR00001, old value: threadB, new value: threadC }
managerID: LVL0004, operation: transferred Record from server [LVL] into server [DDO]: {recordID: TR00001, f
```

Log messages in DDO Server:

```
[LVL] into server [DDO]: {recordID: TR00001, first name: mockFirstName, last name: mockLastName, address: threadC
```

The content of the record is consistent in the whole system.

5. Important parts:

The implementation of configuration and log function is introduced in the last report, so I focus on the new part in this assginment.

5.1 IDServer

I set up a new server to create unique recordIDs for new records. LVL, MTL and DDO Server have to communicate with IDServer to get the unique recordID when they want to create new StudentRecord or TeacherRecord. In IDServer, I use AtomicInteger class, which uses CompareAndSwap mechanism, to ensure the consistency of unique records. Compared with traditional synchronized methods, AtomicInteger has better performance. As we know, blocking and waking up threads are very time-consuming, and when some threads share the same AtomicInteger, they will operate its value in the memory directly and not be blocked. I use it to maximize the concurrency while ensuring the uniqueness of recordIDs.

```
private static AtomicInteger studentNum = new AtomicInteger( initialValue: 0);
private static AtomicInteger teacherNum = new AtomicInteger( initialValue: 0);
```

5.2 implementation of transferRecord():

I use TCP programming to send the records to the target server for its reliability. The first step is to find the record by its recordID. If the record exists in the server, we need to serialize it. The

serialization method for Student is shown as below.

```
public String toSerialize() {
    StringBuilder sb = new StringBuilder("student,");
    sb.append(recordID).append(",").append(firstName).append(",").append(lastName).append(",");
    for (String str: coursesRegistered) {
        sb.append(str);
        if (str.equals(coursesRegistered[coursesRegistered.length - 1])) {
            continue;
        }
        sb.append("&");
    }
    sb.append(",").append(status).append(",").append(statusDate).toString();
    return sb.toString();
}
```

```
public String toSerialize() {
    StringBuilder sb = new StringBuilder("teacher,");
    return sb.append(recordID).append(",").append(firstName).append(",").append(lastName).append(",").append(add
            .append(phone).append(",").append(specialization).append(",").append(location).toString();
}
```

After serialization, I use TCP to transfer the record to another server for its reliability, and the address of the server is stored in the configuration file. I also start a new thread in each server to listen on the port.

```
ServerSocket serverSocket = new ServerSocket(port);
System.out.println("Transfer Record Thread is ready.");
while (true) {
    Socket connectionSocket = serverSocket.accept();
    //Get values from client
    BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocl
    //Get OutputStream at server to send values to client
    DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream
    //Get the input message from cli.ent and then print
    String message = inFromClient.readLine();
    Record record;
    if (message.startsWith("student")) {
        record = StudentRecord.deserialize(message);
    } else {
        record = TeacherRecord.deserialize(message);
    }
    if (record == null) {
        outToClient.writeBytes(generateLog( status: "[ERROR]", extractManagerID(message) ,
        continue;
    }
    synchronized (recordMap) {
        List<Record> recordList = new LinkedList<>();
        recordList.add(record);
        insertRecords(recordList);
    }
    outToClient.writeBytes( s: generateLog( status: "[SUCCESS]", extractManagerID(message) ,
}
```

When a thread accepts a new record transferred from another server, it will check the format of the record and deserialize it to the object in the current Java Virtual Machine. Then it will try to get the access of shared data structure ‘recordMap’, and add the record into it if successes. Finally, it will generate the log message according to the result and return it. The server invoking the method will get the log message and write it into its log file.

5.3 server startup method:

We should indicate the location of the server as parameters.

```
public static void main(String[] args) {
    startServer(args, Location.MTL);
}
```

Then the server will start Count Thread and Transfer Record Thread which is introduced in my report. At the next step, the server will set up some initial data and the context in the method initiate(). Then I start the ORB Naming Service and bind this server object with the location of the server.

```
startCountThread();
startTransferRecordThread();
initiate();
try {
    ORB orb = ORB.init(args, configuration.getProperties());
    // Portable Object Adapter (POA)
    // get reference to rootpoa & activate the POAManager
    POA rootpoa = (POA) orb.resolve_initial_references( object_name: "RootPOA");
    rootpoa.the_POAManager().activate();
    // create servant and register it with the ORB
    ServerImpl serverImpl = new ServerImpl();
    // get object reference from the servant
    org.omg.CORBA.Object ref = rootpoa.servant_to_reference(serverImpl);
    // and cast the reference to a CORBA reference
    Server href = ServerHelper.narrow(ref);
    // get the root naming context
    // NameService invokes the transient name service
    org.omg.CORBA.Object objRef = orb.resolve_initial_references( object_name: "NameService");
    // Use NamingContextExt, which is part of the
    // Interoperable Naming Service (INS) specification.
    NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
    // bind the Object Reference in Naming
    String name = location.toString();
    NameComponent[] path = ncRef.to_name(name);
    ncRef.rebind(path, href);
    System.out.println(location.toString() + " Server is ready.");
    // wait for invocations from clients
    orb.run();
} catch (WrongPolicy | CannotProceed | org.omg.CosNaming.NamingContextPackage.InvalidName |
    e.printStackTrace();
}
```