Class Management System

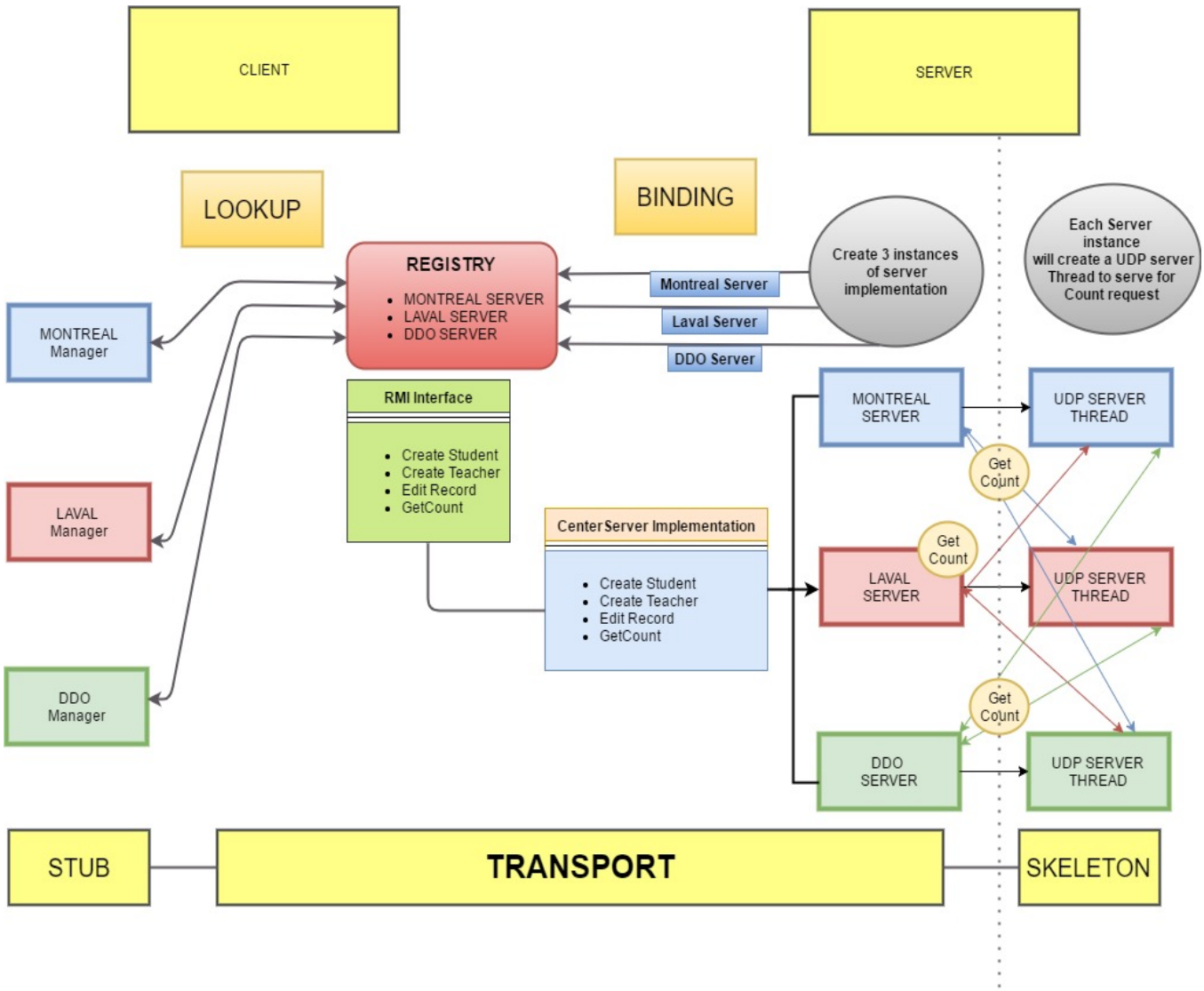
Prepared By:

Shyam Kantesariya(40042715) Prasanth Ambalam Jawaharlal (40042116)

Objective:

Design and develop a distributed system which provides an infrastructure for Class Management System shared across three data centers. Expose Java RMI APIs to the user which abstract the underlying distributed nature of the platform and provide network transparency. Implement multi threading with proper synchronization to handle concurrent requests safely.

Overall Design Architecture:



RMI Interface:

- createTRecord(firstName, lastName, address, phone, specialization, location)
- createSRecord(firstName, lastName, coursesRegistered, status, statusDate)
- getRecordCounts()
- editRecord(recordID, fieldName, newValue)

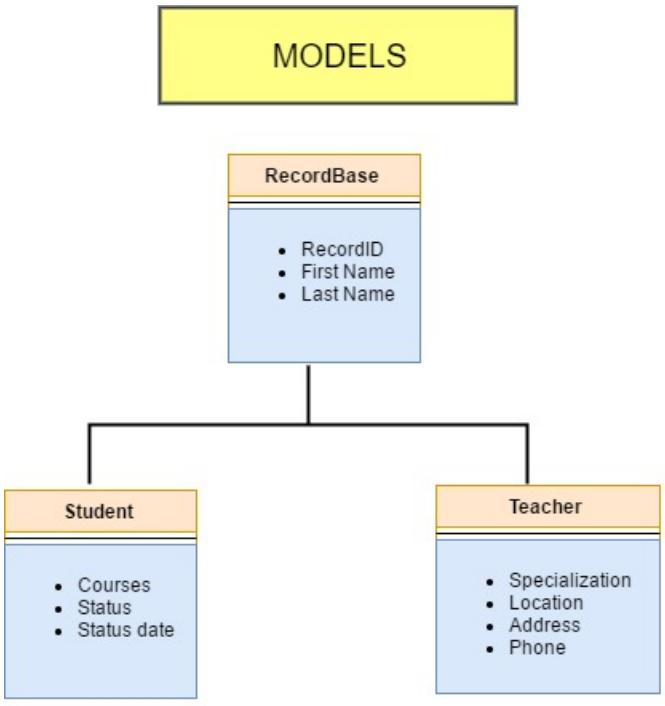
RMI Server : (CenterServerImpl)

- This Class implements the RMI interface.
- Three instances of RMI server implementation has been created. One for each location (MTL, LVL, DDO);

RMI Registry:

- The three instances of CenterServerImpl has been binded to the registry in three different names, thereby exposing the objects to the client.
    - registry.bind("MontrealServer", mtlServer);
    - registry.bind("LavalServer", lvlServer);
    - registry.bind("DDOServer", ddoServer);

Data Modelling:



Hash Map

KEY	VALUE (List<RecordBase>)
"A"	RecordBase (Student) RecordBase (Teacher)
"B"	RecordBase (Teacher) RecordBase (Teacher) RecordBase (Student)
...	...
"K"	RecordBase(Student) RecordBase(Student)

RecordBase:

- This is the base model which holds the common properties like firstname, lastname and recordID.

Student:

- This extends the base class RecordBase and has its own properties like Courses, Status and StatusDate

Teacher:

- This class extends the base class RecordBase and has its own properties like Specialization, Location, Phone and Address.

HashMap:

- Signature : **HashMap<String, List<RecordBase>>**
- Hashmap is used as a database for storing/retrieving/editing teacher and student records.
- Since both Teacher and Student derives from RecordBase, both the record types can be stored in the same HashMap.
- **Storing Procedure:**
    - The first letter of the last name of any record has been used as the key.
    - Since Lastname is used as the key in the hashmap. It has been validated against
        - Empty
        - Null
    - If Validation Passes
        - The record will be added to the list which corresponds to the key in the hashmap.
        - Log this event in journal file on persistent storage.
    - Else
        - Last Name validation failed exception will be thrown.
- **Editing Procedure:**
    - The record will be searched in the hashmap based on the RecordID
    - If a matching record is found
        - The input field name and value will be validated
        - If Validation is success
            - Then the record will be updated with the new value
            - Log this event in journal file on persistent storage.
        - Else
            - Validation Failed Exception will be thrown
    - Else
        - Record Not Found Exception will be thrown

Logging for troubleshooting :

We have used the inbuilt logger provided by Java (java.util.logging) to perform logging in both server and client side.

Log Format:

Generally each log entry should have the following data

- Timestamp
- Class and function name which performs the action

Center Server:

Each server log (Montreal, Laval, DDO) will be saved in their respective folder

- LOGS/MTL/logger.log
- LOGS/LVL/logger.log
- LOGS/DDO/logger.log

Following events will be captured in log file

- Record Created
- Record Edited
- Exceptions
- UDP communications

Client:

Every Manager is a client. A log file will be created for each manager and actions performed by him/her will be logged. The actions logged are as follows

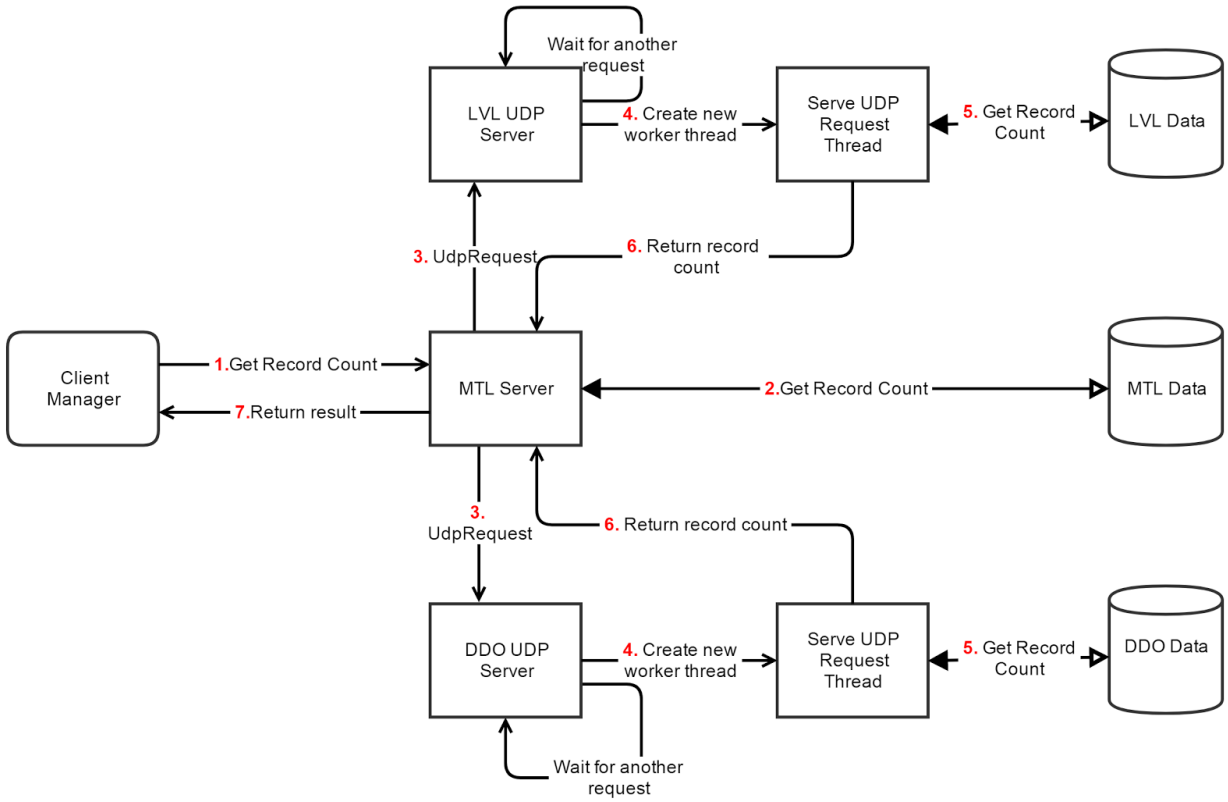
- Creating record
- Editing Record
- Exceptions

Implementation:

- Create a Unique logger for each Server. (java.util.logging)
- Add a file handler to save the contents to the respective log file.
- Log using various levels like (INFO, WARNING, ERROR (java.util.logging.Level)) based on the severity.

UDP Server Design:

On receiving get record count request each server will fork get count request threads to remaining servers. Communication among all three servers take place using UDP/IP protocol.



Execution flow:

1. Manager Client issues get record count request to respective server
2. Server access its data set to fetch the record count details
3. Generates two new UDP request threads to send get count request to remaining two locations.
4. UDP server of respective location will receive the request and generate new worker thread to serve that request
5. Worker thread will access respective data set to fetch the record count details
6. Worker thread will respond to the request with record count details
7. Sever who received get record count request will collect results and return to the Manager client

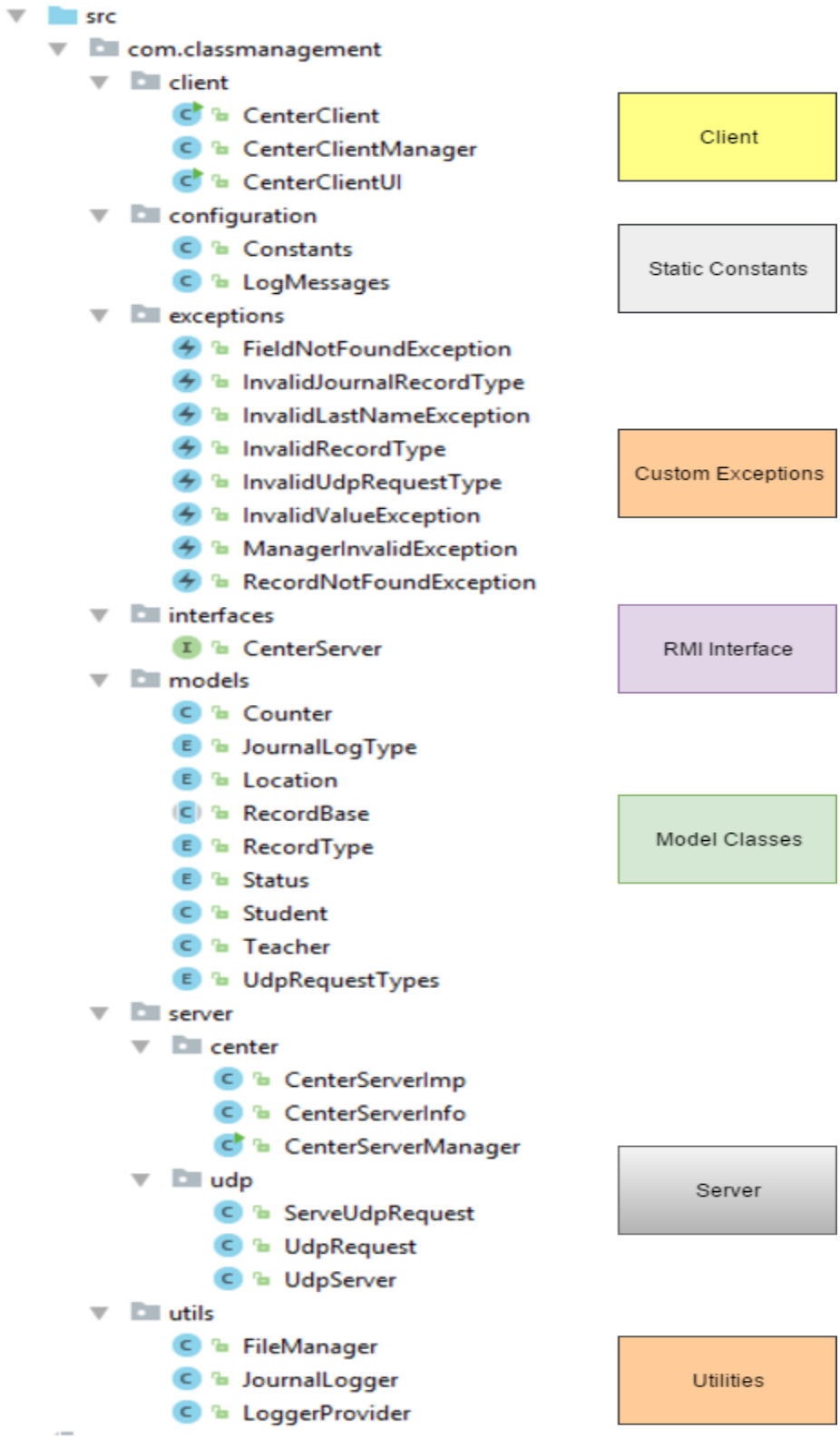
Concurrency:

- Location Server creates new thread to communicate to each UDP server
- UDP Server creates a new worker thread for each coming request and delegates the service request.

Synchronization:

- Each request to access Data set is synchronized. Hence only one thread can get the hold on it which makes it thread safe.
- Additionally, each thread is a new object hence possesses its own memory space hence no shared resources among other threads.

Code design and structure:



Challenges:

- Fine grained locking on shared objects
    - HashMap: Central data set
    - Counter: Unique record ID generator
- Concurrency in UDP server

Reliability:

We have implemented two features for reliability measures.

- **Journaling**
    - Motivation: As HashMap remains in memory. any server crash event will result into whole data lost. Journaling is the solution to provide backup of HashMap data.
    - Design: Any Edit/Add action performed on HashMap will be recorded in the journal file which remains on persistent storage with event identifier and required data.
    - Failure recovery: In case of any server crash event, parse through the journal file to recover whole data.
- **Serialization of Counter objects**
    - Motivation: Counter objects are being used to generate unique record id. Any inconsistency in that operation may cause duplicate record Ids and inconsistent data eventually.
    - Design: Make counter objects serialized and store it into a persistent storage on each modification.
    - Failure recovery: Read counter object from storage and deserialize it during server recovery phase.

Test Scenarios:

1. Add Student record with different inputs
2. Add Teacher record with different inputs
3. Update student record
4. Update Teacher record
5. Get record count for all servers
6. Student status date should be updated whenever status changes.
7. Exception handling for the following scenarios:
    a. Invalid last name while creating a record.
    b. Invalid Record ID during editing a record.
    c. Invalid value. (foe eg: Location, status)
    d. Trying to update Invalid field (for eg: firstname)
    e. Invalid Manager ID (Manager ID should start with MTL, LVL, DDO)

The above scenarios should also work when making multiple requests asynchronously using threads.

Learnings:

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

References:

- Java Serialization:
  https://stackoverflow.com/questions/2836646/java-serializable-object-to-byte-array
  http://www.javapractices.com/topic/TopicAction.do?Id=57
- Random access a binary file:
  http://docs.oracle.com/javase/6/docs/api/java/io/RandomAccessFile.html
- Java coding standards:
  https://google.github.io/styleguide/javaguide.html
- Java Set implementation:
  http://tutorials.jenkov.com/java-collections/set.html
- Get record count on hashmap:
  https://stackoverflow.com/questions/5496944/java-count-the-total-number-of-items-in-a-hashmapstring-arrayliststring
- Get all values of enum:
  https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html
- Get current directory of Java application:
  https://stackoverflow.com/questions/4871051/getting-the-current-working-directory-in-java
- Java int datatype range:
  https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
- Private port range:
  https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
- Java reflection:
  https://stackoverflow.com/questions/3333974/how-to-loop-over-a-class-attributes-in-java