Distributed Employee Management System (DEMS) Using Java RMI

SOEN 423, Fall 2018

By Shunyu Wang

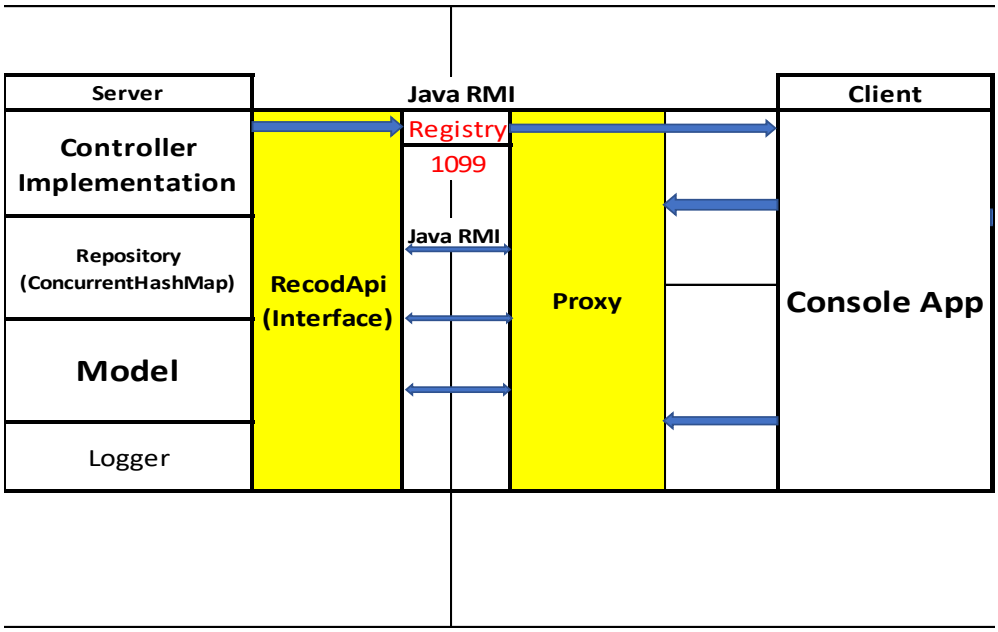
Last Updated on Oct 6, 2018

1. Overview

Implementing a distributed employee management system to demonstrate the Remote Method Invocation in Java. The system has client and three servers located in different countries and adopts Java RMI as middleware to handle message passing between users and hosts. Among servers, data communication is done by UDP. The system support serving multi-requests concurrently due to the benefit of Java RMI, and all shared resources on server end are thread-safe. An end user which is any HR manager could log in with their user name and be properly directed to his or her corresponding server to add and modify the employee records.

2. Functional Description

2.1. The Client-Server model



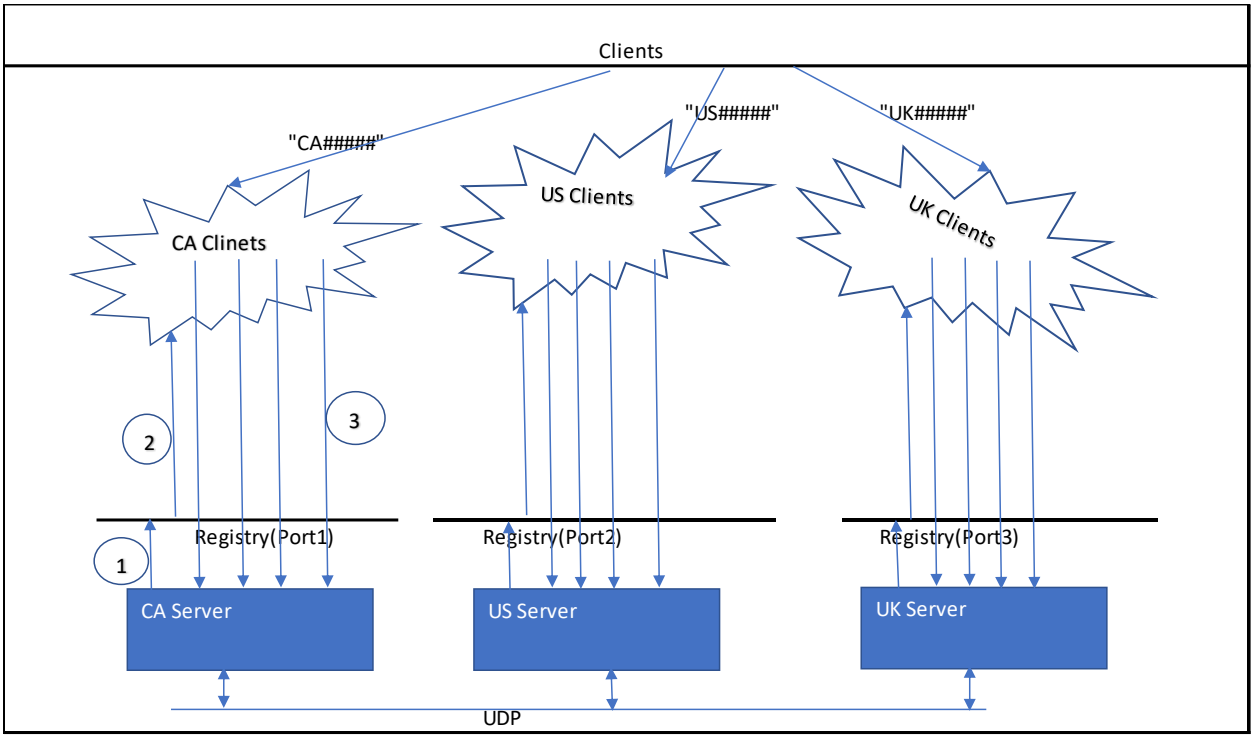
Each single server contains the same interface, here we call RecordApi. It serves as the stubs or proxy in the client so that the client could know exactly the behaviors of the remote object, which has already been registered into the RMI registry.

In the server, multiple-layers architecture is employed. The controller implements the business logic, and calls the methods provided by the repository to manipulate employee records. It also contains a logger which could log the users' actions upon the returned message from the

repository. The repository contains a ConcurrentHashMap as a field as the data storage and well support concurrent requests. The models are responsible for data models, and examples are abstract Record model which is inherited by EmployeeRecord and ManagerRecord.

In the client, the simple console application asks for HR manager's user ID, and match it with his or her corresponding RMI registry location based on its format. And then it acquires the remote object from indicated RMI registry and calls the servers available from the Proxy.

2.2. The Distributed System



To establish a distributed system, the localhost hosts three servers by hosting their registries on different ports. Arrow 1 illustrate this process, upon the server starts, the program should register an instance that implementing the interface into its RMI registry. And then, clients will fetch that instance from its registry from that port of which server they would like to access. Arrow 2 illustrates this process. The instance will be casted as interface type and the client-side object could invoke all those methods in the interface. As Arrow 3 indicates, the clients could invoke the server functions as they are installed locally.

Server to server communication is through UDP. During the server program starts, it will launch a daemon thread of UPD that keeps listening to requests and sending responses back to requester.

2.3. Domain models

2.3.1 Interface and Controller

The interface describes what the server controller object can do, and also server as a proxy in client-side.

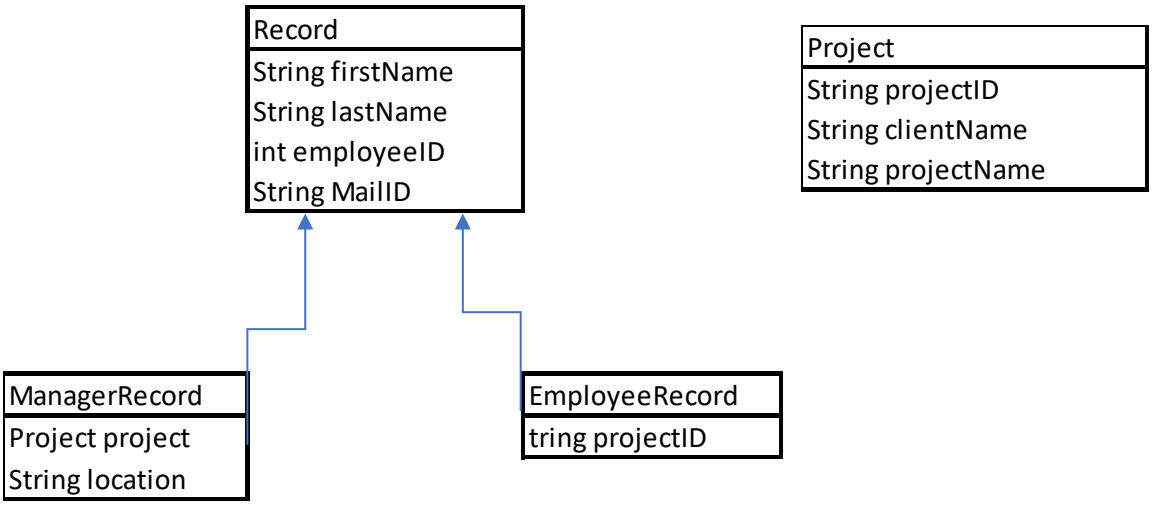
public interface **RecordApi**
public String createMRecord(String firstName, String lastName, Integer employeeID, String mailID, Project project, String location) throws RemoteException; public String createERecord(String firstName, String lastName, Integer employeeID, String mailID, String projectID) throws RemoteException;; public String getRecordCounts() throws RemoteException;; public String editRecord(String recordID, String fieldName, String newValue) throws RemoteException;; public String printData() throws RemoteException;;

In the context of RMI, the interface RecordApi should extend Remote. The real implementation on server is done by RecordController. The logger is used to log specific actions into to a log file depending on the results returned by the repository(See 2.3.3).

RecordController
IRecordRepository repo Logger logger

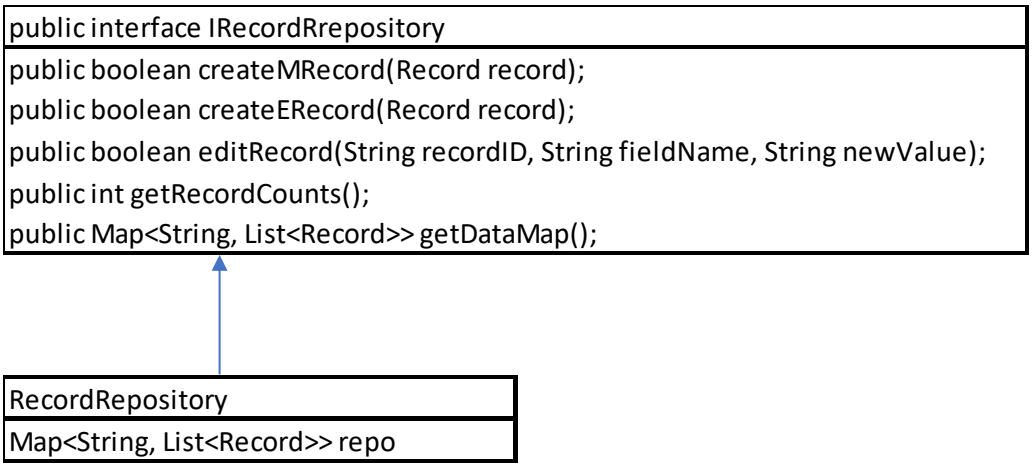
The controller has a repository and logger as a filed and invokes their method to process requests.

2.3.2. Model Classes



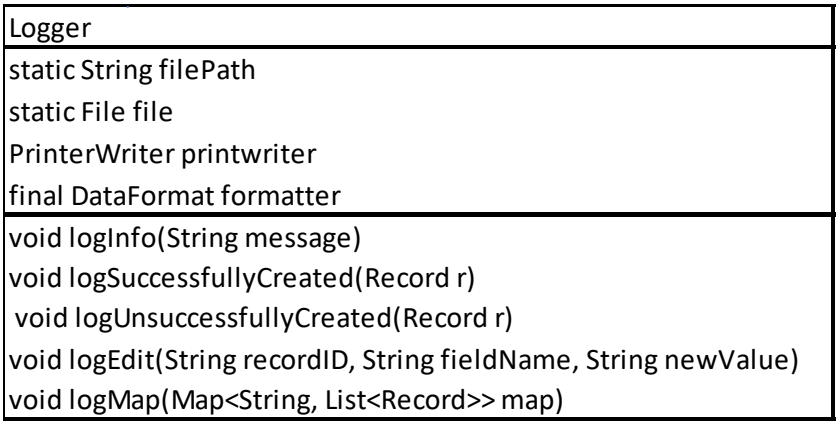
Since the Project class is used in interface and proxy as a type, it has to implements Serializable interface.

2.3.3. The Repository



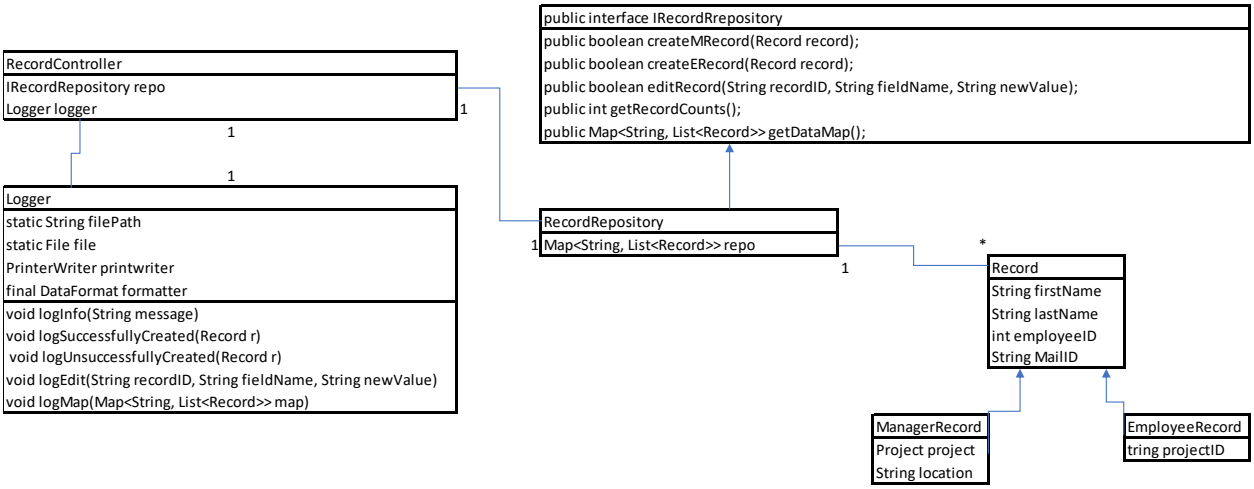
The data structure used in RecordRepository is a ConcurrentHashMap, so that all concurrent requests could be handled properly. All methods will have a return type so that the controller could tell whether the operation is implemented successfully or not.

2.3.4. Logger



The server has a default filePath and has a file related to that path. The logger will log date using PrintWriter in each method on that file after the server is started. Each method is “synchronized” so that there will be only one thread can write into the file.

2.4. Class Diagram



2.5. Multi-threads Programming

The RMI middleware will create and handle multithreads when there are multiple clients tending to access the server. However, in the server implementation, we need to guarantee that the data storage and log file should be thread-safe. To achieve this goal, we use ConcurrentHashMap for as the dedicated data storage and synchronized key word to modify methods in Repository and also use “synchronize” key word on each method in the logger class. So, the shared storage space and log file are not under the threat of being modified by concurrent threads.

2.6. Server Communication

Each server runs a UDP Server as a daemon thread upon it is started, and listens to the port where the UDP request from. Then it will invoke the RecordCounts method in repository, format the count and send the response back to the requesting server’s address and port. When the user invokes the method on CA Server, the implementation will send UDP requests to US Server and UK Server. After responses arrived, the method will return these responses and its own record counts in a well formatted string.

3. Unit Tests and Integration Tests

3.1 Unit Tests

The key features, such as add or edit records, are primarily implemented in the repository level. So unit testing on the methods of Repository class is important.

Test Case	Method	Input	Initial State	Expected Output	Output as Expected
1	boolean createMRecord(Record record);	1 Manager Record	Empty internal map	return true; map.size = 1	YES
2	boolean createERecord(Record record);	1 Employee Record	Empty internal map	return true; map.size = 1	YES
3	boolean editRecord(String recordID, String fieldName, String newValue);	recordID = ER10001, fieldName = projectID, newValue = P21231	Internal map has an initial record with id ER10001 whose projectID is P0001	return true; Record ER10001 has P21231 as projectID	YES
4	boolean editRecord(String recordID, String fieldName, String newValue);	recordID = MR10001, fieldName = location, newValue = "UK"	Internal map has an initial record with id ER10001 whose location is CA	return true; Record MR10001 has UK as location	YES
5	boolean editRecord(String recordID, String fieldName, String newValue);	recordID = MR10001, fieldName = clientName, newValue = "Smith"	Internal map has an initial record with id ER10001 whose clientName is Jack	return true; Record MR10001 has clientName as clientName in its project	YES
6	int getRecordCounts();	N/A	Internal map has 3 ManagerRecords and 2 Employee Records. There are only two last names starting with "W", "L"	map.keySet.size = 2, map.get("W").size + map.get("L").size = 5	YES
7	Map<String, List<Record>> getDataMap();	N/A	Internal map has 3 ManagerRecords and 2 Employee Records. There are only two last names starting with "W", "L"	map.keySet.size = 2, map.get("W").size + map.get("L").size = 5	YES

3.2. Integration Tests

Since multiple clients may access operations on the same server concurrently, the system should manipulate concurrency properly. So I let Client class extends Thread class. And put repetitive method calls in the overrided run() method. For example, creating manager records for 10 times and creating employee records for 10 times. Then creating 10 Client threads and starting them concurrently. We would implement a few more times. First there shouldn't be any exceptions thrown by the system regarding concurrency issue. Then initiating a new console app from Client class. Calling getRecordCounts should get exactly 200 records on that server. And then adding more operations, such as editing same the record for 10 times, and rerunning the code should result no exception. Covering all methods in the interface should bring us to the conclusion that the systems well handled multiple clients' accessing concurrently.

4. Important Issues and Implications

1) Interface and Proxy should be exactly the same and have the same package directory in both(client and server) project.

2) Any customized objects passed through RMI should have their classes implement Serializable interface. Otherwise, it would result in an Unmarshal Exceptions.
3) Registry ports and UDP ports should be unique on localhost.
4) Close Scanner, PrintWriter, and Socket shouldn't be forgot so that they can release system resources.
5) Using RMI correctly is important. The interface should extend Remote, and the implementing Class should extend UnicastRemoteObject.
6) The sequence of launching the system is important. Launching three servers in any order and then any number of clients works fine. Otherwise, it doesn't work because there is no registed remote object in RMI registry, which even doesn't exist.