

Distributed Employee Management System (DEMS) Using CORBA

SOEN 423, Fall 2018

By Shunyu Wang

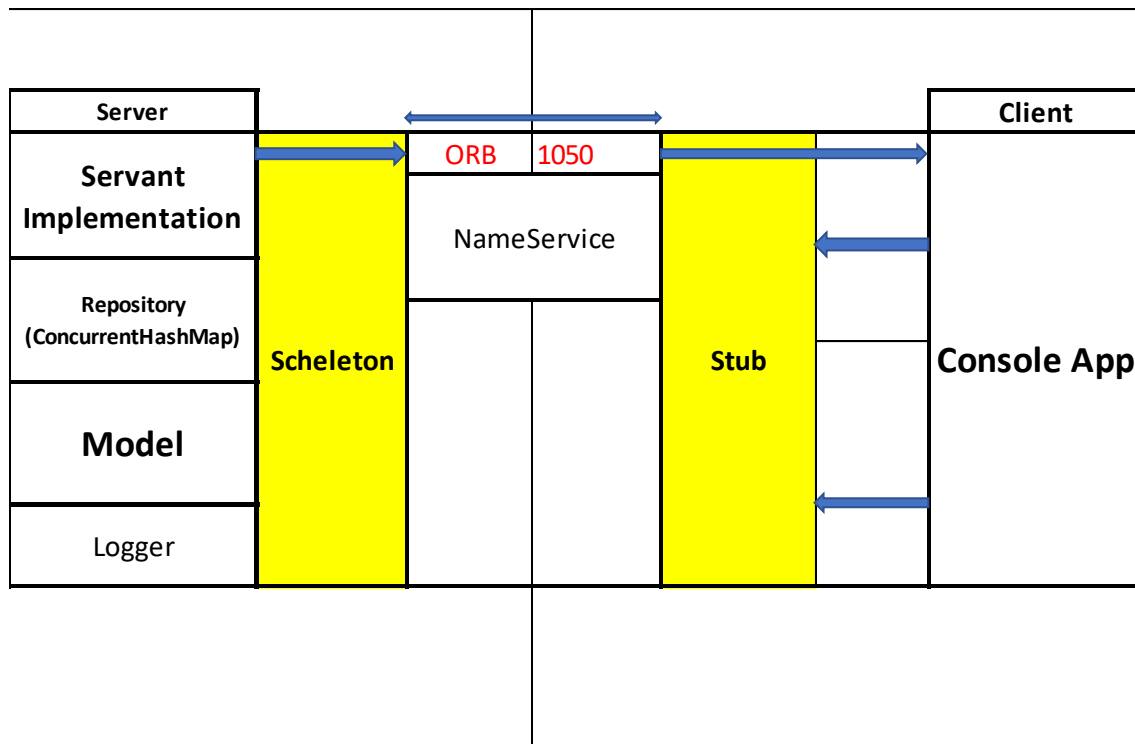
Last Updated on Oct 27, 2018

1. Overview

In this distributed system, CORBA is used to replace Java RMI as middleware to facilitate server and client communications. In this design document, focus is given to the additional UDP communications and CORBA implementations. Servant implementation and data layer doesn't change significantly and remains the same except for some minor ad justifications to usage of CORBA.

2. Functional Description

2.1. The Client-Server Model

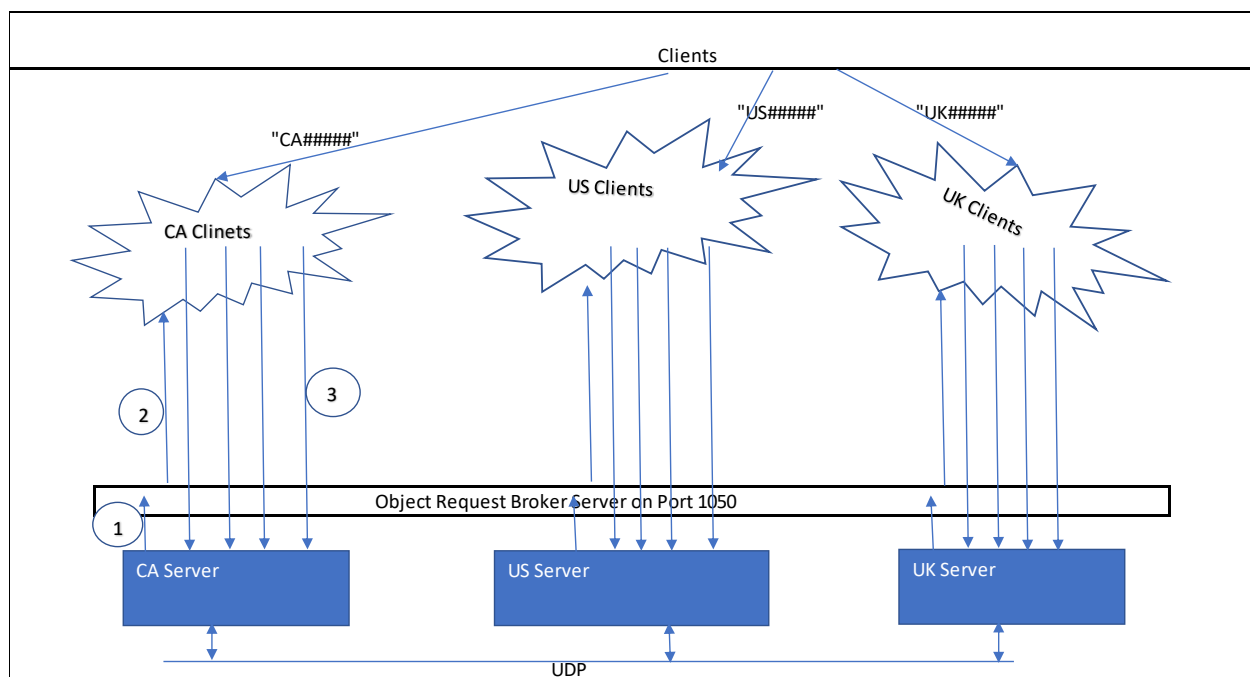


By firstly defining IDL file which is the interface in CORBA, the middleware could automatically generate client stub and server skeleton. Each single server contains the same skeleton, here we call RecordPOA. _RecordStub serves as the stubs or proxy in the client so that the client could know exactly the behaviors of the remote object, whose reference has already been registered into the CORBA NameService.

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 HashMap as a field as the data storage. 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 depending on the prefix of the user ID, the application will fetch the reference to the object in CORBA NameService. Then the application could call methods in stub to pass parameters and get real implementation of the remote object.

2.2. The Distributed System



To establish a distributed system, the localhost hosts three servers by hosting their servants associate with a unique name on Object Request Broker server. Arrow 1 illustrates this process, upon the server starts, the program should register a reference to the object implementing the interface into CORBA NameService. And then, clients will fetch that reference from the NameService depending on the unique name the implementing object binds. 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. The ORB will take care of routing the request from client to the server and activate implementing object.

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 IDL are used to generate stub and skeletons, and also the user-defined object passed through ORB.

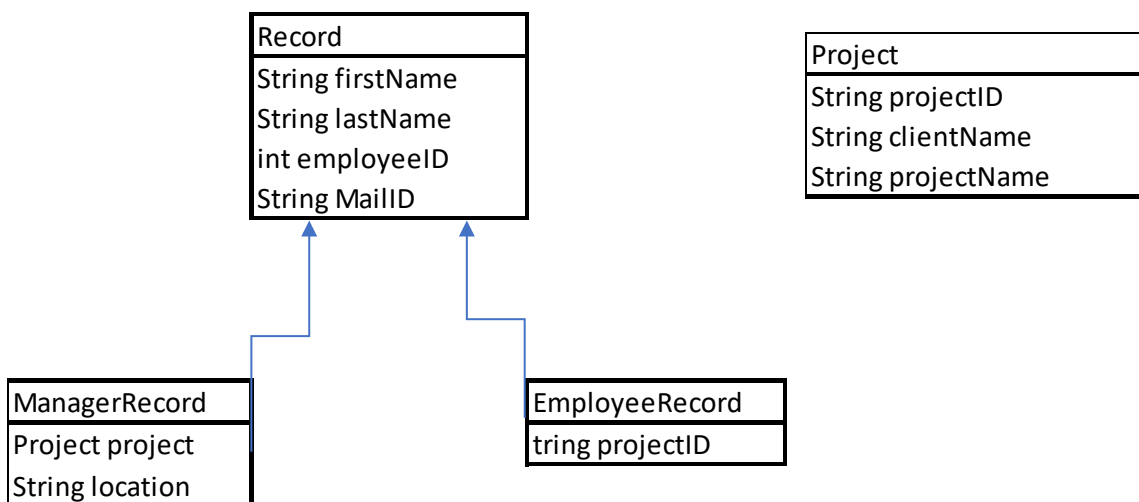
```
module RecordApp
{
    interface Record
    {
        struct Project {
            string projectID;
            string clientName;
            string projectName;
        };
        string createMRecord(in string managerID, in string firstName, in string lastName, in long employeeID, in string mailID, in Project project, in string location);
        string createERRecord(in string managerID, in string firstName, in string lastName, in long employeeID, in string mailID, in string projectID);
        string getRecordCounts(in string managerID);
        string editRecord(in string managerID, in string recordID, in string fieldName, in string newValue);
        string transferRecord (in string managerID, in string recordID, in string remoteCenterServerName);
        string printData();
    };
};
```

In the context of CORBA implementation, the implement object RecordController should extend skeleton. 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
private ORB orb 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

public interface IRecordRepository
public boolean createMRecord(Record record);
public boolean createERecord(Record record);
public boolean editRecord(String recordID, String fieldName, String newValue);
public int getRecordCounts();
public Map<String, List<Record>> getDataMap();
public boolean isExisted(String recordID);
public Record getRecord(String recordID);
public boolean deleteRecord(String recordID);

RecordRepository

Map<String, List<Record>> repo



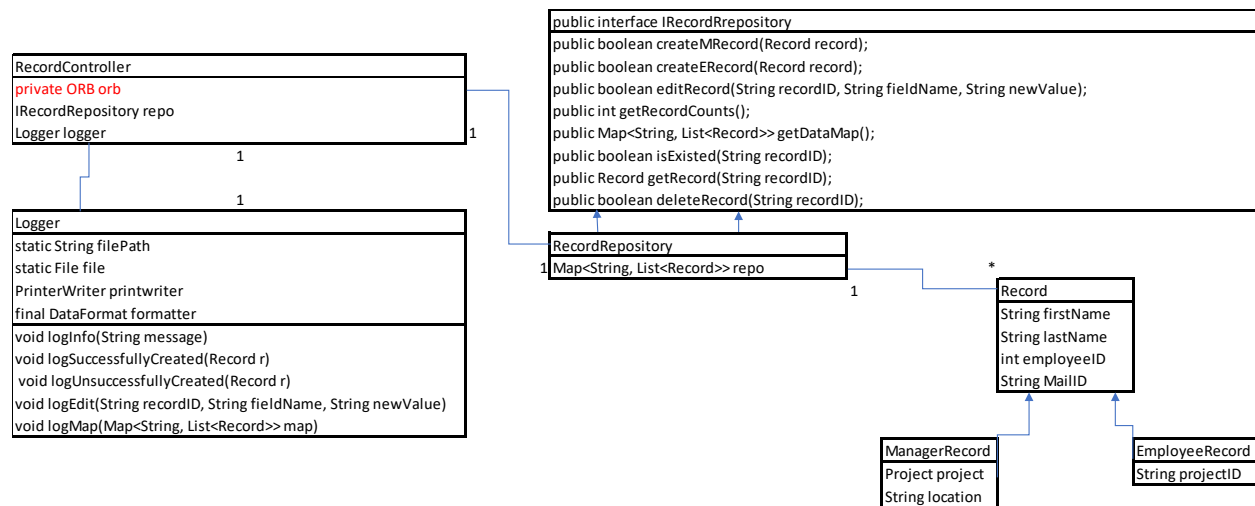
The data structure used in RecordRepository is a ConcurrentHashMap, so that all concurrent requests could be handled properly. All methods will have a return type so that the controller could tell whether the operation is implemented successfully or not.

2.3.4. Logger

Logger
static String filePath
static File file
PrintWriter printwriter
final DataFormat formatter
void logInfo(String message)
void logSuccessfullyCreated(Record r)
void logUnsuccessfullyCreated(Record r)
void logEdit(String recordID, String fieldName, String newValue)
void logMap(Map<String, List<Record>> map)

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 CORBA 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, log file, access to UDP response should be thread-safe. To achieve this goal, we use synchronized methods for `RecordRepository` which includes a `HashMap` as the dedicated data storage, and also use “synchronize” keyword 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.

However, due to the addition of the new method in the interface, there are lots of UPD requests and response between servers and clients. If synchronization is not guaranteed, one UDP client could receives response that does not respond to the request it has sent. It may receives

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 comes from. However, due to the addition of the new method in the interface, there are lots of UPD requests of different purposes and responses among servers and clients. A UDP Client that includes a few static methods that are used to send requests. If synchronization is not guaranteed, one UDP client could receives response that does not respond to the request it has sent. It may mismatch UDP requests in short when threads runs concurrently.

Thus, “synchronized static” is used for each method in the singleton `UDPClient` class. Synchronized static method will lock the whole class when one client is calling the method in the class. Only the request and response cycle is completed and the result is returned, other thread could get the right of using this class. This guarantees, in one server, only one request is waiting for the response. Thus, there is no mismatch problem.

Here is the `UDPClient` Class.

Class UDPClient
<pre> public synchronized static String getRecordCounts(String address, int port) public synchronized static String checkRecordInRemoteServer(String address, int port, String recordID) public synchronized static String transeferRecord(String address, int port, Record record) public synchronized static String rollbackTransfer(String address, int port, String recordID) </pre>

Here a set of rules is defined. For example, the request message is “1” for getRecordCounts, and “2” + recordID for checkRecordInRemoteServer. The message is short and doesn’t take too much bandwidth. The UDP server will match request controller branch by examining the first char. Each response is either the content it request in string or “true” or “false”. For example, when the application client invokes 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 such as “US 3” and “UK 2” arrived, the method will return these responses and its own record counts in a well formatted string.

2.7. Passing Object through UDP

A utility class contributed by meldoze is used to convert object to byte arrays and vise versa. ObjectOutputStream is used internally to achieve such purpose. When transferring record, the object is passed as byte arrays, following the rule number 3, as I discussed above.

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 createERRecord(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 overridden 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 the test 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 the same 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) Stub and skeleton should be exactly generated by ORB and thus there is no mismatching during invocation.
- 2) Synchronizing UPD client is important to guarantee no mismatching of request and response.
- 3) ORB server should be run first so that server and client works correctly. When the application is down, effort is need to bring down the ORB.

- 4) Close Scanner, PrintWriter, and Socket shouldn't be forgot so that they can release system resources.
- 5) Using CORBA generate files correctly is important. The implating class should extend POA and contains an instance of ORB, and the reference acquired from ORB should be cast an interface type, in this case, "Record"
- 6) Binding and Rebinding name and object reference should be focused. Binding an object reference to a name already associate with another object will through AlreadyBound exception. Thus, "rebind" is a safer way to do binding.