CONCORDIA UNIVERSITY

DEPARTMENT OF CSE

COMP 6231, Fall 2013

Assignment 2

Daniel Ricci, 9118756

B.Sc. Computer Science

M.Sc. Applied Computer Science [IP]

Date Submitted: October 24th, 2013

1. Introduction

The Distributed Police Information System is a distributed software application that allows users to input various types of records containing criminal and missing person’s information. The goal of this system is to demonstrate to other individuals how a concurrent system would interact with various processes across a LAN using the Java CORBA plugin.

## 1.1 Purpose and Scope

The main purpose of this assignment is to experiment with handling concurrency across distributed objects and to introduce ourselves to the Java CORBA plugin. A lot of work was put into the organization of the code since it is extremely important to think about where the system can do the least amount of work as possible. Furthermore, a server to server communication protocol was established to allow for Process to Process simulated communication with the use of the UDP Protocol.

## 1.2 Document Format and Expectation

This document will **briefly** outline the requirements for this assignment along with various test cases and personal experiences. **Please note**, that there are no sequence diagram charts or UML diagrams of any nature in this document; rather **there are screenshots** used to provide you with an idea of how the system works. Furthermore, the text in this document will describe the various workflows which exist throughout the system; *since it was not written on the assignment paper that UML diagrams were mandatory, I decided to simply write out how my application work.*

*\*\* Before continuing \*\**

**This document contains information pertinent to Assignment 2 (Distributed Police Information System) given to students of the class COMP 6231 for the Fall 2013 term. This assignment was completely solely by Daniel Ricci** [**thedanny09@gmail.com**](mailto:thedanny09@gmail.com) **without any help from students attending the same class; under no circumstances should the contents of this assignment be used outside the scope of this demo for any purpose without prior consent of Daniel Ricci ID# 9118756.**

1. System Requirements & Features

Section 2 – System Requirements & Features outlines the requirements that the software is based on (domain) and the main features that have been implemented and/or modified along with their purpose. Greater detail of these system requirements and features will be explained in the next sections.

## 2.0 RMI TO CORBA Architectural

The RMI application must be adapted so that it contains the necessary object adapters, skeletons, stubs, and ORB objects and IDL interfaces.

## 2.1 Start Server

Starts the Station Servers by initialization the server Object Request Broker, which then creates a Portable Object Adapter which is turned into an Interoperable Object Reference which uniquely identifies the station server object on a remote CORBA server.

## 2.2 Register Server as a UDP Listener

During the process of the server startup, there will be functionality to have each server open up a UDP listening port using a thread separate from the main thread. This functionality will allow other server objects to make requests such as record count.

## 2.3 Register Server as a UDP Invoker

During the process of the server startup, there will be functionality to have the servers capable of invoking messages to each other using a UDP protocol; this will also run as separate threads from that of the main thread.

## 2.4 Create Criminal Record

Officers will have the ability to create a criminal record for the station that they belong to. **The badge ID of the officer will complement this feature**

## 2.5 Create Missing Record

Officers will have the ability to create a missing record for the station that they belong to. **The badge ID of the officer will complement this feature**

## 2.6 Get Record Count

Officers will have the ability to requests how many records exists on the server. The server for whom the officer belongs to will query other servers and find out the number of records added to its own and will display the result. **The badge ID of the officer will complement this feature**

## 2.7 Edit Record Status

Officers will have the ability to edit the status of an existing record. **The badge ID of the officer will complement this feature**

## 2.8 Transfer Record

Officers will have the ability to transfer a record from their station to another station. Upon successful transfer of the record from station A to station B, station A will no longer hold that reference while station B will hold a the record **with a new identification number reference which is normalized to its process**.

1. Important and Difficult Parts

Section 3 – Important and Difficult Parts describes the important parts of this application along with the difficulties that exist.

## 3.1 Important Parts

This assignment took a lot longer than the RMI assignment did believe it or not. Switching from one architecture to another is a big deal even if RMI was implemented perfectly it still doesn’t prevent the programmer from having to redo a lot of work. For example, my RMI application contained references in the Interface to enum objects and not primitive types, therefore when I created my IDL I have to create inside of it an enum, and then have that generated enum object referenced throughout the application. Furthermore I had to turn my Date parameter into a long parameter and then do conversions and checks on the server which also took a bit more time to do.

It was crucial for me to go through the entire application and see how I can abstract my code so that reusability, portability, and cohesion could be maximized. I ended up handling the upper bounds of the Record ID in Build #2 and I also added a method to normalize a given IRecord so that it would be assigned a new ID with respect to its current values (First name, last name, etc).

## 3.2 Difficult Parts

The hardest part of my application was definitely the UDPListener and UDPInvoker structures that I set up. The idea is that each StationServer will register a UDPListener on a separate thread, and the clients will invoke actions on the CORBA server. Whenever interaction between two servers is needed, a UDPInvoker object will be created and will perform a given action. For a UDPInvoker to perform a given action, it created a ChannelPacket object which contains the operation to perform (RECORD\_COUNT or TRANSFER\_RECORD), and a payload Object. This object gets serialized and is sent over to the listener, which deserializes the object back into a ChannelPacket and then extracts the operation to perform. Based on the desired operation it will handle the payload data accordingly and will then send back information after this has been performed.

1. Architecture and Data Structures

This section explains some of the architectural decisions that were made for this assignment, along with the data structures used, and some talk about the synchronization we performed for the multi-threaded portions of the code.

## 4.1 Architecture

The architectural style that’s used is a client-server 2 tiered model. The client is the officer which calls certain functions that are referenced by remote stub objects. These look like regular method invocation calls however the actually implementation is done on the server side.

The solution has 4 projects.

1. Client
   1. Holds all of the client classes such as OfficerClient and Officer.
2. Server
   1. Holds all of the server classes such as StationServer and Record.
   2. Holds the implementation repository information for the implementation of the CORBA IDL.
3. Common
   1. This holds common constants used throughout the application such as the UDP port numbers that the stations register on.
   2. This holds the corba package which includes the compiled IDL’s, client side stubs and server side skeletons.
4. Tests
   1. Holds all of the unit tests, currently they are the record tests. Because of the time given we were not able to implement a proper unit testing framework.
   2. These tests have been adapted to work for the CORBA architectural changes

The actual running of these projects are very simple.

1. Run the StationServer.java by compiling and running like you normally would from Eclipse
   1. The Server project will now have a logs folder which shows the logs of each station server
   2. Each server is now listening on a specified UDP port for incoming requests
   3. Each Station Server object is now represented through an IOR file which is located in the /bin of the Common project inside of the /\_ior folder.
2. Run the OfficerClient.java
   1. The client will generate a folder called logs from the Client project based on the selected officer.
   2. It uses the IOR files to narrow down functions called to a specified interface repository.

The design patterns used for this assignment are

1. Factory Design Pattern, used to expose an interface for referencing records (IRecord) and a static factory creator which allows the server to create different types of IRecord objects, also checking that the passed information for the record is correct such as status and last name.
   1. For the CORBA assignment I added a function that accepts an IRECORD object and returns a normalized IRECORD object for the ID.
2. Dependency Injection, used on Officer (IOfficer) and Record (IRecord) to ensure that the runtime engine makes the decision of which class is assigned, also it makes coding a lot easier.

## 4.2 Project Organization

As mentioned above, the solution contains four projects which can be seen from the screenshot below.

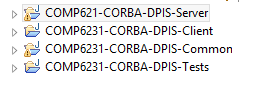


Figure the solutions structure consisting of four projects

## 4.2.1 General Project Structure

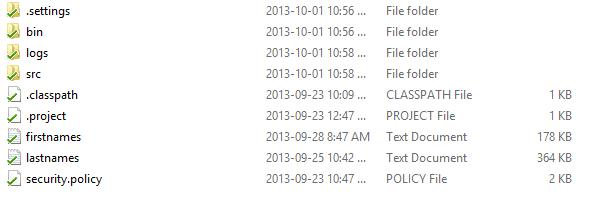


Figure Client organization

The organization of the client is very simple to understand. The firstnames and lastnames are files that hold thousands of different names that are used for the automated tests. These names all have an alphabetic capital first letter, however the system ensures that the last names first character is a letter and is uppercase; the letter need not be A-Z so long as it is not numerical.

The logs folder holds all the client side loggings. If two officers use the system and one is from SPVM and the other is from SPL, then there will be two log files, one labeled SPVM10000 and SPL10001. All client and server side log files show the date and time that the operation was performed along with a detailed message about the operation and in most cases its values or new values. The server logs work similarly to that of the client however there only every exists 1 log per station, and it holds details about receiving messages from other servers when UDP calls are invoked, and also mentioned that officers are contacting it.

## 4.3 Data Structures

Not many data structures were needed for this assignment. The assignment uses a Hashmap to store key, value pairs. Someone would look up the key, and would be returned a reference to an ArrayList of IRecord objects. Using ArrayList we are able to allocated memory dynamically based on when the application needs to store data. \*\* It is important to note that the Hashmap does not initialize to 26 characters, it instead waits until a new entry needs to be put in, which we will discuss in the next subsection when discussing synchronization techniques

**Example: HashMap<Character, List<IRecord>> map = new HashMap<Character, ArrayList<IRecord>> <IRecord>>();**

Furthermore, ArrayList was used to store Threads so that we could join them and prevent multiple threads from running on the main thread. The code below is a snippet used for retrieving the record count of all the sibling StationServer objects

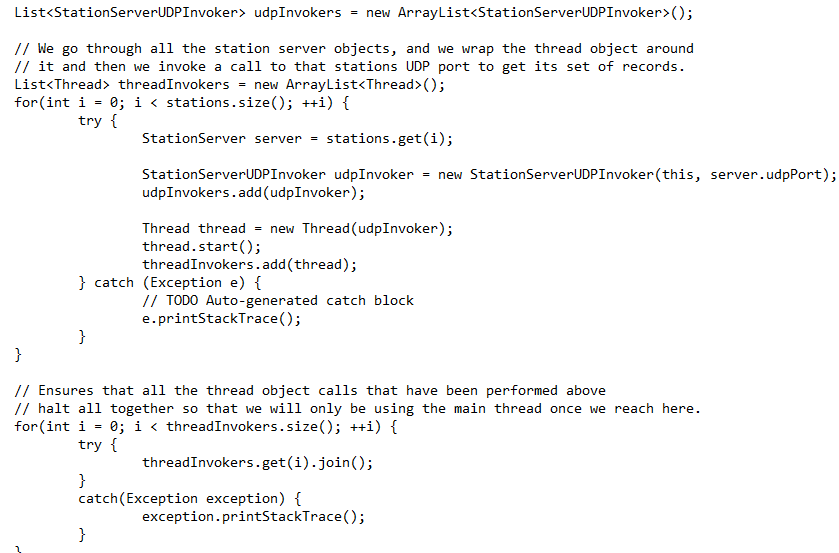


Figure Example showing how to create thread invokers and wait (join) them into one single thread before continuing

## 4.4 Concurrency Techniques

The application uses synchronized (object) throughout the application. It was decided that no function would be entirely blocked because it never required it, just a portion of the code such as unique record Identifiers needed it using synchronized (this), and in the case where we had to block on the hashmap, we would only block on the entire hashmap when inserting a new key since this required the new ArrayList of IRecord objects to be inserted as well.

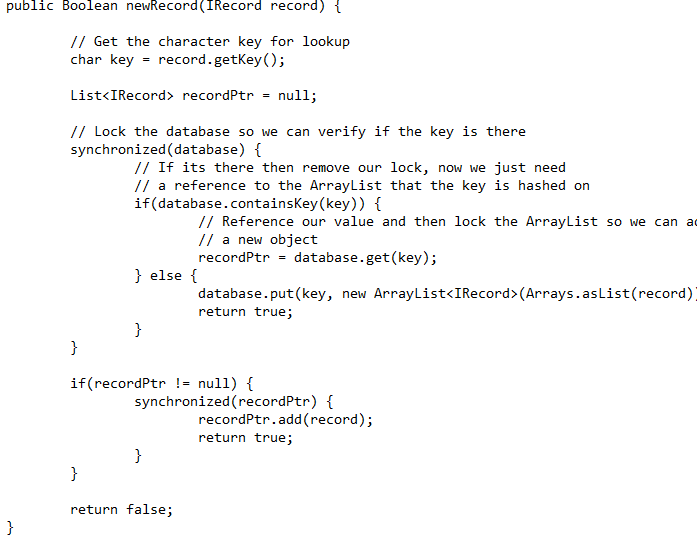


Figure Snippet for inserting a new record

As can be seen above in Figure 4 for example, we lock the database object to determine if the key already exists and we just need to put the object in it, if it doesn’t exist we lock the entire object and perform the insertion, else if the lookup key (first character of last name), then we just reference and lock the ArrayList<IRecord> value that is mapped to it by the key and then do our insert operation.

Another place that used synchronization techniques was in the GetRecordCount feature. When a StationServer boots up for the first time it registers itself uniquely onto a UDP Listener which is done with a thread as such:

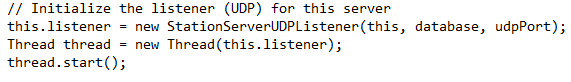


Figure Snippet of a StationServer registering on a UDP port

After this is done and the log file displays that the station serve is registering on port xxxx, a client can call the GetRecordCount which invokes three threads that contact that callee StationServer’s siblings and then calls itself on its on server for the benefit of code reusability. After the join() is done on all three threads we have a string that represents a concatenation of the messages of all three threads which is in the form of ServerName: NumberOfRecords, ….

This can all be seen in the StationServer.class, StationServerUDPInvoker.class, and StationServerUDPListerner.class.

1. Transfer Record & Server-Server Communication

The purpose of the CORBA assignment was to turn the RMI architecture into the CORBA architecture, normalize the interface into an IDL interface, append badgeID to every function implemented in RMI, and implement the new TransferRecord. TransferRecord was a hard function to implement because it implied making a decision to expand on server-server communication based on the getRecordCount function done in RMI.

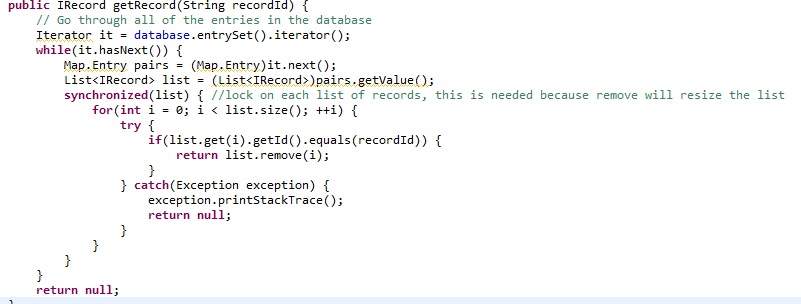
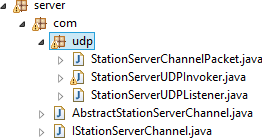


Figure 6 get the record from the station server and remove it from the hash table

The side effect of doing the above will be that the key will not be removed, however this isn’t that big of a deal and in fact will ensure consistent performance throughout the application being that the server is an “always on” application.

The difficult part was the server-server interaction; there had to be a way to distinguish between getRecordCount and transferRecord.



AbstractStationServerChannel holds serialize and deserialize functionality and a reference to a StationServerChannelPacket which packages an operation command (GET\_RECORD\_COUNT, TRANSFER\_RECORD). It is then sent between StationServerUDP Invoker/Listener classes.

1. Unit Tests

The following are the unit tests and testing formulas used for this application.

1. A first name must not be null or empty
2. A last name must not be null or empty or contain a number as its first character
3. A Criminal Records status can only be ON\_THE\_RUN or CAPTURED
4. A Missing Records status can only be MISSING or FOUND
5. All Records generated hold unique identification numbers
6. The officer test for uniqueness would have been performed however it uses the exact same code as Records and it has a lot of side effects that would not be well suited for unit testing, therefore **based on the time given** this could not be properly extracted into a unit test.
7. Changing the status of a record must be changed to a valid status

\*\* The following could not be put into unit tests

1. A record cannot be transferred to its own station
2. Dates much be valid
3. RecordID of a report must be valid
4. OfficerID of an officer must be valid
5. Synchronization on the list of records

\*\*FOR ANY MORE QUESTIONS OR CONCERNS PLEASE EMAIL ME AT [THEDANNY09@GMAIL.COM](mailto:THEDANNY09@GMAIL.COM) OTHERWISE I WILL ASSUME THAT EVERYTHING IS CLEAR\*\*