COMP 6231 Technical Report

submitted in partial fulfillment
of the requirements for the
Assignment 2

by
Manfred Ramón Díaz Cabrera
Student ID: 40014085

Montréal, Québec
07/05/2016

OVERVIEW

The assignment demands the extension of the Staff Management System implemented in Assignment 1 by adding a new operation that allows the clinic manager to transfer records between different locations. Besides, a parameter identifying the client manager who is performing the request should be add to all five operations.

In this Assignment the underlying IPC technology must be changed from Java RMI to CORBA using Java IDL.

GOALS

As per assignment specification, the following items delimit the scope of this work to:

1. Write the Java IDL interface definition for the modified DSMS with all the 5 specified operations.
2. Implement the modified DSMS. You should design a server that maximizes concurrency. In other words, use proper synchronization that allows multiple managers to correctly perform operations on the same or different records at the same time.
3. Test your application by running multiple managers with the 3 servers. Your test cases should check correct concurrent access of shared data, and the atomicity of ***transferRecord*** operation.

SPECIFICATIONS

System requirements has been elicited from the assignment document and represent using *Behavior Driven Development*[1] notation, a simple representation for depicting users' stories and non functional requirements in a concrete way. These sections establish a common understanding for development, testing and deployment purposes.

Non Functional Requirements	
NFR1: Manage Records Transparently **As** a Client, **I want** Clinic Manager manage the records in their clinic only **So I** have control over the accessed information	**NFR2: System Auditing** **As** a Client **I want to** log all actions users perform **So I can** plan security accordingly
NFR3: Platform should be Java **As** a Client, **I want** the system implement in Java **So I can** have multi-platform capabilities	**NFR4: Java RMI for C/S Communication** **As** a Client, **I want** the communication between client and server uses Java RMI **So I can** simpler RMI invocation
NFR5: IP Stack for S/S Communication **As** a Client, **I want** the communication between servers uses low level TCP/UDP **So I can** decouple the servers	**NFR6: Deployment for three locations** **As a** Client, **I want** the system deployed in three locations **So I can** have a performance through resource sharing
NFR7: High Concurrency Degree **As** a Client **I want** the system designed to maximize concurrency **So I can** have a higher performance while executing the operations	**NFR8: Clinic Manager Identification** **As** a Client **I want** Clinic manager identified with their location plus a unique four-digits number **So I can** keep track of the operations performed

Functional Requirements	
FR1: Create Doctor Record **As a** Client Manager **I want to** create a doctor related record	**FR2: Create Nurse Record** **As a** Client Manager **I want to** create a nurse related record

[1] Dan North, Introducing BDD, Online: https://dannorth.net/introducing-bdd/, Accesed: May 2016.

So I can keep track of doctor's staff status	**So I can** keep track of nurses staff status
FR3: Get Record Count **As a** Client Manager **I want to** know all doctors or nurses records in all locations **So I can** keep track of the staff status	**FR4: Edit Record** **As a** Client Manager **I want to** edit a record field **So I can** keep the staff information updated
FR5: Doctors Record Id → [FR1] **As a** Client **I want** the doctor's records identified with the "DR" prefix plus a five digits number **So I can** properly identify each record	**FR6: Nurse Record Id → [FR2]** **As a** Client **I want** the nurses records identified with the "NR" prefix plus a five digits number **So I can** properly identify each record
FR7: Transfer Record **As a** Client **I want to** be able to transfer a record **So I can** keep track of my staff movements	

SOLUTION

Domain Model

In order to fulfill the functional requirements, a domain model was designed in Assignment 1 reflecting the major concepts in the discussion domain for the assignment as depicted in **Figure 1**. No major changes have been introduced in the domain model to fulfil the new requirements.

Architecture

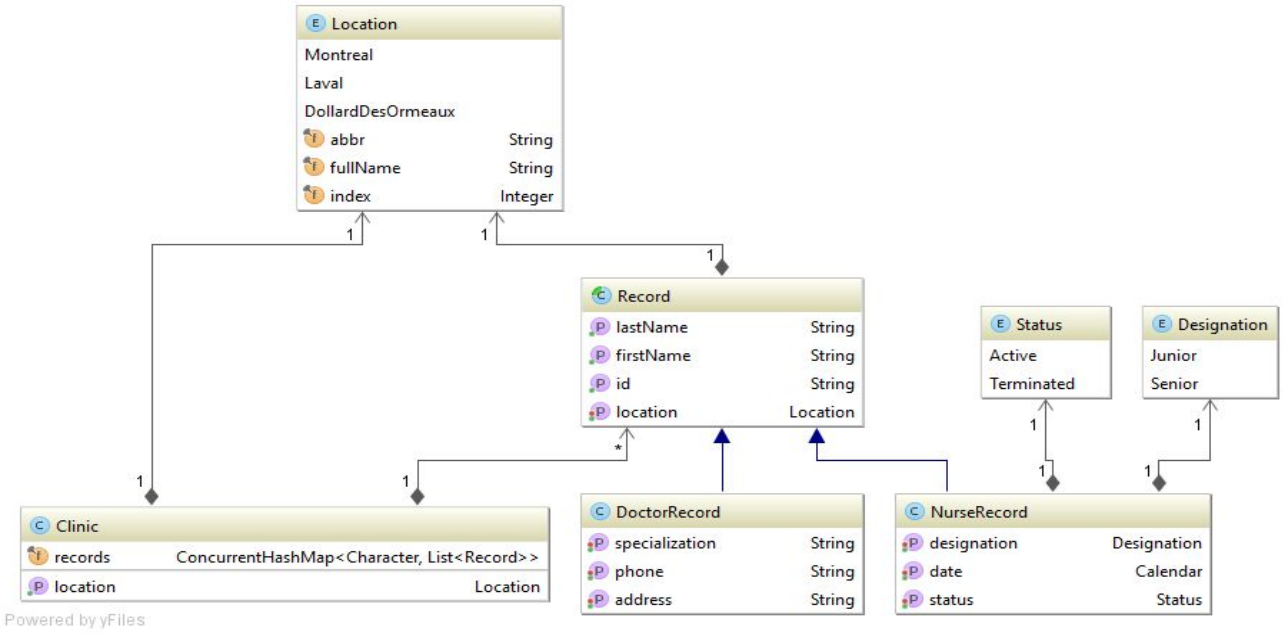
Using as a starting point the solution presented for Assignment 1, some changes have been carried out in order to accommodate the new requirements in the previous architecture.

In the server side of the solution a new model has been presented in order reuse the existing implementation with both IPC technologies. All implemented features are now abstracted into a *ClinicServer* concept from which *ClinicServerRMI* and *ClinicServerCORBA* inherits, leaving only translation features (reply/response messages conversion) to these descendants, maximizing the

reuse. A similar solution has been implemented on client side, where there exists two client endpoint managers, one for CORBA and one for RMI which have allowed the reuse of the previous client side implementation and test suites.

Given all this, a new partition of the system into components has been implemented resulting in the addition of two new components (Table 1) *rmi* and *corba* which now host the respective implementation of the server feature in each technology.

Figure 1. Domain Model



Client-Server Communication

As per explicit requirements, Java IDL CORBA functionality was used for client-server communication. Java IDL offers several options for generating interoperability code. Given that our current *ClinicServerCORBA* already inherits from ClinicServer implementation, the Java IDL Tie Model[2] was implemented given the provided ability to use the actual servant object implementation as a delegate without inheriting from the generated Portable Object Adapter.

Table 1 Component and Responsibilities

Component	Brief Responsibility Description
shared	Contains all cross-cutting models, concerns (security, base logging, exception handling) and representations (interfaces, messaging).
server	Encapsulates all server related features: clinic server implementation, UDP

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlTieServer.html

	multicast server-server communication.
client	Encloses the client UI, user Access Transparency Layer (Session class), both CORBA and RMI can be selected as IPC technologies.
rmi	Implements RMI communication logic from Assignment 1
corba	Contains CORBA server side implementation.

Generated from the IDL definition presented in **Figure 2**, a model for CORBA interoperability was designed and implemented (**Figure 3**). Worth noting the creation the *CorbaJvmManager* concept, which main responsibility is to get rid of the intrinsic complexity of handling Java CORBA implementation by providing a much simpler approach for registering servants and resolving names.

Figure 2 ClinicServices.idl

```
module interop {
    struct Message {
        string Method;
        short Status;
        string Content;
    };
    interface ClinicService {
        Message createDRecord (in string managerID, in string firstName, in string lastName, in string address,
                               in string phone, in string specialization);
        Message createNRecord (in string managerID, in string firstName, in string lastName, in string designation,
                               in string status, in long statusDate);
        Message getRecordCounts(in string managerID, in string recordType);
        Message editRecord(in string managerID, in string recordID, in string fieldName, in string newValue);
        Message transferRecord(in string managerID, in string recordID, in string remoteClinicServerName);
    };
};
```

In the aspect of CORBA names resolution and given the location-aware nature of the server implementation ***[NFR6]***, each server instance has been promoted to the CORBA Name Service using a running location *[server.location]* even if the *ClinicServices* implementation is unique for all the instances. This configuration parameter allows each CORBA servant object to be published with a namespaced name e.g: "*ca.concordia.encs.distributed.mtl*", making each server endpoint unique inside the CORBA Name Service running instance.

Per ***[NFR1]*****,** the system must provide users with access and location transparency. The concept of Session implemented for Assignment 1 has been reused for implementing a transparency layer for client code. Also, all logging aspects implemented in Assignment 1 have been kept and, by making use of the new ***managerId*** parameter added to the *ClinicServices* interfaces, servers are now able to log the user that is invoking in each request.

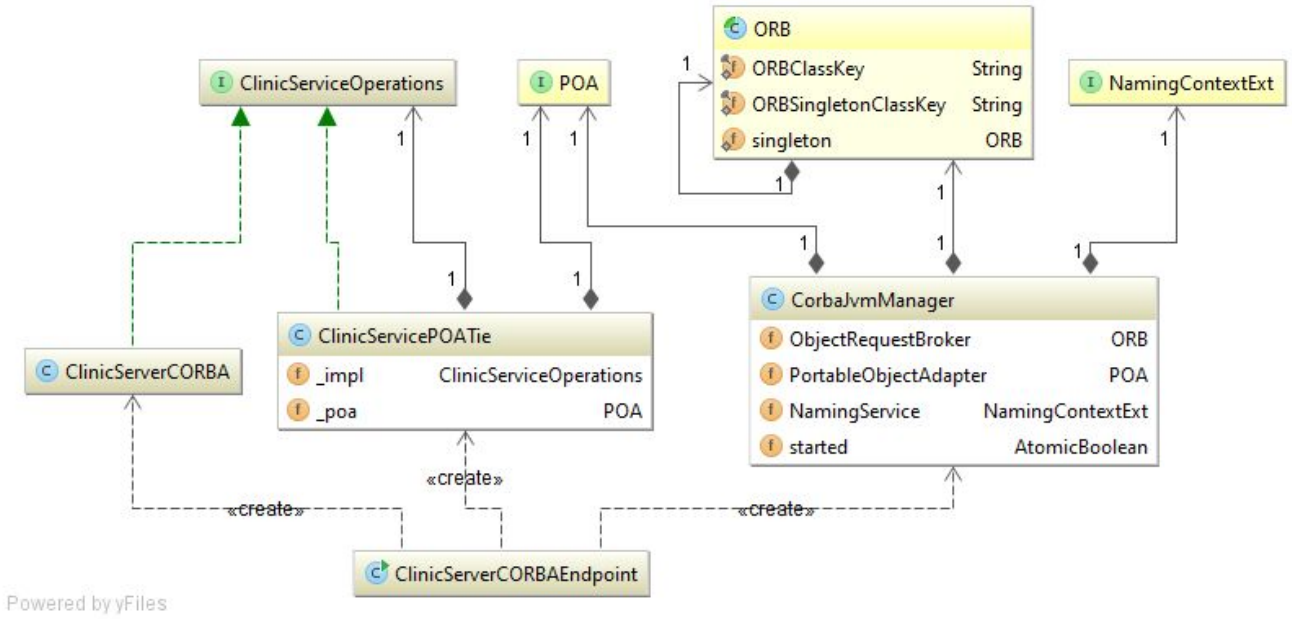


Figure 3 CORBA Server Side Tie Model

Server-Server Communication

Figure 4 displays the model implemented to cover multicast UDP communication between server instances required for Assignment 1. The majority of this model have been fully reused in order to add the new message that is required to be passed: *Transfer Record* message. For this purpose only two changes have been introduced: the new message consumer *TransferMessageConsumer* and a new field to the *IdentifiableMessage*, the *To* field.

The new *To field* now provides the ability to specifically address a server inside the cluster multicast communication, a requirement implicitly imposed by the **[FR8]**. *TransferMessageConsumer* consumer is the class that now makes the server implementation aware of the invocation needed for exchanging records between different nodes.

Development environment

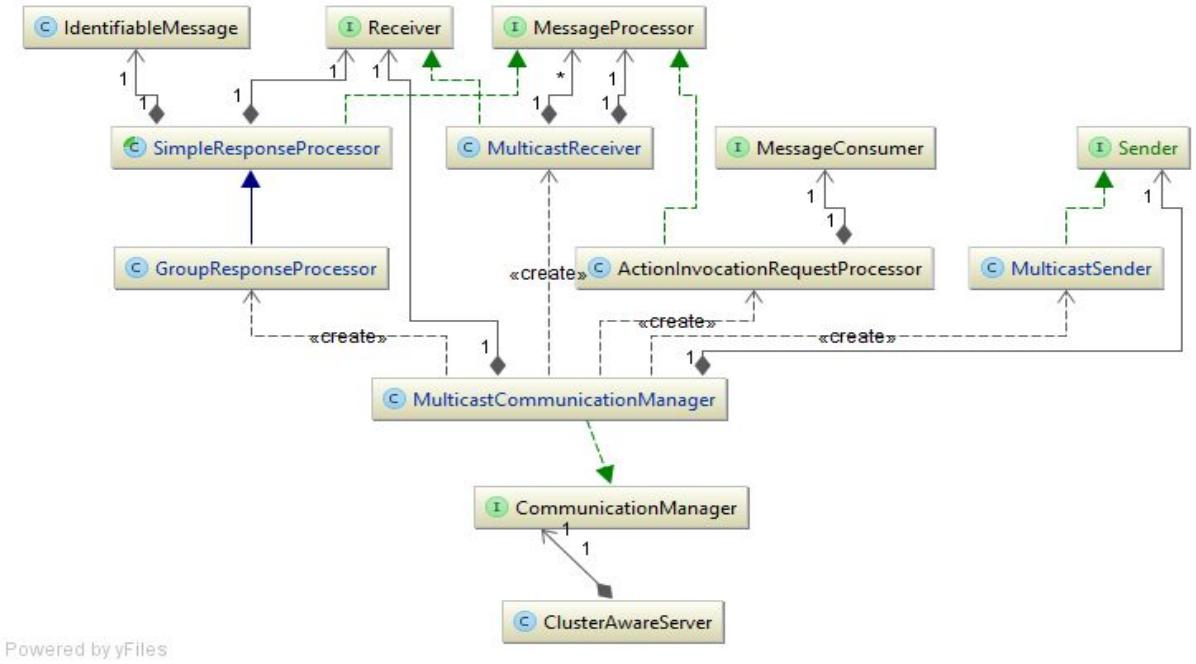
For developing purposes several tools have been used to facilitate the whole SDLC. The tools listing can be found in **Table 3** that depicts the tools, versions and the purpose inside the development process.

Testing

As an integral part of the development process, several tests were created for ensuring the software quality and the consistency of the implementation with the requisites. For ensuring the correct

operation of the domain model classes, several unit test were created covering the main system requirements like: *createDRecord, createNRecord, getRecordCounts, editRecord.* Scenarios where the system should fail were also designed for test: *editRecordMustFailWhenForbiddenField*, *editRecordMustFailWhenIncorrectValue.*

Figure 4 Multicast Communication Model



Also an integration test was create from the *ConsoleUI* class, in the ***client*** component, that tests the multiple and concurrent requests to the server cluster. A fixed thread pool was used for executing a configurable -via command line argument- number of threads that concurrently request the servers. This implementation heavily relied on Apache Commons *RandomStringUtils* class, that provided an easy way to generate a huge amount of random data (*first name, last name, addresses*).

Table 3 Tools Listing

Tool	Version	Purpose
IntelliJ Idea	15.04	Integrated Development Environment
Gradle	2.13	Build System
junit	4.12	Test system

CONCLUSIONS

A few extensibility points have been left in the solution for future implementations and enhancements. Along with other considerations, like improving security (proper session management and reuse) and logging capabilities, these are the main place were some optimizations can be carried on to improve several aspects of the system like scalability and failure handling.