Manfred Ramon Diaz Cabrera
315-3475 Saint Urbain
Montreal, QC H2X2N4
(514) 248-8457

COMP 6231 - Assignment I

31st May 2016

OVERVIEW

The assignment demands the implementation of a Staff Management System. The system is going to be used by Clinic Managers to manage the information regarding doctors and nurses in three locations: Montreal, Laval and Dollard-des-Ormeaux. The system should manage two types of records: doctors and nurses. For all records some operations for retrieving information must be allowed to perform to clinic managers in a transparent way, without knowledge of resource location. Also all servers and clients subsystems should main a detailed log of the actions performed in the system for auditing purposes, recording pertinent and as much information as possible.

As an specific Non Functional Requirement, all the implementation must be Java based, employing Java RMI for client/server communication purposes and lower level protocols (TCP/UDP) for server/server messages exchange. Concurrency must be taken as the main challenge for this implementation.

GOALS

1. Write the Java RMI interface definition for the *ClinicServer* with the specified operations.
    a. Implement the *ClinicServer*.
2. Design and implement a *ManagerClient*, which invokes the clinic's server system to test the correct operation of the DSMS invoking multiple ClinicServer (each of the servers initially has a few records) and multiple managers.

SPECIFICATIONS

System requirements has been elicited from the assignment document and represent using *Behavior Driven Development*[1] notation, a simple representation for depicting user stories and non functional requirements in a concrete way. This 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

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

Functional Requirements

FR1: Create Doctor Record **As a** Client Manager **I want to** create a doctor related record **So I can** keep track of doctors staff status	**FR2: Create Nurse Record** **As a** Client Manager **I want to** create a nurse related record **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 doctors 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

SOLUTION

Domain Model

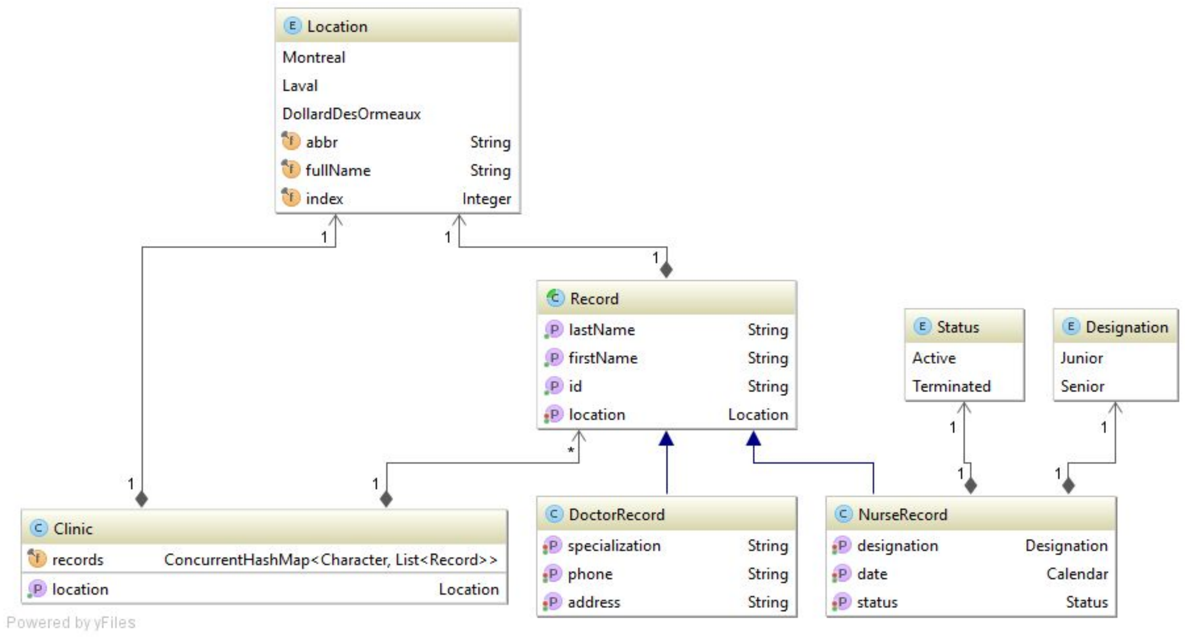
In order to fulfil the functional requirements, a domain model was designed reflecting the major concepts in the discussion domain for the given assignment as depicted in Figure 1. Due notice the use of Java's ConcurrentHashMap[2] implementation, as per several performance benchmarks executed[3] ConcurrentHashMap outperforms most of Java concurrency aware collections by fragmenting the Hash Map representation. Also a synchronized list has been used for storing records inside each key thus maximizing performance and concurrency while accessing the records stored in the server.

Some boolean-representable values as nurses *active* and *designation* properties were implemented using enumeratives. This way provided a twofold advantage as a good path for achieving extensibility and a strong typed validation using Java's *Enum.valueOf()*.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

[3] Performance benchmarks of Java Concurrent Lists. Available on: http://crunchify.com/hashmap-vs-concurrenthashmap-vs-synchronizedmap-how-a-hashmap-can-be-synchronized-in-java/

Figure 1. Domain Model



Architecture

After modelling the domain of discussion, the overall system architecture was depicted using the main non-functional requirements and some implicit derived from cross-cutting aspects. Three components were defined to encapsulate the different features implementation. Each component name and core responsibility are depicted in Table 1. The system was completely implemented using *Java 1.8 SDK*, taking advantage of many of the improvements offered by this new version of the platform in concurrency control and Java RMI library.

Most configuration parameters has been established using JVM properties. This properties values can be passed as argument to any running JVM instance and then can be collected at runtime offering a high degree of flexibility. This configuration is stored in *ServerContextHolder, ClientContextHolder* and *EnvironmentContextHolder* classes in each component.

Communication in all forms is designed using the *Message* abstraction (method, status, content) in a resemblance of the HTTP message exchange protocol. This facilitates the homogeneity of the communication platform in both client-server and server-server manners.

Logging

There were a few concerns that had to be address in the solution. The first one is related to logging capabilities of both the server and the client implementation. The server implementation can be a simple log but the client had to have the special feature of a log file for each authenticated user. Two mainstream logging libraries were evaluated for providing logging capabilities: *Apache log4j 2*[4] and *Logback*[5]. The sifting appender together with the ability to have a Managed Diagnostic Context (MDC) per logging action offered by Logback made it the preferred logging library[6]. Two XML files: *logging-client-config.xml* and *logging-server-config.xml* were created for Logback Joran-based runtime logging configuration.

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, RMI endpoint, UDP multicast server-server communication.
Client	Encloses the client UI, user Access Transparency Layer (Session class)

Client-Server Communication

As per explicit requirements Java RMI library was used for client-server communication. Since Java 1.5 the use of precompiled stubs and skeletons has been deprecated[7]. The approach implemented takes advantage of the on-the-fly stubs and skeletons feature.

Given the location-aware nature of the server implementation ***[NFR6]***, each server instance should be configured using a running location *[server.location]* even if the *ClinicServer* implementation is unique for all the instances. This configuration parameter allow each server RMI object to be published with a namespaced name e.g: "*ca.concordia.encs.distributed.mtl*", making each RMI endpoint unique inside a RMI registry and providing a uniform and decoupled way of service location from clients trying to access a location-specific server.

User Transparency Layer

Per ***[NFR1]***, the system must provide users with access and location transparency while managing the staff on their location. Using the established user id, the server namespaced name on RMI registry, a concept called *Session* is created, which transparently links the user with the server instance associated with its current location.

[4] http://logging.apache.org/log4j/2.x/
[5] http://logback.qos.ch/
[6] http://logback.qos.ch/manual/loggingSeparation.html
[7] http://docs.oracle.com/javase/8/docs/technotes/tools/unix/rmic.html

After this moment, every user invocation is proxied by the Session object, which also logs all user activity in its user-dedicated log file.

Server-Server Communication

Server-server communication is a requirement derived from ***[NFR5]*** and ***[FR3]***. Given the requirements two solutions were evaluated to accomplish this task. The first one can be a connection between servers using TCP. While this type of communication is more reliable and robust may add some undesired effects like configuration overhead, coupling, etc.

The servers in three locations can be considered a cluster, so in this case most of the messages directed to any of the members of the cluster can be considered a group message so a form of indirect, group-based communication should be employed. From the several solutions availables for group communication, multicast UDP is the one that adapts better to the requirements. Sending messages to a multicast address is a good way of decoupling the servers and given that in this assignment network can be considered reliable, using UDP is an acceptable approach because a more reliable layer on top of it should not be implemented.

The **Figure 2** display the model implemented to cover multicast UDP communication between server instances. A new concept has been introduced over the preexisting *Message* abstraction: *IdentifiableMessage*. An identified message extends the *Message* notion by adding the Id and ReplyTo numeric representation. Given that in a multicast group every subscribed server will receive all messages sent, there should exist a way to discriminate the which messages are intended for the server and which are not.

The message ID has the bit structure represented in Table 2. This offers -or limits depends on the perspective- the cluster size to 64 nodes. The max number of message is 134,217,726. A rolling strategy has been implemented for those cases where this number is reached. This way we can identify when a message is generated by a particular server instance -check *Location.index* feature.

Table 2 Identifiable Message Id

SERVER_ID	MESSAGE_ID
6 bits	26 bits

The ReplyTo is an numeric value with the same structure of the message id which represents the message id the message is responding to, giving the server the capacity to know when a multicasted message is a response to a message it has sent to the group.

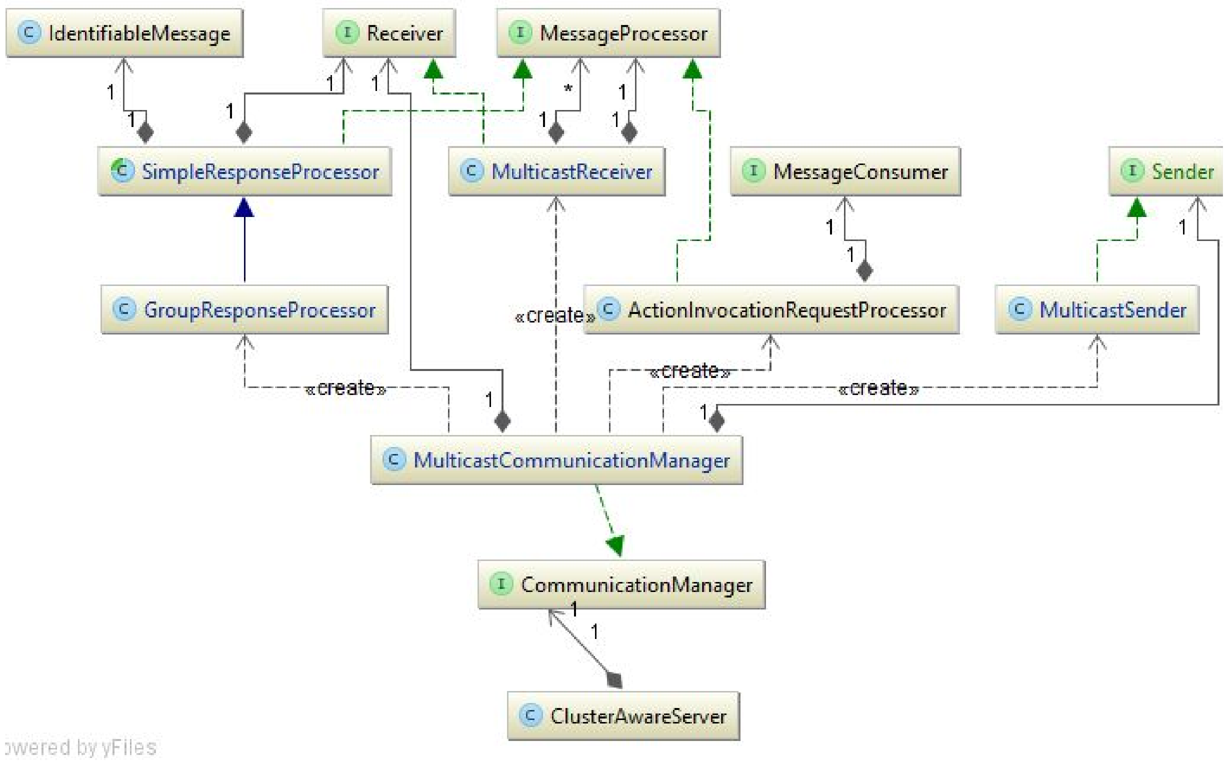
As depicted in Figure 2, an abstraction of in-cluster server *ClusterAwareServer* has been created to manage the clusterization notion of the server instance. Each cluster-aware has a

CommunicationManager abstraction. This extensibility option, filled by the current multicast implementation *MulticastCommunicationManager,* allows the server-cluster communication and it is open for future extensions or modifications (change of paradigm, protocols).

The multicast solution implemented have a multithreaded receiver which is listening a promiscuous mode (receiving everything) to the multicast address configured in [cluster.address] JVM property. In order to filter the messages, the *MulticastReceiver* instance stores serveral *MessageProcessor* implementation that are executed for every message received.

An example of such processor is *GroupResponseProcessor.* This class holds the logic for collecting chorus responses for messages that expects one or more response from the rest of the cluster nodes. There is also a *ActionInvocationRequestProcessor* class that mimics a remote invocation directed to the cluster nodes. Both classes are mainly used in the implementation of **[FR3].**

Figure 2 Multicast Communication Model



Threading capabilities has been implemented for the multicast sending and receiving so all the operation are performed in a non-blocking way. In order to *gracefully degrade* when the number of requests exceeds server threading capabilities a fixed thread pool provided by Java platform *Executors.newFixedThreadPool* has been used for scheduling all the concurrent tasks.

Another key concern addressed while implementing server-server communication was message serialization. Although binary serialization can be considered given the homogeneity of all the server (Java as a middleware), the proposed solution uses JSON as an external data representation. JSON serialization is carried on by google-gson[8]. Besides been a more friendly and comprehensible representation, JSON can be used for resolving homogeneity issues inside distributed systems, so choice is an openness feature for future implementations.

Development environment

For developing purposes several tools have been used to facilitate the whole SDLC. The tools listing can be found in Table 3.

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.

Failure handling is a major area were some changes can be introduced. While the system currently has a centralized exception manager various tolerance and recovery techniques can be implemented to increase the system robustness, but they are out of the reach of the present assignment.

Recalling the goals of this assignment, it is possible to affirm that they have been fulfilled by the current system implementation.

[8] https://github.com/google/gson