Implementation of an agents based system for anonymous medical database retrieval

Imanol-Mikel Barba Sabariego

Grau en Enginyeria Telemàtica, ETSETB, UPC

July 3, 2016

This page intentionally left blank

Abstract

The aim of this thesis is to describe and develop in detail the implementation of a Dynamic Health Data Aggregation System for research purposes in the framework of ISyPeM2.

The motivation of this work responds to the need of simple, efficient and secure systems to perform medical research using patient sensitive data, which for the time being is handicapped by several requirements such as previously established agreements between institutions, heterogeneous databases and privacy issues. This thesis aims to document the implementation of a system that addresses these issues.

To implement the system, we used the principles of P2P networks for discovery of data sources (the nodes in the system), the functionality of a Multi-Agent System for coordination between data sources and the use of anonymization techniques to preserve the patient's privacy.

The thesis first examines the motivation and application for such systems, as well as the concepts needed to understand the components that conform the system and then it will proceed to present the state of the art for similar or related systems and technologies being developed for the same purpose. Afterwards, the components in the implementation will be detailed. After that, there is an in-depth explanation on how the system works and the components it is comprised, along with other implementation details, as well as a description of the development environment.

In the final chapters, the results from the tests performed and the conclusions of this work will be presented, and in the appendices, there is a description of the datasets used for the performance tests, as well as the last review of the implementation code.

Abstract

El objetivo de esta tésis es describir con detalle el desarrollo e implementación de un Sistema de Agregación de Datos Médicos para la investigación en el marco de ISyPeM2.

La motivación de este trabajo responde a la necesidad de sistemas sencillos, eficientes y seguros para realizar investiación médica utlizando datos sensibles de los pacientes. Actualmente, estos sistemas estan limitados por varios requerimimentos como acuerdos preestablecidos entre instituciones, bases de datos heterogéneas, y problemas de privacidad. Esta tésis documenta la implementación de un sistema que trata estos problemas.

Para implementar el sistema, utilizamos los principios de redes P2P para descubrir las fuentes de datos (los nodos del sistema), la funcionalidad de Sistemas Multi Agente para la coordinación entre fuentes de datos y el uso de técnicas de anonimización para preservar la identidad del paciente.

Esta tesis primero indaga en la motivación y la aplicación para estos sistemas, como también en los conceptos necesarios para entender los componentes que conforman el sistema, luego se procederá a presentar el *State of the art* para ver tecnologías similares o relacionadas que estén siendo desarrolladas con el mismo propósito. A continuación entraremos de cabeza en los componentes de la implementación. Después hay una explicación detallada de como funciona el sistema y los componentes que lo forman, junto con otros detalles de la implementación y una descripción de el entorno de desarrollo.

En los últimos capítulos, se presentarán los resultados de los test que se han llevado a cabo. En los apéndices se incluye una descripción de los *datasets* utilizados para los experimentos, junto con una copia de la última revisión del código de la implementación final.

Abstract

L'objectiu d'aquesta tesi és descriure amb detall el desenvolupament e implementació d'un Sistema d'Agregació de Dades Mèdiques per a la recerca en el marc de ISyPeM2.

La motivació d'aquest treball respòn a la necessitat de sistemes senzills, efficients i segurs per realitzar investigació mèdica fent servir dades sensibles dels pacients. En aquests moments, aquests sistemes són limitats per varis requeriments com acords preestablerts entre institucions, bases de dades heterogenies i problemes de privacitat. Aqueta tesi documenta la implementació d'un sistema que adreça aquests problemes.

Per implementar el sistema, utilitzem els principis de xarxes P2P per descobrir les fonts de dades (els nodes del sistema), la funcionalitat de Sistemes Multi Agent per la coordinació entre fonts de dades i l'ús de tècniques d'anonimització per preservar l'identitat del pacient.

Aquesta tesi primer examina la motivació i aplicació d'aquests sistemes, com també els conceptes necessaris per comprendre els components que el formen, després es procedirà a presentar el *State of the art* per veure tecnologies similars o relacionades que estiguin sent desenvolipades amb el mateix propòsit. A continuació ens capbussarem en els components de la implementació. Després hi ha una explicació detallada de com funciona el sistema i els components que el formen, junt amb altres detalls de la implementació i una descripció de l'entorn de desenvolupament.

En els últims capítols, es presentaràn els resultats dels test que s'han dut a terme. En els apèndixs s'inclou una descripció dels *datasets* que s'han fet servir pels experiments, juntament amb una còpia de l'última revisió del codi de l'implementació final.

Acknowledgments

I would like to express my gratitude to various individuals who helped me in the development of this thesis and without which I wouldn't have been able to carry on with this work.

First of all, my supervisor Josep Pegueroles, who from the very first moment guided me and trusted me with the development of this work, and whose invaluable help got me to have one of my most fulfilling personal and professional experiences abroad. Also thanks to my supervisor in HES-SO, Michael Schumacher-Debril, for giving me the opportunity to contribute to this amazing project.

I would also like to thank for their contributions and help to Alevtina Dubovitskaya (HES-SO), Visara Urovi (HES-SO), Stefano Bromuri (HES-SO) and Alex Olivieri (HES-SO). Also thanks to the wonderful people from AISlab and MedGift research groups: Sebastián, Ranveer, Roger, Alba, Oscar, Yashin, Pol, Manfredo, Antoine, Thomas, Fabian, Fabien, Damian, Michael, and of course the aforementioned, for the camaraderie and the time I spent with all of you. The time we spent together is a long lasting footprint that made me grow as a better person and that I will cherish for the years to come.

Una abraçada molt forta al despatx de Revistes del Campus Nord per haver-me acollit com un més i haver-me donat els millors anys de la meva estada a l'universitat. També als meus amics de l'UPC Campus Nord i de Sabadell, en especial: Josep A., Magí A., Elena A., Júlia A., Ignacio B., Alex C., Sergi D., Germán D., Alberto G., Toni I., Juan Francisco L., Alejandro L., Alejandro N., Ferran O., Ferran Q., Jordi S., Ivet S., Maria S. i Alex U.

Finalmente, a mi família, por confiar en mí estos años y haber estado a mi lado durante los tiempos buenos y los malos, que sean muchos más años. A mis padres y mi hermano Marcos, gracias de corazón.

Contents

Chapter 1

Introduction

Lately there has been numerous advances in the healthcare industry thanks to the development of new technologies in prevention, diagnosis and prognosis of illnesses. These advances owe their success to the integration of computerized systems for their ability to analyze data and assist healthcare professionals with more tools to maximize the effectiveness of the different treatments.

Nonetheless, there is still a lot of work to be done regarding medical data management. If one were to perform a transfer of medical data about a patient, there would be privacy issues to be faced, as the data are of the utmost confidentiality. Also, most medical centres do not regard, or do not value the need to have interoperability between various medical communities, partly because of the legal requirements for the data privacy are often a nuisance and also because occasionally, such matters don't fit into their business model and the revenue is not clear. As a result of this, there is no solid common grounds to achieve this in a global scale. Moreover the lack of a generally adopted standard for medical data exchange poses a problem, as interoperability requires a common schema for data exchange to be possible. Fortunately, this last aspect has already been approached by some, but not all medical centres as we will see later.

The development of the system here proposed is driven by the necessity to overcome the problems of privacy and data exchange between heterogeneous environments for research purposes. By improving the process of privacy-preserving medical data collection (its viability, speed and easiness) for research purposes, a great deal will be contributed to medical and pharmaceutical research. This is achieved by providing a system that automatically and securely aggregates the data, which is the exact purpose of the system here developed.

It is thus clear that the goal of this work is to develop a reliable, effective and easy to use system to aid researchers, while protecting patient privacy all along. The main milestones in the project are:

- Providing a secure mechanism for data exchange, aggregating useful data for research without compromising patient privacy.
- Developing a solution that is able to interoperate between various institutions without necessarily having preestablished agreements, but cooperating through negotiation instead.
- Building shared databases with the aggregated data for research purposes, whilst keeping the data both secure and useful for medical research.

The focus of this thesis is the implementation of a system designed to satisfy these goals, which is ISyPeM2[2]. ISyPeM2 is an ongoing project, one of its goals being developing a system that aggre-

gates medical data dynamically for research purposes, specially for Therapeutic Drug Monitoring [TDM]. The following diagram shows the system architecture:

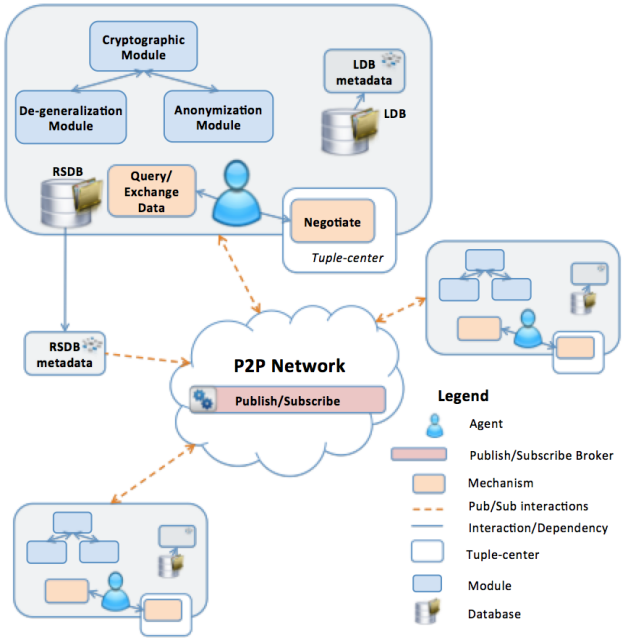


Figure 1.1: System diagram for ISyPeM2

In Chapter 2, to explain the challenges faced during the implementation of this system, the key concepts to justify some of the design and implementation decisions will be explained, so the reader can have a better understanding of the different systems employed for this purpose.
Though in terms of aiming for a solution that provides dynamic data exchange for research it may be, work in the area of medical record exchange is not novel. Consequently, in Chapter 3, we give an overview on existing technologies and solutions developed with similar goals to our system, along with some of the ongoing research in this field.
After this, in Chapter 4, the proposed framework for the system will be presented, which will be an overview of how the system works as a model.
Following that, in Chapter 5, the actual system implementation is thoroughly detailed, how each of the components are implemented, and which design decisions that we made, along with a functional description of the development environment.
Later on in Chapter 6, we explain the tests performed to benchmark the system against various factors (speed, effectiveness, negotiation capabilities, etc.), along with their results.
Lastly, in Chapter 7, the conclusions derived from this work will be discussed.

Chapter 2

Fundamentals

Considering the complexity of the system, it is due that some of its components are explained in-depth to better understand it, as well as some of the theory principles it is based on. In this chapter we will achieve this by introducing the reader into the basic concepts needed to comprehend the work in this thesis. We will explain in detail some eHealth elements like the Electronic Health Record [EHR] and some implementation-wise elements like Agents or Anonymization.

2.1 EHR

An EHR[3] is a sequential record of the patients health information, digitalized and compiled by the different health care systems of the various institutions he visits during his lifetime. An EHR can:

- Contain information about a patient's medical history, diagnoses, medications, immunization dates, allergies, radiology images, and lab and test results.
- Offer access to evidence-based tools that providers can use in making decisions about a patient's care.
- Automate and streamline providers' workflow.
- Increase organization and accuracy of patient information.
- Support key market changes in payer requirements and consumer expectations

It is the first approach to the maintenance of medical information by computerized systems rather than storing it into physical records, which essential to medical data processing and specially in record exchange, since it allows to be created, managed, consulted and modified by patient-authorized staff in several health communities, rather than a single institution.

The importance of EHR is to provide common grounds for systems to exchange medical information following a common format. The system here presented processes a collection of EHRs from the local system and provides anonymity to its fields as described in section 2.4 when data exchange between two nodes takes place.

2.2 Agents

Software Agents[4] are already well-known in computer science. It is a software component that acts on behalf of a user to achieve one or more goals and that is capable of exhibiting reasoning and/or learning to achieve such goals.

There are multiple types of agents, classified by several criteria: if they plan ahead how to achieve their goal (Planning agent) or if they take decisions based on reactions to their environment (Reactive agents).
They can also be classified depending on the environment they work: if they can relocate their execution into different systems they are mobile agents, and if they operate in a shared environment with several other agents, it's a Multi-Agent System [MAS].

In this implementation, our agents will be part reactive and part planning agents depending on the goal.

2.2.1 MAS

Since our system is designed to run in an environment comprised of several nodes, each with their own agents, this system is a MAS. A MAS is an environment in which various agents communicate between them to achieve their goals. In such environment, coordination between the agents is essential and needed to orderly and effectively communicate between them to achieve the goals.

For this purpose, there are several coordination models that we will be discussing.

2.2.2 Coordination models for MAS

A coordination model is a set of rules that define the behaviour of agents on a MAS, which helps in organizing the agents to maximize the effectiveness of their goals and avoid conflicts between them that would otherwise impede or detriment the efforts of the rest of agents. In this work we will discuss two major models for agent coordination: *Yellow Pages* and *Shared Blackboard*. The first one involves a lookup system, like an active directory, colloquially named *Yellow Pages*. In this model, agents look for peers to communicate in this directory and then address them either individually, or several of them at the same time through separate channels, but communications are otherwise mediated by the agent initiates the lookup.

The other model consists in having a *Shared Blackboard* in which agents can perform basic *R/W* operations through coordination primitives. All the read and written messages are seen by a group of agents, which is used to coordinate them. One example of such model, and one we will be using for the implementation as it will be seen in 5.2.2, is TuCSoN[5].

Nonetheless, our system uses both models in conjunction: A lookup system (The publish/subscribe broker) to search for agents in the network, and a shared blackboard to negotiate with them simultaneously.

2.3 Publish/Subscribe paradigm

The Publish/Subscribe paradigm[6] is a messaging pattern that involves two main roles: The publishers, who send messages, and the subscribers, who under certain conditions, receive said messages.

Publishers send a categorized message of some kind to a piece of *middleware*, usually called *broker*. This broker registers the message and is category and proceeds to send it to the subscribers if they have previously notified the broker that they should be sent messages from this category. This pattern allows senders to not specify the recipient of their messages, but rather let the broker relay it to those who stated interest in it. Conversely, it allows recipients to not specify from whom they want to receive the information, but what information they want to receive instead.

Our system uses a Publish/Subscribe broker to perform peer discovery in the network when looking for a certain type of medical cases, as it is discussed in 4.2.

2.4 Ensuring data privacy

The data this system manages is, like all medical data, extremely sensitive. For this purpose, the information is anonymized. This means that in case the medical data is acquired by a malicious agent trusted in the network or in possession of the cryptographic keys, it would still not be possible to infer the identity of the patient, vulnerate the privacy, or use it in any other purpose besides medical research. This is achieved by removing personally identifiable information from the datasets.

There are various types of anonymization models suitable for health data protection as shown in [7], but as discussed in [2, Section 2.2], $(k, k^m)-$anonymity is suitable for medical datasets containing both relational (i.e., single-valued) and transactional (i.e., set-valued) attributes. As further explained in [2, Section 2.2], $(k, k^m)-$anonymity proposed in [8] ensures that for any record r in the dataset and any set of m or less items in transaction attribute of r, there should be at least $(k-1)$ records that are indistinguishable from record r.

Considering that the system uses single-valued attributes, the anonymization algorithm used is $k-$anonymity achieved through generalization, which by changing the structure of the tree and type of the node it can be extended to $(k, k^m)-$anonymity. The $k-$anonymity ensures that the information for each person contained in the dataset cannot be distinguished from at least $(k-1)$ individuals whose information also appear in said dataset. As a side note, further security options can be implemented, such as full system encryption with patient-mediated access control as described in[9].

2.4.1 Generalization

Generalization is a method to achieve k-anonymity by selectively "blurring" data from attributes (i.e., instead of stating someone is 16 years old, we might just say he's teenager). This "noisy" information helps create records that are indistinguishable from each other, without proving to be a critical information loss for most applications that don't require it. This is done to remove personally identifiable data within an EHR, in an intelligent manner, through the use of agents, to prevent direct or indirect identification of the patient while still keeping the usability of the data for research.

What generalization uses to obscure the information are genralization trees, which are trees that encode the possible ranges or values of a parameter in binary tree form. Taking age ranges as an example, the allowed values are 0-100, which is the whole spectrum we allow our attribute (age) to take. This range covers the whole tree and is represented as $*$. From 0-50 we use 0 and for 50-100 we use 1, from 0-25 00, from 25-50 01, from 50-75 10, from 75-100 11 and so on. If we take the example of gender as an attribute (which is categorical, the tree just encodes all their different values separately. The following graph illustrates how generalization trees are formed:

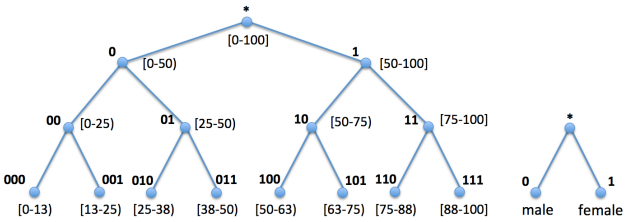


Figure 2.1: Generalization tree example for age and gender

Chapter 3

State of the art

In this chapter we will analyze the state of the different technologies that are directly or marginally involved in the development of solutions that try to achieve the same goal of our system such as the current approaches to EHR exchange, as well as some academic work in the field.

3.1 EHR exchange

Since the emergence of EHR storage within medical institutions, there has been efforts to implement solutions to provide EHR exchange both within the own institution and to others. One such effort is the Health Level 7 [HL7] Clinical Document Architecture [CDA][10] standard, intended to provide systems with medical data exchange capabilities through the use of messages and specific markup for clinical documents. Another effort is the Integrating the Healthcare Enterprise [IHE] profiles[11] for medical data exchange between entities, which are system design patterns that provide diverse functionalities. Among this, some of them are of interest regarding medical data exchange, such as Cross-Document Sharing [XDS], Cross-Community Access [XCA] and Cross-Community Patient Discovery [XCPD] profiles. These profiles are designed to provide medical systems with interoperability, mainly to enable medical case transfer between communities patient-wise.

These profiles are analyzed in [12, Section 4.1], and as it is pointed out there, they are not suitable for a system that dynamically exchanges data between heterogeneous systems because of limitations within the standards, such as inability to exchange documents between multiple communities or the need for preestablished agreements and/or knowledge of the nodes to negotiate with.

Because using either of these standards would severely cripple the ability to dynamically exchange data between heterogeneous systems, our implementation is not IHE nor HL7 compliant.

3.2 Related work

Related work in this area is diverse, though related to implementation of systems with a goals very much like our own there are two main projects.

In [12], Urovi et al. describe a system to exchange EHRs in a secure manner over a P2P network using an agent based coordination framework. However, this system is intended for secure record exchange between different centers as to provide continuous healtcare through different clinical centers, and

does not contemplate dynamic aggregation of data for research purposes, and because of this the information is secure as in it's encrypted for secure transfer between communities, but it's not anonymized because it's unsuitable for continuous healthcare (as the whole unmodified record is needed) and thus it doesn't preserve privacy. Nonetheless, this work shares a lot of similarities to our own, notably the use of TuCSoN to build the MAS.

In [13], Brugués et al. present MOSAIC, a system designed to exchange medical data through multilateral agreement for research. This system had two elements in common with our own: use of agents and a lookup system for peers to exchange data with. The agents were classified in two types: Contributor agents and Petitioner agents. The contributor type would be created by the medical centre hosting the data to tend to incoming requests and the petitioner agents would be responsible for making those requests to the contributor agents.
This system was designed to build research databases for private use, and thus, security requirements aren't as constrictive as in ISyPeM2, that means that the cases are exchanged as is, no anonymization made or any data altered. Besides that, it also contemplated that the different institutions would optionally require more medical data in exchange as queries were made to them, as a result, contributor agents could optionally set a number of medical cases of a certain kind as a requisite, and petitioner agents would have to resolve the requisites imposed by the contributor agents. This problem was solved through the use of multilateral agreement through agents. This is not our case since the data is openly offered, and any data that a node wishes to obtain should be collected by his own doing. Finally, this system is not capable of building a shared, anonymized research database, which is one of the goals we proposed.

Chapter 4

Proposed Framework

In this chapter we will introduce the proposed framework for the ISyPeM2 system, being the goal of this thesis the implementation of it. All the elements will be discussed from a theoretical point of view, and in the following chapter, the actual implementation will be explained in detail.

4.1 ISyPeM2 Node

ISyPeM2 is designed to be a solution meant to be run on each medical community, forming a network of centres that exchange data. Consequently, each medical centre is a node in the network. The following diagram represents all the elements in an ISyPeM2 Node:

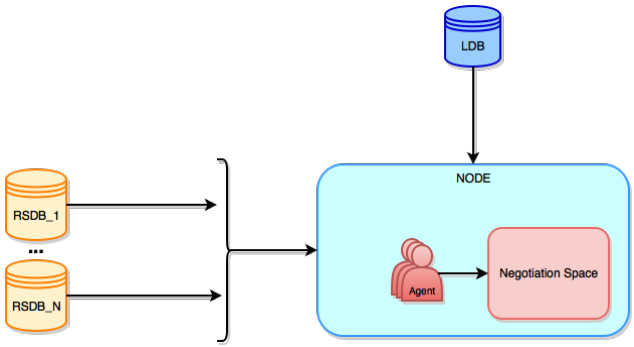


Figure 4.1: System diagram for ISyPeM2

Now we will describe the elements as seen in the diagram:

- Local Database [LDB]: The LDB is the actual collection of medical cases from each medical centre. This information is exclusive solely to the very own centre and will not be shared directly with the network. Whenever a node negotiates with another one, they agree on giving a number of cases from the LDB, but anonymized to keep patient privacy.
- Research Database [RSDB]: The RSDB is a database being built or already built with anonymized cases from negotiations with other nodes. As such, they don't contain any patient-identifiable data and is solely useful for research purposes. ISyPeM2 is designed in such way that RSDB built by a node are shared with the rest of the network, so all the data compiled is usable for the rest of the nodes

- Negotiating Agent(s): Each node has one or more agents that perform negotiation whenever a user from the own node starts a negotiation process or when another node in the network requests negotiation with our node. They carry out the negotiation and control the logic on anonymizing the case attributes and acceptance of the negotiation parameters.
- Negotiation space: This is an environment inside each of the nodes which purpose is hosting a number of agents inside so they carry out the negotiation. Following the coordination models (as explained in 2.2.2) it serves as the *Shared Blackboard* that all the agents have access to coordinate with themselves.

The nodes are designed to be a way to represent the medical centres within the network, that means that all the requests are done in its name. Also, it provides the network with an identifiable element to which address other node's requests. The purposes of a node are several:

- Identify the medical centre by an uniquely identifying element and an addressable element.
- Maintaining and organizing a private collection of cases (a LDB).
- Giving the user an interface to with the rest of the nodes, by starting negotiation processes or querying the network for other peers.
- Host one or more environments in which agents can carry out negotiation processes.
- Creating and dispatching agents to negotiate with other nodes or participate in negotiation processes when required.

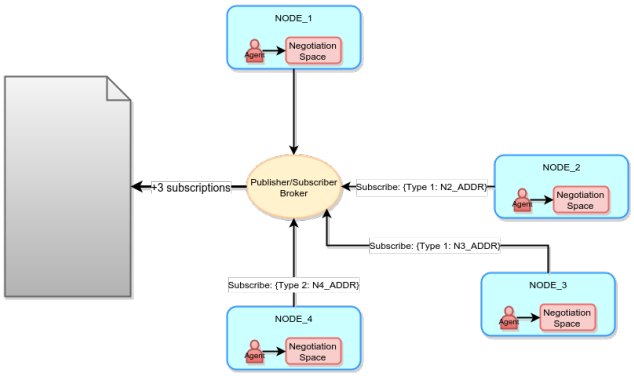
All in all, a node is just an abstraction of a medical centre in our network, and not just of its elements, but also of its very own goals and actions. The nodes will, as it is what this system provides, negotiate a collection of medical cases with other nodes in the network. The same node that provides the cases will also be responsible of ensuring that the information being given is anonymized properly.

4.2 Publish/Subscribe broker

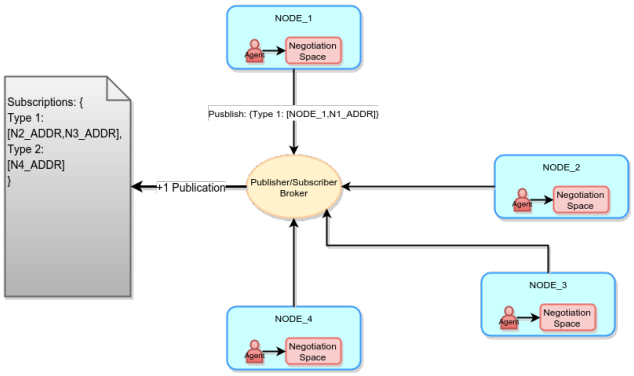
For the system to work, we need a way to explore the network for other nodes. In some networks, topographical discovery is just unviable, so we need to provide the nodes with a mechanism to find other nodes to negotiate, as well as giving them the ability to find only the nodes that do have the data wanted by the requesting node, instead of broadcasting a request, as this would unnecessarily increase network traffic and system load. For this purpose, the system is designed to use a component called *Publish/Subscribe Broker*, and basically it's a lookup system on the network that uses the Publish/Subscribe paradigm explained in 2.3.

The broker role consists in giving publishing nodes a way to register availability of a certain kind of medical information within the network to the other nodes, otherwise said, it's an announcement mechanism that tells the rest of the network that they have a certain type of medical cases available for sharing and how can this publishing nodes be contacted. In this way, the rest of the nodes can subscribe to this publications to be notified when a peer node has published the type of information they were interested in, instead of performing active discovery of peers or forcing the publishing nodes to broadcast the network for availability.

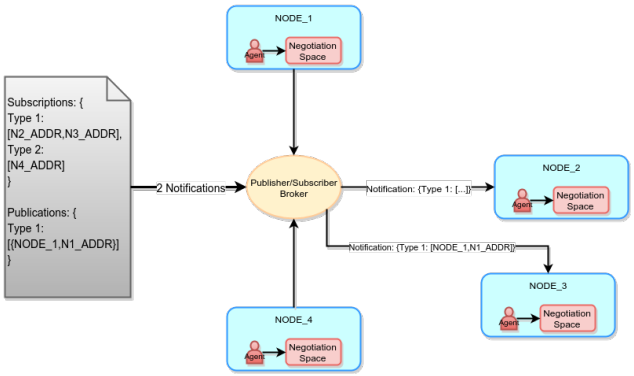
The following diagrams illustrate how the mechanism works and how it interacts with the nodes:



(1) Some nodes subscribe to a certain type of information



(2) A node publishes availability of a certain kind of data



(3) The subscribed nodes get notified

Figure 4.2: Publish/Subscribe process between peers

By using this Publish/Subscribe broker, we are able to implement a lookup system in which subscribing nodes express their interest in finding peers that offer a certain kind of information and be asynchronously notified when those nodes publish themselves in the broker, leaving the subscribing node able to perform other task in the meantime.

4.3 Agents and negotiation

The agents in ISyPeM2 are one of the most important components. One of the critical points in the system is the anonymization algorithm, which is supposed to, as explained in 2.4, obscure the

attributes from the medical cases to provide $k-$anonymity. However, there is a problem: depending on the value, not all the fields are obscured in the same way, it depends on it being a ranged value, a fixed numerical value, a categorical value, etc. This means that depending on the value, more or less obscurity in the attributes is needed.

The process of negotiation between agents consists in both agents agreeing to:

- The number of cases they will be exchanging
- The desired values for the attributes to be exchanged (i.e. Maybe the agent is only interested in caucasian males of age between 25-35)
- The $k-$anonymity parameters, which are basically the k parameter and, as it is achieved through generalization, the generalization trees.

If they manage to agree to these parameters, the generalization is applied and the medical cases are transferred. Since different attributes are differently obscured, we need to tell the software to reason about the amount of generalization needed depending on the three negotiation parameters.

Some other uses for agents would be identifying the attribute just by the name, but because attribute names are not always standardized between institutions, this means the agents would have to deal with semantic networks and that is, at the moment, out of the scope of the project.

Chapter 5

Implementation

In this chapter, the details on the system implementation are presented. We will dive into the constructs of the system, as well as the design choices and their inherent limitations. After that, the whole development environment will be described.

The state of the implementation as of right now is functional, but has few of the final features as development is still underway. As of now, the system is able to:

- Publish and subscribe to information in the Publish/Subscribe Broker.
- Perform all possible operations on the LDB (create, destroy, add, delete, modify).
- Perform simultaneous negotiations between agents and exchange medical cases between them.
- Perform generalization on attributes, however, this functionality hasn't still been merged into the negotiation process, so the information exchanged is not anonymized. The development process consisted in designing a component of the system, implementing it, write a unit test for that component (where possible) and documenting said document. That process was repeated for each of the system components being implemented.

  Now, we will detail the implementation of each of the components in the order they were implemented.

5.1 Publisher/Subscriber Broker

The Publish/Subscriber Broker was the first component we implemented. The broker we used was an existing solution known as Orion Context Broker[14].
The Orion Context Broker is a Publish/Subscribe broker that provides NGSI9/10[15] interfaces, which are based in the NGSI Context Management specification proposed by the Open Mobile Alliance [OMA].In NGSI terminology, we can distinguish three elements:

- Entity: An Entity is an abstraction of a real-word element, for instance: a tree.
- Attribute: An Attribute is a property of an Entity. In the tree example, the name, the height and the number of leaves are Attributes.
- Context Element: A Context Element is the information exchanged about one or more Entities. We will mostly refer to Context Elements rather that Entities, as Entities are an abstract

element and the Context Element is the actual information being exchanged.

The Orion Context Broker NGSI9/10 interface is implemented through a RESTful platform. Therefore, all the operations (query, publish, subscribe) are performed by sending JSON-formatted messages to the REST platform. An example of such operations can be seen in Appendix B.4. All the context information is stored internally in a private MongoDB[16] instance.

As for our platform, the main element of a query is what type of cases are we looking for, as in what are the cases we want to look for about. Possible types of cases could be about a medical condition (cases about Diabetes Type 2, cases about Malaria, and so on) or also about drugs administered (cases on people taking a certain drug, for TDM or drug trials). For this reason we are using the type of medical case as a Context Element with the following attributes:

- `numNodes`: Indicates the number of nodes that have published cases of the Context Element that contains it.

- `description`: Human-readable description about that type of medical cases. This is ignored by the system and is only designed to provide a meaningful description of the Context Element to the end-user.

- A indefinite number of `node` attributes: Each time a node publishes in a Context Element, an attribute that represents that node is added to the Context Element. Since attributes could not contain nested entities, we had to summarize the information of a node in the information that an attribute holds, which are name, type and value. The name would be the name of the node, type would be *"node"* and the value would be a string containing the address of the node, the number of cases of that type it can provide and a timestamp showing the last modification to that collection of cases in particular. That string would be have to be parsed by the client requesting a Context Element.

Following the above, the procedure that nodes follow for publishing into the context broker consists in incrementing the numNodes attribute and adding a node attribute with the publishing node's information.

The subscription operation in the Orion Context Broker is performed against one or more context elements and one or more of their attributes. The inherent limitation of this mechanism is that in a given moment, a node can subscribe to all of the current attributes on a Context Element, but if we add more attributes, since they are not part of the subscription, the node will not be notified. To circumvent this we added the *numNodes* parameter, so everytime a node is added or removed from a Context Element, they will alter the *numNodes* parameter that all the subscribing nodes are unconditionally subscribed too, and consequently, they will redo their subscription to include or remove the modified nodes.

Since the system is distributed, any node could modify an attribute or a context element to which another node is subscribed to any time. Instead of relying on the subscribed nodes polling the Broker a certain interval of time, the Orion Context Broker uses a referral URL, which is an URL that the subscribing node provides to the Broker, and to which the Broker will send a JSON-formatted notification whenever the subscribed information is modified. For that purpose, every node is suited with an embedded HTTP server to receive and process these incoming notifications.

5.1.1 Notification Server

Since the notifications are just HTTP POSTs with a JSON-formatted body, we need an HTTP server to receive and process these notifications. Initially, it was thought to use servlets on a Tomcat server, but we still needed a way to communicate the servlet with an already running application, implementing some sort of middleware that performed message passing between them both. We realized that it was less troublesome to use a lightweight embedded HTTP server, and for that purpose we used Jetty[17].

Originally, the embedded Jetty server would be listening in port 8000 TCP on each node, however, support for listening to several Brokers was added in the last commits while trying to modify minimally the code. As a result of this, for each notification server activated for subscribing to a new Broker, a different port is used. The first notification server will run in port 8000, the second in 8001 and onwards, without any sort of port status checking. This is obviously suboptimal and error prone, a better approach would be running a single instance on port 8000 and making each Broker notify to a different path of the web server. Unfortunately, we weren't able to test the commit for this new behaviour and the old code was left. As for the server itself, it will forward each notification to the Node and it will be the Node itself that processes the notification and detects the changes between the last reported state (stored in the node) and the context information from the notification, taking any pertinent action.

5.2 ISyPeM2 Node

The development of the ISyPeM2 node first started with the implementation of the interface between the LDB and the Broker, which meant that the LDB storage system was to be developed at the same time. That system went under numerous revisions as certain aspects of the medical case organization within a LDB were overlooked. Finally, a storage backend called Case Provisioner was implemented to perform all the operations transparently from the storage medium. Currently, the system is capable of storing medical cases in almost every SQL-compliant system with minimal modification (currently MySQL and SQLite are implemented and tested), and also there's another storage backend, labeled *Local Case Provisioner*, that essentially stores all the information in memory, which is suitable for testing.
Following this, the Multi-Agent environment was defined and implemented, along with the negotiation process and the medical case transfer system.

As for the Node implementation itself, it is an element on the network and has the following attributes:

- Name: Unique identifier within the network. It's a string of characters indeterminate length and format.
- Address: Addressable element of the node. Considering the nodes are running on a IPv4 network, this address is either a IPv4 address or an IPv4-resolvable hostname that can be accessed by any other node in the network.
- Published Aggregations: A list of case aggregations (defined later) published by the node. This allows the node to keep track of the cases that it has published in the network.
- Active agents: A list of agents that are currently actively performing negotiation processes.

- Active subscriptions: A list of subscriptions done in a Broker
- Notify servers: A list of notification servers active for each Broker the node may be subscribed to. This is a way to enable the node to be subscribed to multiple context brokers within the network.

5.2.1 Medical Case Management

Medical cases are considered in our system as an object with the following attributes:

- A Unique Identifier (currently a 128 bit hash).
- A Patient Identifier that is unique for every patient, but that can be repeated (as in taking multiple measurements from the same patient).
- An arbitrary number of quantifiable or qualitative attributes.
- A type, which defines what is the medical case about and as defined before, can be a certain drug or a medical condition.
- A text string labeled *"Health data"* which currently holds all the information about the case. For instance, if the case is about a certain drug, it could contain the observed concentration and the dosage amounts respectively, if the case is about a medical condition, it could contain medically relevant information about the treatment and the evolution of the condition.

When querying a node for a number of cases, the node sending the query will specify a set of attributes and values to be matched against the cases in the LDB, meaning that the cases being asked for by the querying node are those who have **at least** those attributes in common and the values are either equivalent or inside the specified range.

5.2.1.1 Medical Case organization

Since we're dealing with multiple cases, we need a way to classify and group them. In our system we have two main groups of medical cases: Case Collections and Case Aggregations.

Case Collections are defined as a group of one or more cases of the same type and with the same attributes. Case Collections have the following attributes:

- Name or ID: The name of the Case Collection. This is purely for identification purposes, however, it is encouraged that unless another collection exists with the same name, a human-readable name is used.
- Type: The type of the cases within this collection
- Description: Human-readable description of the Case Collection. The system does not process this in any way, it exists merely to give information to the end user.
- Attributes: The list of attributes that all the cases in this Collection have in common. These attributes have some values associated that are used to keep statistics on the values of the attributes of the cases in the collection. That means that each time a case is added or removed, the Collection attributes are recalculated to specify the minimum and maximum values of that attribute between all the cases. If no such quantification is possible, the attribute is marked with a wildcard to denote that there are multiple values.

- Keywords: Keywords describing the content of the cases in this collection. These are used to do more fine-grained searching, if some query contains a number of keywords in it, only the Collections that contain **at least** those keywords will be returned.
- Node of origin: Specifies from which node did the case collection come from.
- Cases: The actual collection of medical cases.

For instance, if we have 100 cases about Malaria, and 80 of them have Age and Gender as attributes, and the 20 remaining have Age and Weight as attributes, we have two different Case Collections.

Case Aggregations are a group of one or more Case Collections of the same type, but not necessarily having the same attributes. Case Collections have the following attributes:

- Name: Name of the Case Aggregation. This is the type of medical cases it has, so it is either the name of the medical condition or drug that the cases are about.
- Timestamp: Timestamp with the last modification to any of the Case Collections in this Case Aggregation.
- Description: Human-readable description of the Case Aggregation. The system does not process this in any way, it exists merely to give information to the end user.
- Collections: The list of Case Collections in this Aggregation.

This means that in each LDB we will have only one Case Aggregation of each type of case, but one or more Collections in each Aggregation.

Following this organization, what a node publishes in the Publish/Subscribe Broker is metadata describing his Case Aggregation, as detailing each of the attribute combinations of each Case Collection can be too much for the Broker, this task is delegated to the negotiation process as will be explained in 5.2.3.

Considering a node has a single LDB, this LDB will have any number of Case Aggregations, but only one for each type, that in turn encompass one or more case collections, each one with its set of attributes that also contain one or more medical cases. Thus, building a LDB consists in having at least one Case Aggregation stored within a node. The RSDBs building procedure is explained in 5.2.3.

5.2.1.2 Storage Backend

In order enable the nodes to store the cases, a Storage Backend interface named Case Provisioner was implemented. This interface defines a series of operations that can be performed against the medical case organizational units (Aggregations and Collections), which are read, write and remove. A separate operation, DB querying, was implemented to perform queries in a more efficient way, using storage-specific routines/operations, rather than fetching all the current Case Aggregations using the interface and analyzing each one of them, since this could cause database locks and perfomance issues.

While the interface is enough to define these operations, each storage engine needs an implementation of those operations. As long as one is able to implement the operations in the interface in a specific storage medium, one would be able to use such storage medium as medical case storage without the node needing to deal with the storage engine or have any knowledge of it.

It should also be noted that many of the operations performed to the storage backend are done so using non-specific operations, for instance: To read the description of a Case Aggregation, the current implementation does fetch the whole Aggregation from the storage medium and read the description from it. This means that even though in a storage-specific implementation the system could fetch just the description (as originally intended), this use of the interface does not only fetch that, but the rest of attributes and all the Case Collections that the Aggregation contains. If the Aggregation is big enough and the medium is not in memory it could lead to very large disk I/O operations just to read a description. To resolve these performance issues, each of these operations would need a storage-specific implementation, thus increasing the complexity of the individual implementations of the interface. Since the system implementation is not final, this is left as is, yet to be improved.

During the development of the Publish/Subscribe Broker, case storage was performed on memory using a Provisioner named Local Case Provisioner, as it was simpler for testing. However, the final implementation of the storage engine is using MySQL. With scalability in mind, there's an SQL-agnostic implementation of the Case Provisioner, which only needs minimal code added for each SQL engine. Currently only MySQL and SQLite are implemented and tested. The use of a SQLite engine implementation is discussed in 5.2.4.

5.2.2 Multi-Agent System Implementation

The Multi-Agent System was implemented using TuCSoN[5]. TuCSoN is a coordination model that defines a common space that agents can use to interact with other agents and the local execution framework that hosts the space itself. This shared space is called *Tuple Centre [TC]*. In a node there can be multiple TCs, however, only first-order logic ground terms are admissible names. In any node running the TuCSoN service, there will always be one called `default`.

In these TCs, agents can use communication primitives to read and write tuples. A communication primitive is essentially an operation performed in the shared space that affects the messages (tuples) that are present in such space. The most important communication primitives are:

- `out/1`: Writes a tuple to the target Tuple Centre.
- `in/1`: Blocks and waits until the tuple or tuple template specified in the argument appear in the TC, then it reads and consumes it.
- `rd/1`: Blocks and waits until the tuple or tuple template specified in the argument appear in the TC, then it reads it.
- `inp/1`: Performs a non-blocking `in` operation.
- `rdp/1`: Performs a non-blocking `rd` operation.
- `set/1`: Writes the list of tuples passed as argument, replacing any existing tuples in the target TC

TuCSoN is essentially an Agent Coordination System built on top of a Prolog engine developed by the own authors called `tuProlog (2p)`. Because of this, the nature of TuCSoN is intimately related to Prolog, one direct consequence of this being that tuples are atoms or compound terms with any sort of parameters a Prolog compound term may have (atoms, numbers, lists or another compound term). Besides this, to do the in and rd operations (and their non-blocking variants) we can use tuple templates, which are essentially Prolog variables that match tuples.

To send a primitive to the TC, one must specify the primitive and its arguments and the TC the primitive is targeted to, for instance: `out(testTuple(arg1,arg2)) @ tuplectr1`. When the TC is not specified, it is sent to `default`.

One of the most powerful aspects of TuCSoN is the ability to program the TCs to react to certain coordination operations through the use of a specification language called ReSpecT that defines reactions to coordination operations. ReSpecT reactions are expressed with the following tuple:

```
reaction(E,G,R)
```

Where `E` is the coordination operation to which the reaction reacts, `G` is the stage of the coordination operation, which may be *invocation* if the operation has been invoked or *completion* if the operation has been carried out. Finally, `R` is the actual reaction, which is basically a list of coordination operations and/or Prolog predicates to run when the reaction takes place.

In the implementation, each node has at least one agent active at any time. This agent is called Spawning Agent and listens to the `default` TC. It's purpose is to spawn a Negotiating Agent whenever a peer node's agent comes to `default` and asks to negotiate, following the typical multi-threaded server *Thread-per-Request* pattern. The spawned Negotiating Agents will in turn participate in the negotiation representing the node that spawned them.

It should be noted that in the final code implementation, all the input operations (`rd` and `in`) are used in their non-blocking counterparts in constructs that emulate blocking behaviour, for instance, using `rdp` in a loop keeps retrying the operation until a tuple is successfully read. The reason behind this is that TuCSoN has deadlock issues when a large number of agents are running simultaneously. This has improved significantly in the latest version because the code is faster and thus it is less likely for deadlocks to happen. Nonetheless, it is still entirely possible, but using the non-blocking operations indefinitely and thus emulating blocking behaviour prevents deadlocks.

5.2.3 Negotiation Process

The negotiation process is a process that is created by a node in order to obtain a certain number of cases to build a RSDB. It has the following states:

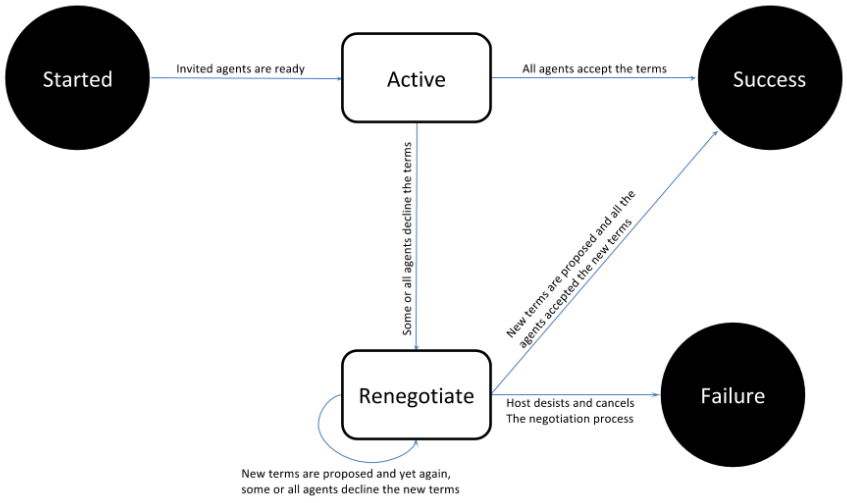


Figure 5.1: Lifecycle of the negotiation process

It starts when the node is given the type of cases desired, a number of cases to obtain, the desired attributes for these cases and some keywords to perform the query. After this, the node creates a Negotiating Agent that initiates the process. This Negotiating Agent is called the *host* of the negotiation.

Once the process is started, the host creates a TC within his own node, which is the TC where the negotiation will take place. The name of the TC is the name of the node with a randomly generated hash appended to it. If the TC already exists and has not been marked as reusable, the agent will try to create another using a different hash.

Once the TC has been successfully created, the agent injects a ReSpecT script that controls the state of the negotiation. The following is an explanation on the evolution of the negotiation states:

1. Started: When the host of the negotiation first writes into the TC the list of peer nodes (the invite list) that will be taking part of the negotiation, the state of the negotiation is set to *Started*. The host of the negotiation will proceed to send invites to all the nodes in the invite list. When each of them arrive to the TC, they will write a tuple `hello(NodeName)`.

2. Active: When all the agents write the `hello` tuple, a reaction that sets the state to *Active* is triggered. If a peer takes too long to respond, it is removed from the invite list so that the negotiation doesn't indefinitely stay on *Started* state. In this moment, the node proposes the terms of the negotiation and the peers evaluate them.

   2.1 Renegotiate/Failure: If the terms proposed by the host are not accepted by one or more peers, the host can either desist and terminate the negotiation by setting it into the *Failure* state and marking the TC as reusable or it can set the state to *Renegotiate* and propose new terms. When the *Failure* state is reached, all agents terminate as if the negotiation had finished successfully.

3. Success: If the terms are accepted by all the peer agents, the case transfer occurs. When each node finishes transfering the cases to the host, the host itself marks the agent as finished writing a `finished(NodeName)` tuple. When all the agents from the invite list have been marked as finished, a reaction that sets the negotiation state to *Success* is triggered, effectively ending the negotiation process as all agents terminate when this state is reached.

In the end of this process, the host agent will either have obtained a number of cases or the negotiation failed. In the former case, if the amount of cases obtained isn't the amount desired, the host node will remain on the negotiation TC waiting for other peers (or existing peers) to publish in the Publish/Subscribe Broker cases of the same type, in which case an invite will be issued to them and the negotiation process will return to *Active* state, repeating this cycle until the host obtains the number of cases desired.

5.2.4 Transfer Server

Since we already had the means to deploy embedded HTTP servers using Jetty, the case transfer is done using a separate instance of Jetty running in port 7999 TCP. When an agent accepts the conditions of the host in a negotiation process, it will create a separate Case Collection with the anonymized cases, and using the SQLite Case Provisioner, it will create a SQLite database file. Finally, the database file is compressed using Gzip[18].

The resulting file is identified with its MD5 hash by the transfer server. Afterwards, the transferring agent answers the receiving agent with an URL with the previous hash as path. When a request

is made to such path, the HTTP servers transfers the file, which is received in the remote end, and since the MD5 is also the filename, it is hashed with MD5 to check wether the transfer was successful or not, and finally decompressed and read back again into a collection using the SQLite Case Provisioner.

Gzip was chosen as a means to reduce transfer times in big collections because SQLite stores all the data in plain text (though structured) in a file, and since most of the data is text in a specific language and patterned numerical data, both of these being quite compressible, good compression is achieved even with a simple compression algorithm such as Gzip, meaning we can also benefit of its fast compression and decompression times.

5.3 Development environment

For the development of the system, the Java language was used. The decision to use this language is a consequence of using TuCSoN, as its programming API is written in Java. However, The reason to use TuCSoN for the MAS system is that the researchers involved with the project are already familiar with it and Java, thus making it a requirement. Nonetheless, I believe that Java is a good choice for a programming environment in this situation since it requires both a high-level language to program the complex tasks the agents perform and code that runs reasonably fast.

Originally, Java 7 was used having stability in mind. Nonetheless, we had to upgrade TuCSoN to the most recent version to surmount some issues with multithreading, which required Java 8. The programming environment used is thus Java 8 at its latest version.

The machine used for development runs GNU/Linux, specifically, Xubuntu 14.04. The system runs on top of a VT-x capable Intel Xeon CPU with 8 logical cores and 16 GB of RAM. While a lesser CPU would've sufficed to develop and run the tests in Chapter 6 (for 10 agents or less), the 16 GB of RAM are a requisite as explained in 5.3.1.

To store the code in a Version Control System [VCS] git was chosen, as it was the VCS that the developers were familiar with.

5.3.1 Virtualization environment

To test the system, a virtualized environment was set up. As we had no specific virtualization needs, VirtualBox[19] 5.0.1 was used as virtualization engine. The virtualization environment is comprised of several virtual machines (which will be detailed in the following sections) and a host-only network: `192.168.1.0/24`, which isolated the virtual machines and the external network environment to avoid unsolicited traffic interfering with the virtualization environment. Outbound access from this network was routed through a virtual machine called `Router`, which is described in the following section. The following figure illustrates how the virtualization environment was set up:

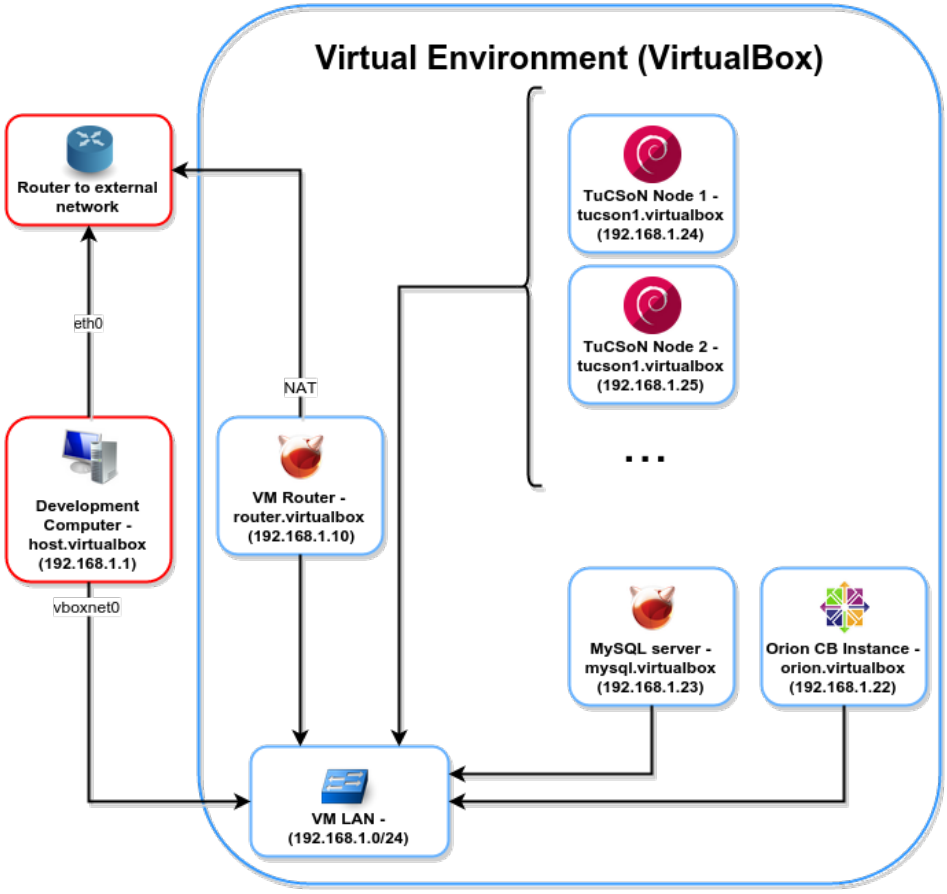


Figure 5.2: Virtual Environment Setup

All the virtual machines were running with a KVM-compliant Paravirtualization layer (Available under VirtualBox 5.0 and above) and Hardware-assisted virtualization through Intel VT-x. Virtual Machines running FreeBSD as guest OS had a Hyper-V layer instead.

5.3.1.1 Router VM (router.virtualbox)

The Router VM is a Virtual Machine that was used for networking-related purposes. The specifications are as follows:

- Guest OS: FreeBSD 10.1 x86
- CPU: 1 core
- RAM: 512 MB
- Disk: 8 GB

The functions of this machine were the following:

- Routing traffic from the Virtualization Environment to the Internet.
- Host a DHCP server to assign an IP address to each Virtual Machine in the environment, which are statically assigned per MAC address.

- Host a DNS server to provide with hostnames to each Virtual Machine in the environment.

5.3.1.2 Orion Context Broker Instance (orion.virtualbox)

This Virtual Machine hosts an instance of the Orion Context Broker. The specifications are as follows:

- Guest OS: CentOS 6 amd64/RHEL 6 amd64 (Required by binary package)
- CPU: 2 cores
- RAM: 4 GB (Requirement in the official documentation, even though it is possible that 2 GB is more than enough)
- Disk: 20 GB The sole purpose of this machine is, as stated before, to host an instance of the Orion Context Broker.

5.3.1.3 TuCSoN Nodes (tucsonX.virtualbox)

Each of these Virtual Machines represent a Node in the network. It's the machine that runs the actual Java code. The specifications are as follows:

- Guest OS: Debian 8 amd64
- CPU: 2 cores
- RAM: 1 GB (It is possible that 512 is still enough)
- Disk: 8 GB

This machine also requires JRE 8 to run the ISyPeM2 and the TuCSoN Node Service, which is preferrably installed as a system service.

5.3.1.4 Database (mysql.virtualbox)

This Virtual Machine acts as a MySQL server to provide the ISyPeM nodes with a storage backend for medical cases. It has the following specifications:

- Guest OS: FreeBSD 10.1 amd64
- CPU: 2 cores
- RAM: 2 GB
- Disk: 20 GB

It should be noted that on a real-life case scenario, each node would have its own MySQL instance (or other compatible storage backend) in each node, however, with resource saving in mind, we deployed a single MySQL instance into which each node operates using its own database.

Chapter 6

Results

To ensure nominal operation of the system and check its performance, a series of tests were performed. In this chapter, the tests performed and their results are presented.

6.1 Dataset

In order to perform tests on the system, a medical dataset was provided. The dataset was comprised of two separate databases, one with 8898 records (called `Gentamicin_large`) and a second one with 224 records (called `Gentamicin_small`), totalling 9122 medical records.The medical records are a data collection on Gentamicin dosages on newborn babies.

The data is statically anonymized as no personal data is included in the dataset, however, since a data protection and/or usage policy was not specified, the dataset is not disclosed as a precautionary measure unless otherwise stated by the providing party (which is also not disclosed).

As it will be further explained in 6.2, the data had to be split between the 10 nodes that our simulation environment had, so we populated each node's database with 800 records, 90% (720 out of 800) from `Gentamicin_large` and 10% (80 out of 800) from `Gentamicin_small`. The result was parsed from the dataset spreadsheet and exported using a Python script into an SQL dump which was used for testing.

6.2 Tests

The testing environment follows the structure shown in 5.3.1, with a varying number of agents (according to each test).

One of the first tests we wanted to perform was a benchmark, however, there were no systems to benchmark with since the existing systems weren't exactly similar to ours in functionality, or worked in different environments (no anonymization needed, trusted environments, etc.).
However, a benchmarking test comparing the performance of our system looking up in the network for medical cases and performing an exchange with peer nodes against directly querying them to the database was performed, but the results weren't significant, as access times were significantly lower when querying the database directly instead of looking up peers in the network and negotiating with them.

The main feature we wanted to test on the system was the dynamicity, which we defined as, in regular operation, how many cases would a node give and how many would it obtain in average. The reason to analyze this sort of behaviour is because many health professionals and institutions are reluctant to implement and use systems like our own because they do not want to use a system that gives away much of their private medical information (even when anonymized) and not obtaining any information in exchange. For this, we performed this dynamicity tests, analogous to fair share ratio in P2P systems.

Since we wanted to test this behaviour in different environments, we tested with 3, 5 and 10 agents. The tests consisted in starting all the agents simultaneously with previously populated databases, each of them different in content but of equal size. Each agent would select a random query from a predefined pool of queries and perform it, querying all the other peers. Then, all the negotiations would be performed and each node would report all the cases given and received. To test the influence of the predefined query pool in the test, we tested with a pool of 10 and then 20 different queries. The parameter measured is the case gain in each node, which is the number of records obtained minus the number of records given. Each test was performed 10 times.

6.2.1 Tests varying the number of agents, 20 static queries, 10 runs

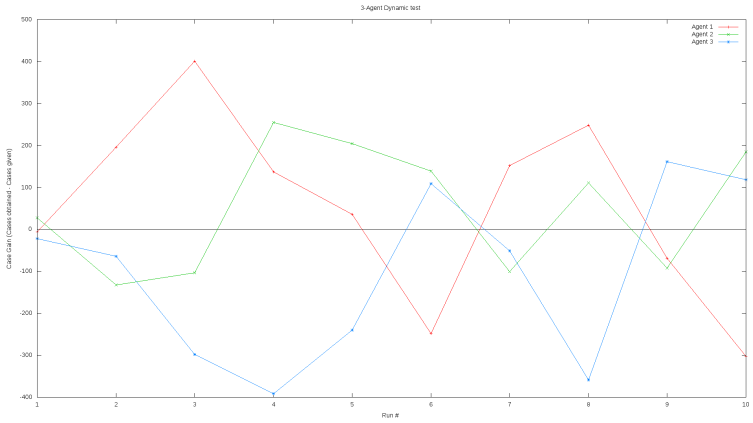


Figure 6.1: Dynamic test with 3 agents

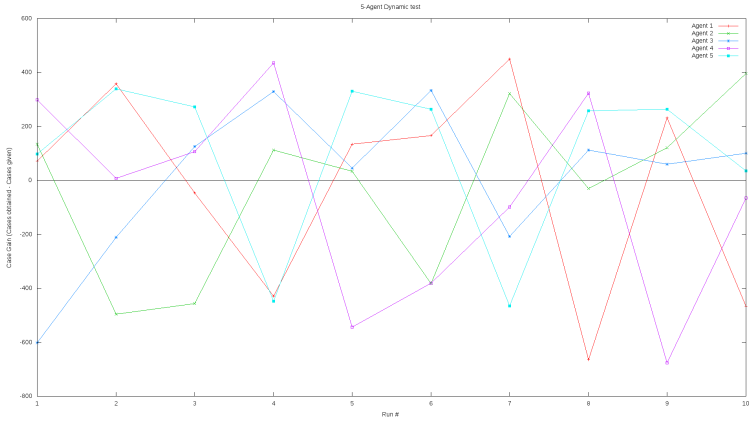


Figure 6.2: Dynamic test with 5 agents

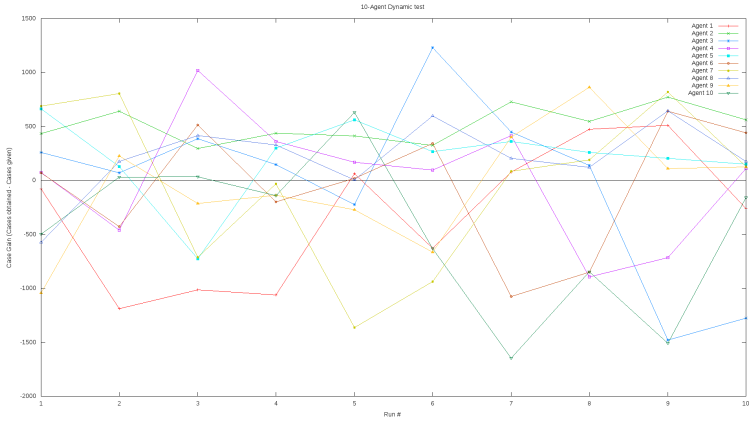


Figure 6.3: Dynamic test with 10 agents

6.2.2 Tests with 10 static agents, varying number of queries, 10 runs

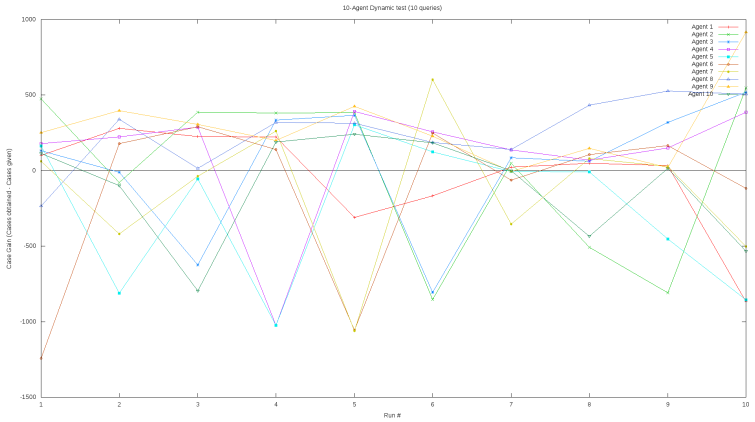


Figure 6.4: Dynamic test with 10 queries

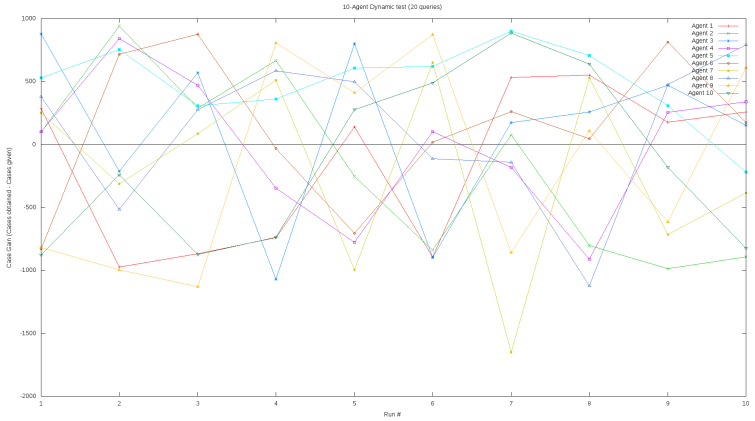


Figure 6.5: Dynamic test with 20 queries

6.2.3 Tests with 10 static agents, 20 queries, 100 runs

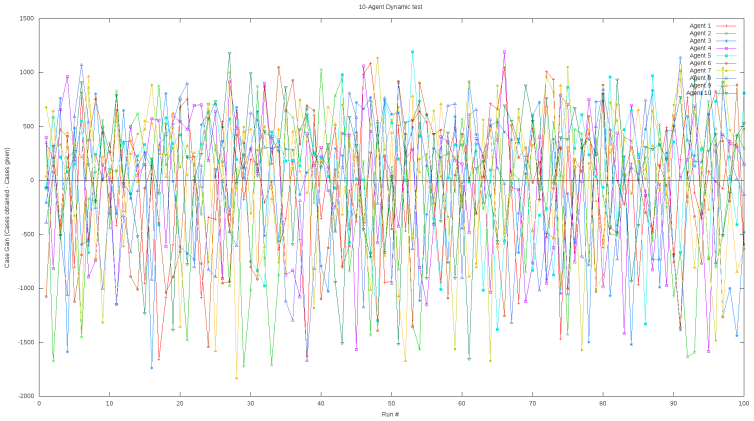


Figure 6.6: Dynamic test with 100 runs

The detailed data table for the figures with their averages is available in Appendix C

6.3 Result analysis

Since what we wanted to test is the share ratio for each node, we averaged the case gain (number cases received minus the number of cases given) for each agent through all the runs.

In general, the share ratio is acceptable, some nodes do obtain or lose more cases that others, but the average is low. Furthermore, in the test performed for 100 executions we can see that the averages are much closer to 0, so we can say that the system converges to a fair share ratio.

With these conditions, the fair share is guaranteed when the following conditions are met:

- The system is used with sufficient frequency by the node.
- The queries performed are sufficiently random (a node that performs the same query all the time will either be very successful, affecting negatively all other nodes share ratio, or will perform very bad.

Chapter 7

Conclusions

To conclude this work, first we must look back to our goals. The main goals were to build a platform that securely, through anonymization, provided a a way to exchange medical records between heterogeneous environments without a previous agreement between institutions and ensure that the system keeps the dynamically aggregated data for future usage of other nodes in the network.

Since this work is still underway, not all the objectives could be met, but the system is able to provide a way to negotiate medical records between heterogeneous environments with no previous agreement, so I believe that the current implementation is in a good direction.

I strongly believe that further work in the current implementation will culminate in a system that is able to perform at a desired level, both in functionality and performance.

The remaining milestones for this project are:

- Improve anonymization capabilities: The current anonymization module (not written by the author) currently works for a small set of attributes.
- Build a reliable anonymous aggregated repository (RSDB), which may be able to reallocate and be hosted on multiple locations to provide High Availability in case of host failure or abandonment of the network.
- Implement integration of LDBs and RSDBs between hosts so the nodes are able to provide records from either in a transparent way to the requesting node.

In conclusion, the state of the current implementation is still partial, but meets some of the objectives and sets the codebase for the rest of milestones to be achieved, which through further development, will reach the desired functionality.

References

[1] systemd. http://www.freedesktop.org/wiki/Software/systemd/.

[2] Alevtina Dubovitskaya, Visara Urovi, Imanol Barba, Karl Aberer, and Michael Ignaz Schumacher. Dynamic Health Data Aggregation for the Research Purposes.

[3] EHR Basics. http://www.healthit.gov/providers-professionals/learn-ehr-basics.

[4] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11:205–244, 1996.

[5] Andrea Omicini and Franco Zambonelli. Tucson: a coordination model for mobile information agents. In *IN PROCEEDINGS OF THE 1ST WORKSHOP ON INNOVATIVE INTERNET INFORMATION SYSTEMS*, pages 177–187, 1998.

[6] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[7] A. Gkoulalas-Divanis, G. Loukides, and J. Sun. Publishing Data from Electronic Health Records while Preserving Privacy: A Survey of Algorithms. *Journal of Biomedical Informatics*, 2014.

[8] G. Poulis, G. Loukides, A. Gkoulalas-Divanis, and S. Skiadopoulos. Anonymizing data with relational and transaction attributes. In H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, editors, *Machine Learning and Knowledge Discovery in Database*, volume 8190, pages 353–369. Springer Berlin Heidelberg, 2013.

[9] Alevtina Dubovitskaya, Visara Urovi, Matteo Vasirani, Karl Aberer, and Michael Ignaz Schumacher. A Cloud-Based eHealth Architecture for Privacy Preserving Data Integration. *IFIP Advances in Information and Communication Technology*, 2015.

[10] HL7 Standards Product Brief Basics - CDA® Release 2. http://www.hl7.org/implement/standards/product_brief.cfm?product_id=7.

[11] IHE Profiles. http://www.ihe.net/Profiles/.

[12] Visara Urovi, Alex C. Olivieri, Albert Brugués de la Torre, Stefano Bromuri, Nicoletta Fornara, and Michael Ignaz Schumacher. Secure P2P Cross-Community Health Record Exchange in IHE compatible systems. *International Journal on Artificial Intelligence Tools*, 2014.

[13] Magí Lluch-Ariet and Josep Pegueroles-Vallés. The MOSAIC System - A Clinical Data Exchange System with Multilateral Agreement Support. *Electronic Healthcare for the 21st century. International ICST Conference*, 2010.

[14] Orion Context Broker. http://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker.

[15] NGSI 9/10 specification. https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/NGSI-9/NGSI-10_information_model.

[16] MongoDB. https://www.mongodb.org/.

[17] Jetty. http://www.eclipse.org/jetty/.

[18] Gzip. http://www.gnu.org/software/gzip/.

[19] Oracle VM VirtualBox. https://www.virtualbox.org/.

Appendix A

Table of Acronyms

CDA Clinical Document Architecture

EHR Electronic Health Record

HL7 Health Level 7

IHE Integrating the Healthcare Enterprise

LDB Local Database

MAS Multi-Agent System

OMA Open Mobile Alliance

RSDB Research Database

TC Tuple Centre

TDM Therapeutic Drug Monitoring

VCS Version Control System

XCA Cross-Community Access

XCPD Cross-Community Patient Discovery

XDS Cross-Document Sharing

Appendix B

Misc. scripts and documents

B.1 systemd[1] init script for the TuCSoN service

tucson.service

```
[Unit]
Description=TuCSoN Node Service
After=network.target auditd.service

[Service]
Environment="TUCSON_DIR=/root/devel/tucson/TuCSoN/rel/TuCSoN-1.12.0.0301/rel"
ExecStart=/usr/bin/java -cp ${TUCSON_DIR}/tucson.jar:${TUCSON_DIR}/../libs/2p.
    jar alice.tucson.service.TucsonNodeService
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure

[Install]
WantedBy=multi-user.target
Alias=tucson.service
```

tucson.socket

```
[Unit]
Description=TuCSoN Node Service socket
Before=tucson.service
Conflicts=tucson.service

[Socket]
ListenStream=20504
Accept=yes

[Install]
WantedBy=sockets.target
```

B.2 bash script to start the nodes

start_nodes.sh

```
#!/bin/bash

VMS="tucson1.virtualbox tucson2.virtualbox tucson3.virtualbox"
#VMS="tucson1.virtualbox tucson2.virtualbox tucson3.virtualbox tucson4.virtualbox tucson5.virtualbox"
#VMS="tucson1.virtualbox tucson2.virtualbox tucson3.virtualbox tucson4.virtualbox tucson5.virtualbox tucson6.virtualbox tucson7.virtualbox tucson8.virtualbox tucson9.virtualbox tucson10.virtualbox"

if [ "$#" -ne 1 ]; then
    echo "Illegal number of parameters"
fi

LOG_PREFIX=$1

for vm in $VMS
do
    ssh root@$vm "cd /media/sf_Shared;java -jar dynamic.jar -vv" &> ${LOG_PREFIX}_$vm &
done
```

B.3 Python script to reset all the Orion Context Broker publications and subcriptions

cb_clear.py

```
#!/bin/env python

import requests, json
from pymongo import MongoClient

ORION_URL = "http://localhost:1026/"

def removeSubscription(url,subsID):
    headers = {'Content-Type' : 'application/json', 'Accept' : 'application/json' }
        data = '{ "subscriptionId": "' + str(subsID) + '" }'
        r = requests.post(url + "v1/unsubscribeContext",data,headers=headers)
        response = r.json()
        if(response['statusCode']['code'] != '200'):
              print "Error deleting context" + contextName

def removeContext(url,contextName):
        headers = {'Content-Type' : 'application/json', 'Accept' : 'application/json' }
    data = '{ "contextElements": [{ "type": "case", "isPattern": "false","id": "' + contextName + '" }], "updateAction": "DELETE"}'
    r = requests.post(url + "v1/updateContext",data,headers=headers)
```

```
    response = r.json()
    if(response['contextResponses'][0]['statusCode']['code'] != '200'):
        print "Error deleting context" + contextName

def getEntities(url):
    headers = {'Content-Type' : 'application/json', 'Accept' : 'application/
        json' }
    r = requests.get(url + "v1/contextEntities?limit=100",headers = headers)
    return r.json()

def getSubs(url):
    client = MongoClient('localhost',27017)
    db = client['orion']
    return db['csubs'].find()


response_string = getEntities(ORION_URL)
if('contextResponses' in response_string):
    for response in response_string['contextResponses']:
        removeContext(ORION_URL,response['contextElement']['id'])

subs = getSubs(ORION_URL)
for sub in subs:
    removeSubscription(ORION_URL,sub['_id'])
```

B.4 Orion Context Broker operation examples

Query context operation:

```
http://orion.virtualbox:1026/NGSI10/queryContext

-- QUERY --
{
    "entities": [
        {
            "type": "case",
            "isPattern": "false",
            "id": "Gentamicin"
        }
    ]
}

-- RESPONSE --

{
  "contextResponses" : [
    {
      "contextElement" : {
        "type" : "case",
        "isPattern" : "false",
        "id" : "Gentamicin",
        "attributes" : [
```

```
        {
          "name" : "Node 1",
          "type" : "node",
          "value" : "address~tucson1.virtualbox|numcases~800|timestamp
             ~2015-06-02 11:39:57.000"
        },
        {
          "name" : "description",
          "type" : "string",
          "value" : "Small collection on gentamicin concetration on neonates"
        },
        {
          "name" : "numNodes",
          "type" : "int",
          "value" : "1"
        },
        {
          "name" : "Node 2",
          "type" : "node",
          "value" : "address~tucson2.virtualbox|numcases~800|timestamp
             ~2015-06-02 11:39:57.000"
        },
        {
          "name" : "Node 3",
          "type" : "node",
          "value" : "address~tucson3.virtualbox|numcases~800|timestamp
             ~2015-06-02 11:39:57.000"
        }
       ]
     },
     "statusCode" : {
       "code" : "200",
       "reasonPhrase" : "OK"
     }
   }
  ]
}
```

Update context operation:

```
http://orion.virtualbox:1026/NGSI10/updateContext

-- QUERY --

{
   "contextElements": [
      {
         "type": "case",
         "isPattern": "false",
         "id": "Malaria",
         "attributes": [
            {
               "name": "Node 1",
               "type": "node",
```

```
15                "value": "address~tucson1.virtualbox|numcases~15|timestamp
                     ~2015-03-09 17:09:49.566"
16             },
17             {
18                "name": "Node 2",
19                "type": "node",
20                "value": "address~tucson2.virtualbox|numcases~45|timestamp
                     ~2015-03-08 17:09:49.566"
21             },
22             {
23                "name": "description",
24                "type": "string",
25                "value": "Patients affected by malaria"
26             },
27             {
28                "name": "numNodes",
29                "type": "int",
30                "value": "2"
31             }
32          ]
33       }
34    ],
35    "updateAction": "APPEND"
36 }
37
38 -- RESPONSE --
39
40 {
41    "contextResponses": [
42       {
43          "contextElement": {
44          "attributes": [
45             {
46                "name": "Node 1",
47                "type": "node",
48                "value": ""
49             },
50             {
51                "name": "Node 2",
52                "type": "node",
53                "value": ""
54             },
55             {
56                "name": "description",
57                "type": "string",
58                "value": ""
59             },
60             {
61                "name": "numNodes",
62                "type": "int",
63                "value": ""
64             }
65          ],
66          "id": "Malaria",
```

```
67          "isPattern": "false",
68          "type": "case"
69          },
70          "statusCode": {
71             "code": "200",
72             "reasonPhrase": "OK"
73          }
74       }
75    ]
76 }
```

Subscribe context operation:

```
1  http://orion.virtualbox:1026/v1/subscribeContext
2
3  -- QUERY --
4  {
5     "duration": "P1M",
6     "reference": "http:\/\/tucson1.virtualbox:8000",
7     "notifyConditions": [
8        {
9           "condValues": ["Node 1"],
10          "type": "ONCHANGE"
11       },
12       {
13          "condValues": ["description"],
14          "type": "ONCHANGE"
15       },
16       {
17          "condValues": ["numNodes"],
18          "type":"ONCHANGE"
19       },
20       {
21          "condValues": ["Node 2"],
22          "type":"ONCHANGE"
23       },
24       {
25          "condValues": ["Node 3"],
26          "type":"ONCHANGE"
27       }
28    ],
29    "entities": [
30       {
31          "id": "Gentamicin",
32          "type": "case",
33          "isPattern": false
34       }
35    ]
36 }
37
38 -- RESPONSE --
39
40 {
41   "subscribeResponse" : {
```

```
    "subscriptionId" : "55b0fc2f985d838e3a13d1cd",
    "duration" : "P1M"
  }
}
```

Unsubscribe context operation:

```
http://orion.virtualbox/v1/unsubscribeContext

-- QUERY --

{
  "subscriptionId": "51c04a21d714fb3b37d7d5a7"
}

-- RESPONSE --

{
   "statusCode": {
      "code": "200",
      "reasonPhrase": "OK"
   },
   "subscriptionId": "51c04a21d714fb3b37d7d5a7"
}
```

Appendix C

Test results data tables (CSV format)

C.1 Dynamic test with 3 agents and 20 queries (10 runs)

```
Run,Agent1,Agent2,Agent3
1,-6,28,-22
2,196,-132,-64
3,401,-103,-298
4,137,255,-392
5,36,204,-240
6,-248,139,109
7,152,-101,-51
8,248,111,-359
9,-69,-92,161
10,-303,185,118
AVERAGE,54.4,49.4,-103.8
```

C.2 Dynamic test with 5 agents and 20 queries (10 runs)

```
Run,Agent1,Agent2,Agent3,Agent4,Agent5
1,71,134,-602,299,98
2,358,-496,-210,8,340
3,-46,-457,125,106,272
4,-429,112,330,435,-448
5,134,34,45,-544,331
6,167,-383,333,-381,264
7,450,322,-207,-99,-466
8,-665,-30,113,324,258
9,232,121,60,-677,264
10,-467,396,101,-65,35
AVERAGE,-19.5,-24.7,8.8,-59.4,94.8
```

C.3 Dynamic test with 10 agents and 20 queries (10 runs)

```
Run,Agent1,Agent2,Agent3,Agent4,Agent5,Agent6,Agent7,Agent8,Agent9,Agent10
1,-78,434,260,75,663,69,690,-573,-1044,-496
2,-1188,643,69,-460,129,-430,806,174,228,29
3,-1015,296,388,1018,-726,515,-713,417,-214,34
4,-1061,436,146,361,300,-198,-33,329,-139,-141
5,64,411,-223,168,560,21,-1363,4,-271,629
6,-627,326,1231,96,265,344,-938,598,-663,-632
7,78,729,447,416,362,-1076,86,204,402,-1648
8,475,546,138,-894,258,-852,191,120,863,-845
9,509,774,-1478,-717,205,643,819,646,109,-1510
10,-260,560,-1275,110,149,443,130,175,126,-158
AVERAGE,-310.3,515.5,-29.7,17.3,216.5,-52.1,-32.5,209.4,-60.3,-473.8
```

C.4 Dynamic test with 10 queries and 10 agents (10 runs)

```
Run,Agent1,Agent2,Agent3,Agent4,Agent5,Agent6,Agent7,Agent8,Agent9,Agent10
1,102,475,133,178,163,-1244,61,-234,251,115
2,280,-78,-12,224,-812,180,-420,341,397,-100
3,225,386,-624,289,-55,291,-38,14,307,-795
4,223,381,336,-1025,-1025,141,261,320,199,189
5,-310,385,366,392,304,-1056,-1060,314,425,240
6,-167,-853,-807,256,123,248,602,187,227,184
7,22,49,84,137,-7,-64,-354,139,-6,0
8,48,-506,63,69,-8,107,81,433,148,-435
9,32,-808,320,151,-452,167,29,526,20,15
10,-869,548,518,387,-856,-117,-503,507,918,-533
AVERAGE,-41.4,-2.1,37.7,105.8,-262.5,-134.7,-134.1,254.7,288.6,-112
```

C.5 Dynamic test with 20 queries and 10 agents (10 runs)

```
Run,Agent1,Agent2,Agent3,Agent4,Agent5,Agent6,Agent7,Agent8,Agent9,Agent10
1,287,99,879,103,528,-830,251,378,-817,-878
2,-975,941,-212,842,755,715,-313,-515,-997,-241
3,-868,290,569,471,307,875,85,276,-1130,-875
4,-740,665,-1071,-347,362,-31,509,584,806,-737
5,140,-254,800,-776,607,-707,-995,497,411,277
6,-897,-841,-900,103,619,18,651,-113,872,488
7,532,78,173,-181,899,262,-1650,-140,-858,885
8,551,-802,258,-913,707,45,530,-1123,108,639
9,178,-986,471,254,307,812,-714,475,-614,-183
10,257,-894,153,338,-219,173,-384,791,609,-824
AVERAGE,-153.5,-170.4,112,-10.6,487.2,133.2,-203,111,-161,-144.9
```

C.6 Dynamic test with 20 queries and 10 agents (100 runs)

```
Run,Agent1,Agent2,Agent3,Agent4,Agent5,Agent6,Agent7,Agent8,Agent9,Agent10
1,343,354,-207,397,-64,-1075,679,-391,39,-75
```

```
2,204,-1675,213,-819,583,131,320,82,643,318
3,-422,-534,762,655,216,471,332,-503,-474,-503
4,418,15,-1589,962,77,404,120,-1060,442,211
5,-803,247,486,173,152,-1125,-325,589,334,272
6,813,-1456,-591,549,550,-689,-1371,1072,217,906
7,-600,-165,-555,-893,-669,373,858,541,963,147
8,240,-212,802,-742,241,755,79,-252,-719,-192
9,-2,558,14,151,446,442,-1316,-1004,221,490
10,347,-442,74,3,103,-309,65,-130,35,254
11,-417,827,-309,-1150,87,652,87,-1146,581,788
12,360,-14,648,-52,342,-27,-611,-331,-286,-29
13,356,500,-167,497,-118,-892,246,-664,371,-129
14,183,620,139,44,226,-1011,430,-98,-13,-520
15,-951,253,258,148,331,-73,475,240,552,-1233
16,56,105,-1737,572,169,139,884,-919,529,202
17,-1657,879,284,557,-413,-397,245,324,302,-124
18,-1083,139,805,-614,580,-1041,237,418,33,526
19,-596,305,-10,599,339,-894,489,530,623,-1385
20,679,-648,-617,545,422,-660,123,769,-1356,743
21,754,-1481,-676,475,222,212,320,898,52,-776
22,34,40,-729,697,142,221,139,-802,256,2
23,-1088,-315,515,699,335,-769,-59,140,296,246
24,-341,657,636,183,708,-1541,-755,-824,705,572
25,-360,287,38,639,708,103,324,-893,-1580,734
26,546,4,366,-910,172,-953,163,-13,136,489
27,-479,-978,-316,915,567,-937,-475,-474,999,1178
28,654,106,678,480,194,422,-69,-607,-1831,-27
29,-176,-1723,116,14,132,345,497,113,447,235
30,195,-1013,299,289,-751,-802,-306,624,473,992
31,151,873,636,85,-836,-914,280,559,-878,44
32,-207,-721,-512,901,-978,869,501,306,-616,457
33,-11,-1709,418,256,449,153,-78,299,-171,394
34,-641,-878,-562,376,360,1050,-760,313,470,272
35,-357,654,291,-874,178,641,566,-1121,-843,865
36,172,586,283,-833,187,930,108,-1296,454,-591
37,-544,173,-133,-1078,135,334,746,-189,84,472
38,-1633,593,74,434,663,691,362,-1674,-117,607
39,605,-211,159,140,254,650,-1182,-816,162,239
40,-353,1028,-793,306,206,-1096,168,227,185,122
41,110,164,-1027,-95,39,-623,297,125,676,334
42,505,788,295,-252,-73,98,-466,-473,520,-942
43,-805,959,229,436,978,-791,698,122,-316,-1510
44,-93,-841,-451,418,-575,-605,330,809,421,587
45,-378,-323,721,-1567,14,446,216,582,-36,325
46,982,177,665,1058,4,-391,-460,-1173,-413,-449
47,1087,-1431,767,453,-672,257,523,741,-1014,-711
48,558,504,346,-577,-1299,-1393,1134,-212,405,534
49,-950,-668,752,215,738,230,284,765,-699,-667
50,-933,124,614,38,534,-451,556,-956,440,34
51,512,-825,628,-423,196,917,676,906,-1072,-1515
52,-468,263,245,194,0,430,193,279,-1672,536
53,552,-1337,493,290,1189,-1358,-533,-635,780,559
54,642,-1568,-1114,-807,407,902,196,533,78,731
55,540,241,-317,-1152,610,608,246,135,-6,-905
```

```
56,-212,-397,296,-408,-355,426,645,-204,-223,432
57,-942,213,213,406,-1009,465,39,288,706,-379
58,179,254,-759,357,24,-1089,-336,691,236,443
59,492,-104,590,188,332,49,-1564,710,211,-904
60,-255,568,-443,155,327,431,323,-906,-179,-21
61,579,12,914,-483,-13,18,908,608,-890,-1653
62,-308,279,426,386,334,-310,-770,655,-802,110
63,34,-264,219,215,-1019,33,562,-206,251,175
64,712,-14,512,-1036,94,497,-1671,565,481,-140
65,661,574,542,376,-1380,-589,879,114,-613,-564
66,1046,-584,449,1189,-561,-1255,-457,-28,-491,692
67,-91,86,377,-63,-29,515,50,-1318,-77,550
68,-1138,12,-669,-87,384,213,660,-304,582,347
69,60,-845,62,-1123,137,304,7,228,291,879
70,479,597,605,-768,-832,-19,324,-465,-471,550
71,-86,74,726,397,-324,169,416,-1019,-171,-182
72,1010,-273,-916,-954,-262,-525,961,-488,685,762
73,932,541,383,-624,-875,-76,817,-534,-536,-28
74,-1467,45,-1047,785,296,308,-990,796,877,397
75,309,-1428,-122,702,864,-1004,1048,-1054,307,378
76,-663,473,-573,-757,296,195,112,127,120,670
77,300,435,375,-10,610,85,-1572,-705,180,302
78,433,-749,-1497,751,239,594,158,-784,470,385
79,99,387,730,246,32,-1028,-1008,497,-191,236
80,-622,-144,732,-984,-63,884,-878,841,-568,802
81,469,-494,-1070,-424,959,-444,721,466,258,-441
82,-1,-513,-414,436,-484,-482,553,-731,705,931
83,-223,400,221,-1419,471,63,63,9,372,43
84,157,368,-1520,696,650,303,-122,116,287,-935
85,-967,18,7,217,244,0,181,61,653,-414
86,-89,215,473,-142,-1330,-297,14,745,99,312
87,-458,-564,836,-828,968,-479,646,-729,320,288
88,130,518,-990,-36,328,144,377,-733,-69,331
89,492,-248,254,-973,199,-48,-131,462,-205,198
90,-759,-1074,-699,599,357,-682,700,462,596,500
91,192,-660,-1386,22,-667,-1364,1017,1138,936,772
92,769,-1637,-147,192,594,79,-198,372,-39,15
93,-116,-1592,229,690,131,-333,651,181,-805,964
94,-357,69,802,-309,287,-772,306,168,119,-313
95,84,733,967,-1584,-728,611,-703,615,-270,275
96,-15,-802,328,274,731,822,-1484,-763,479,430
97,-76,-507,-1264,422,910,364,1039,873,-1253,-508
98,369,36,-999,340,12,-191,468,214,-135,-114
99,319,425,-1440,313,-410,886,332,15,-848,408
100,-631,297,-478,148,807,-593,478,-132,-631,528
AVERAGE,-29.2,-134.09,-18.28,4.95,100.13,-100.63,99.97,-47.01,5.66,116.43
```

Appendix D

Implementation Code

D.1 ch.hevs.ISyPeM2.ContextBroker package

Attribute.java

```
1  package ch.hevs.ISyPeM2.ContextBroker;
2
3  /**
4  * This abstract class defines an {@code Attribute} of a Context Element from
       the Orion Context Broker.
5  * @author Imanol-Mikel Barba Sabariego
6  */
7  public abstract class Attribute
8  {
9     /** Name of the {@code Attribute} */
10    private String name;
11    /** Type of the {@code Attribute} */
12    private String type;
13
14    /**
15     * Class constructor
16     * @param name Name of the {@code Attribute}
17     * @param type Type of the {@code Attribute}
18     */
19    public Attribute(String name, String type)
20    {
21       this.name = name;
22       this.type = type;
23    }
24
25    /**
26     * {@link #name} getter
27     * @return Name of the {@code Attribute}
28     */
29    public String getName()
30    {
31       return name;
32    }
33
```

```
34      /**
35       * {@link #type} getter
36       * @return Type of the {@code Attribute}
37       */
38      public String getType()
39      {
40         return type;
41      }
42
43      /**
44       * Sets (or adds) the value of the {@code Attribute}
45       * @param name Name of the attribute to set (if applicable)
46       * @param value Value to be set
47       */
48      public abstract void setValue(String name, String value);
49
50      /**
51       * Retrieves the value (or values) of the {@code Attribute}
52       * @return the value (or values) this {@code Attribute} has
53       */
54      public abstract String getValue();
55  }
```

ContextBroker.java

```
1  package ch.hevs.ISyPeM2.ContextBroker;
2
3  import java.util.HashMap;
4  import java.util.HashSet;
5  import java.util.Iterator;
6  import java.util.Map;
7  import java.util.Map.Entry;
8  import java.util.concurrent.locks.Lock;
9  import java.util.concurrent.locks.ReentrantLock;
10
11 import org.json.simple.JSONArray;
12 import org.json.simple.JSONObject;
13 import org.json.simple.parser.JSONParser;
14 import org.json.simple.parser.ParseException;
15
16 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
17 import ch.hevs.ISyPeM2.DB.LocalCaseProvisioner;
18 import ch.hevs.ISyPeM2.Node.CaseAggregation;
19 import ch.hevs.ISyPeM2.Node.Node;
20 import ch.hevs.se.IoT6.T6_5.OrionInterface.Boundary.PublisherEntity;
21 import ch.hevs.se.IoT6.T6_5.OrionInterface.Boundary.QueryEntity;
22 import ch.hevs.se.IoT6.T6_5.OrionInterface.Boundary.SubscriberEntity;
23 import ch.hevs.se.IoT6.T6_5.OrionInterface.Entity.PublicationData;
24 import ch.hevs.se.IoT6.T6_5.OrionInterface.Entity.QueryData;
25 import ch.hevs.se.IoT6.T6_5.OrionInterface.Entity.SubscriptionData;
26 import ch.hevs.se.IoT6.T6_5.OrionInterface.Entity.UnsubscriptionData;
27
28 import com.sun.jersey.api.client.WebResource;
29
30 /**
31  * This class represents an instance of the Orion Context Broker
32  * @author Imanol-Mikel Barba Sabariego
33  *
34  */
35 public class ContextBroker
36 {
37    /**
38     * URL where the Orion Context Broker instance can be reached
39     */
40    private String url;
41
42    /**
43     * The {@code Node} owning this {@code ContextBroker} instance. All the
          interactions with the context broker will be
44     * made on behalf this node.
45     */
46    private Node ownNode;
47
48    /**
49     * {@code Map} storing all the active subscriptions of the node with this
          context broker, using the subscription ID
50     * as key
51     */
```

```
52      private HashMap<String,Subscription> activeSubscriptions;
53
54      /**
55       * {@code Lock} to perform mutually exclusive operations on the context
             broker
56       */
57      private Lock lock;
58
59      /**
60       * Class constructor
61       * @param url URL of the Orion Context Broker
62       * @param node {@code Node} object representing the local node, the one
             that operates on this instance
63       */
64      public ContextBroker(String url, Node node)
65      {
66         this.url = url;
67         this.ownNode = node;
68         this.activeSubscriptions = new HashMap<String,Subscription>();
69         this.lock = new ReentrantLock();
70      }
71
72      /**
73       * Acquires the {@code Lock} to ensure mutual exclusion between critical
             methods.
74       *
75       * <p>Due to the fact that {@link #lock} is instantiated as a {@code
             ReentrantLock}, this method may be called repeatedly
76       * as long as the lock is owned by the current thread. In each critical
             method, the lock is locked and unlocked using
77       * the {@link #startAtomicOperation()} and {@link #stopAtomicOperation()}
             methods, but if we start an atomic operation
78       * before, we can perform multiple critical operations without releasing
             the lock</p>
79       */
80      public synchronized void startAtomicOperation()
81      {
82         lock.lock();
83      }
84
85      /**
86       * Frees the {@code Lock} to finish mutual exclusion situations between
             methods.
87       */
88      public void stopAtomicOperation()
89      {
90         lock.unlock();
91      }
92
93      /**
94       * This method is called whenever a notification is sent to this {@code
             ContextBroker} instance. It checks if the
95       * notifications is from any of the active subscriptions and if so, it
             relays it to the local node running this instance.
```

```
96      * @param notification JSON-encoded string containing the notification from
            the Orion context broker
97      */
98     public void processNotification(String notification)
99     {
100       try
101       {
102          String subsID = (String)((JSONObject)new JSONParser().parse(
               notification)).get("subscriptionId");
103          if(activeSubscriptions.containsKey(subsID))
104          {
105                ownNode.notifyNode(notification,activeSubscriptions.get(subsID)
                   ,this);
106          }
107       }
108       catch (ParseException e)
109       {
110          ErrorPrinter.printLog("ContextBroker", "processNotification",
               ErrorPrinter.SEVERITY_WARNING, e.getClass() + ": " + e.getMessage
               ());
111       }
112    }
113
114    /**
115     * This method checks if a context element is already registered in the
            context broker
116     *
117     * @param name Name of the context element
118     * @return {@code true} if the context element is already registered, {
            @code false} otherwise
119     */
120    private boolean isElementRegistered(String name)
121    {
122       Element elem = new Element("case",name);
123       return parseAnswer(queryData(generateQueryData(elem)));
124    }
125
126    /**
127     * This method checks if a node is already registered as publisher of a
            certain type of case aggregation
128     * @param node Node to be checked
129     * @param caseName Name of the case aggregation
130     * @return {@code true} if the node is already registered, {@code false}
            otherwise
131     */
132    private boolean isNodeRegistered(Node node, String caseName)
133    {
134       Element elem = new Element("case",caseName);
135       elem.addAttribute(new NodeAttribute(node.getName(),"node"));
136       return parseAnswer(queryData(generateQueryData(elem)));
137    }
138
139    /**
140     * Checks an answer from the Orion Context Broker for errors
```

```
141      * @param answer JSON-formatted answer from the Context Broker to be
             checked
142      * @return {@code false} iff the answer contains an {@code "errorCode"}
             JSON key or the string is malformed
143      */
144     private boolean parseAnswer(String answer)
145     {
146        JSONParser parser = new JSONParser();
147        try
148        {
149           JSONObject obj = (JSONObject)parser.parse(answer);
150           if(obj.containsKey("errorCode"))
151           {
152              JSONObject errorCode = (JSONObject) obj.get("errorCode");
153              ErrorPrinter.printLog("ContextBroker", "parseAnswer", ErrorPrinter
                     .SEVERITY_INFO, "Server returned error " + errorCode.get("code"
                     ) + " - " + errorCode.get("reasonPhrase"));
154              return false;
155           }
156           return true;
157        }
158        catch(ParseException ex)
159        {
160           return false;
161        }
162     }
163
164     /**
165      * Retrieves the "numNodes" attribute from a context element representing a
             published case aggregation, which indicates
166      * the number of nodes that offer the case aggregation
167      *
168      * @param name Name of the case aggregation
169      * @return {@code 0} if the case aggregation isn't offered by any nodes or
             if the case aggregation doesn't exist,
170      * otherwise it returns the number of nodes offering the case aggregation
171      */
172     private int getNumNodes(String name)
173     {
174        Element elem = new Element("case",name);
175        elem.addAttribute(new RegularAttribute("numNodes","int"));
176        String answer = queryData(generateQueryData(elem));
177        int numNodes = 0;
178        //WARNING: INTENSE casting ahead
179        if(parseAnswer(answer))
180        {
181           try
182           {
183              numNodes = Integer.parseInt(((String)((JSONObject)((JSONArray)((
                     JSONObject)((JSONObject)((JSONArray)((JSONObject)new JSONParser
                     ().parse(answer)).get("contextResponses")).get(0)).get("
                     contextElement")).get("attributes")).get(0)).get("value")).
                     replaceAll("numNodes~", ""));
184           }
```

```
185             catch (ParseException e)
186             {
187                 ErrorPrinter.printLog("ContextBroker", "getNumNodes", ErrorPrinter
                        .SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
188             }
189         }
190         return numNodes;
191     }
192 
193     /**
194      * Retrieves the list of active subscriptions in this instance of the Orion
                context broker
195      * @return {@code Map} with all the subscriptions in the context broker
             using their subscription ID as key
196      */
197     public HashMap<String,Subscription> getActiveSubscriptions()
198     {
199         return this.activeSubscriptions;
200     }
201 
202     /**
203      * This method attempts to subscribe to a context element.
204      * @param caseName Name of the context element to subscribe
205      * @param responseURL URL where to receive the notifications from the
             Context Broker for this subscription
206      * @return {@code true} if the operation was successful, {@code false}
             otherwise
207      */
208     public boolean subscribeContext(String caseName, String responseURL)
209     {
210         startAtomicOperation();
211         boolean success = true;
212         Element elem = new Element("case",caseName);
213         String answer = queryData(this.generateQueryData(elem));
214         boolean foundNumNodes = false;
215         boolean foundDescription = false;
216         if(parseAnswer(answer))
217         {
218             //WARNING: INTENSE casting ahead
219             try
220             {
221                 JSONObject contextElement = (JSONObject)((JSONObject)((JSONArray)
                        ((JSONObject)new JSONParser().parse(answer)).get("
                        contextResponses")).get(0)).get("contextElement");
222                 if(contextElement.containsKey("attributes"))
223                 {
224                     JSONArray attrArray = (JSONArray)contextElement.get("attributes
                            ");
225                     for(int i = 0; i < attrArray.size(); i++)
226                     {
227                         try
228                         {
229                             JSONObject attrInfo = (JSONObject)attrArray.get(i);
230                             Attribute attr;
```

```
231                     if(((String)attrInfo.get("type")).equals("node"))
232                     {
233                        attr = new NodeAttribute((String)attrInfo.get("name")
                              ,(String)attrInfo.get("type"));
234                        HashMap<String,String> values = ((NodeAttribute)attr).
                              parseValues((String)attrInfo.get("value"));
235                        Iterator<Entry<String, String>> it = values.entrySet()
                              .iterator();
236                        while(it.hasNext())
237                        {
238                           Map.Entry<String,String> pair = it.next();
239                           attr.setValue(pair.getKey(), pair.getValue());
240                        }
241                     }
242                     else
243                     {
244                        if(((String)attrInfo.get("name")).equals("description"
                              ))
245                        {
246                           foundDescription = true;
247                        }
248                        else if(((String)attrInfo.get("name")).equals("
                              numNodes"))
249                        {
250                           foundNumNodes = true;
251                        }
252                        attr = new RegularAttribute((String)attrInfo.get("name
                              "),(String)attrInfo.get("type"));
253                        attr.setValue("value",(String)attrInfo.get("value"));
254                     }
255                     elem.addAttribute(attr);
256                  }
257                  catch (java.text.ParseException e)
258                  {
259                     //Error is already printed in parseValues if exception
                           occurs
260                     success = false;
261                  }
262               }
263               if(foundNumNodes && foundDescription)
264               {
265                  answer = subscribeData(generateSubscriptionData(elem,
                        responseURL, elem.getAttributes()));
266                  if(parseAnswer(answer))
267                  {
268                     Subscription subs = new Subscription((String)((JSONObject
                           )((JSONObject)new JSONParser().parse(answer)).get("
                           subscribeResponse")).get("subscriptionId"),elem,
                           responseURL);
269                     activeSubscriptions.put(subs.getID(), subs);
270                  }
271                  else
272                  {
273                     success = false;
```

```
274                     }
275                 }
276                 else
277                 {
278                     ErrorPrinter.printLog("ContextBroker", "subscribeContext",
                            ErrorPrinter.SEVERITY_ERROR, "Requested context element
                            doesn't have description and/or numNodes attributes");
279                     success = false;
280                 }
281             }
282             else
283             {
284                 ErrorPrinter.printLog("ContextBroker", "searchRecords",
                        ErrorPrinter.SEVERITY_WARNING, "Can't subscribe to context
                        element: Context element doesn't have attributes");
285                 success = false;
286             }
287         }
288         catch(org.json.simple.parser.ParseException exc)
289         {
290             ErrorPrinter.printLog("ContextBroker", "searchRecords",
                    ErrorPrinter.SEVERITY_ERROR, exc.getMessage());
291             success = false;
292         }
293     }
294     else
295     {
296         ErrorPrinter.printLog("ContextBroker", "searchRecords", ErrorPrinter.
                SEVERITY_WARNING, "Can't subscribe to context element: Context
                element doesn't exist");
297         success = false;
298     }
299     stopAtomicOperation();
300     return success;
301 }
302
303 /**
304  * This method attempts to unsubscribe from a context element.
305  * @param subscriptionID Name of the context element to unsubscribe
306  * @return {@code true} if the operation was successful, {@code false}
        otherwise
307  */
308 public boolean unsubscribeContext(String subscriptionID)
309 {
310     startAtomicOperation();
311     boolean success = false;
312     if(activeSubscriptions.containsKey(subscriptionID))
313     {
314         success = parseAnswer(unsubscribe(generateUnsubscriptionData(
                subscriptionID)));
315         if(success)
316         {
317             this.activeSubscriptions.remove(subscriptionID);
318         }
```

```
319         }
320         stopAtomicOperation();
321         return success;
322     }
323 
324     /**
325      * Queries the Context Broker for a list of all the nodes providing a
             certain case aggregation
326      * @param caseName Name of the case aggregation
327      * @return {@code Map} of all the nodes providing the case aggregation and
             the number of cases it has for that
328      * case aggregation
329      */
330     public HashMap<Node,Integer> searchRecords(String caseName)
331     {
332         HashMap<Node,Integer> nodeList = new HashMap<Node,Integer>();
333         String description = "No description available";
334         String answer = queryData(this.generateQueryData(new Element("case",
                caseName)));
335         if(parseAnswer(answer))
336         {
337             //WARNING: INTENSE casting ahead
338             try
339             {
340                 JSONObject contextElement = (JSONObject)((JSONObject)((JSONArray)
                        ((JSONObject)new JSONParser().parse(answer)).get("
                        contextResponses")).get(0)).get("contextElement");
341                 if(contextElement.containsKey("attributes"))
342                 {
343                     JSONArray attrArray = (JSONArray)contextElement.get("attributes
                            ");
344                     //First we iterate to get the description
345                     for(int i = 0; i < attrArray.size(); i++)
346                     {
347                         JSONObject attribute = (JSONObject)attrArray.get(i);
348                         if(attribute.get("type").equals("string") && attribute.get("
                                name").equals("description"))
349                         {
350                             description = attribute.get("value").toString();
351                         }
352                     }
353                     if(description.equals("No description available"))
354                     {
355                         ErrorPrinter.printLog("ContextBroker", "searchRecords",
                                ErrorPrinter.SEVERITY_WARNING, "Description not found for
                                 context element " + caseName);
356                     }
357                     for(int i = 0; i < attrArray.size(); i++)
358                     {
359                         JSONObject nodeInfo = (JSONObject)attrArray.get(i);
360                         if(!nodeInfo.get("type").equals("node"))
361                         {
362                             //This is not a node
363                             continue;
```

```
364                     }
365                     try
366                     {
367                        String nodeName = (String)nodeInfo.get("name");
368                        HashMap<String,String> values = new NodeAttribute(
                              nodeName,"node").parseValues((String)nodeInfo.get("
                              value"));
369                        Node node = new Node(nodeName,values.get("address"));
370                        node.setProvisioner(new LocalCaseProvisioner());
371                        CaseAggregation aggregation = new CaseAggregation(
                              caseName, description);
372                        aggregation.setTimestamp(values.get("timestamp"));
373                        node.addAggregation(aggregation);
374                        nodeList.put(node,Integer.parseInt(values.get("numcases")
                              ));
375                     }
376                     catch (java.text.ParseException exc)
377                     {
378                        //Error is already printed in parseValues if exception
                              occurs
379                     }
380                  }
381               }
382            }
383            catch(org.json.simple.parser.ParseException exc)
384            {
385               ErrorPrinter.printLog("ContextBroker", "searchRecords",
                     ErrorPrinter.SEVERITY_ERROR, exc.getMessage());
386               return nodeList;
387            }
388         }
389         return nodeList;
390      }
391
392      /**
393       * This method changes a published case aggregation's description
394       * @param caseName Name of the case aggregation
395       * @return {@code true} if the operation is successful, {@code false}
             otherwise
396       */
397      public boolean updateRecordDescription(String caseName)
398      {
399         Element elem = new Element("case",caseName);
400         String action = "UPDATE";
401         Attribute attr = new RegularAttribute("description","string");
402         attr.setValue("description", this.ownNode.getAggregation(caseName).
               getDescription());
403         elem.addAttribute(attr);
404         PublicationData pubData = generatePublicationData(elem,action);
405         return parseAnswer(publishData(pubData));
406      }
407
408      /**
409       * Publishes a certain type of case aggregation to the context broker
```

```
410      *
411      * <p>NOTE: This does not actually publish the data</p>
412      * @param cases {@code Map} with the names of each case aggregation that is
             going to be published and the number of
413      * cases the node currently has
414      * @return {@code true} if the process was successful, {@code false}
            otherwise
415      */
416     public boolean publishRecords(HashMap<String,Integer> cases)
417     {
418        boolean success = true;
419        Iterator<Entry<String, Integer>> it = cases.entrySet().iterator();
420        while(it.hasNext()) //Each case is a context element
421        {
422           Map.Entry<String,Integer> pair = it.next();
423           String caseName = pair.getKey();
424           Element elem = new Element("case",caseName);
425           String action = "APPEND";
426           Attribute attr;
427           if(isElementRegistered(caseName))
428           {
429              if(isNodeRegistered(this.ownNode,caseName)) //Each node is an
                    attribute
430              {
431                 //ContextElement exists and Node is registered
432                 /*
433                  * Imanol Barba:
434                  * Being pretty strict here, the CB as is reinterprets an
                        APPEND
435                  * operation as an UPDATE operation if the attribute already
                        exists.
436                  * But that may not be necessarily valid in future versions.
437                  */
438                 action = "UPDATE";
439              }
440              else
441              {
442                 ErrorPrinter.printLog("ContextBroker", "searchRecords",
                       ErrorPrinter.SEVERITY_INFO, "Context element found, but node
                        is not registered. Registering node...");
443                 attr = new RegularAttribute("numNodes","int");
444                 attr.setValue("numNodes", Integer.toString(getNumNodes(caseName
                       )+1));
445                 elem.addAttribute(attr);
446              }
447           }
448           else
449           {
450              ErrorPrinter.printLog("ContextBroker", "searchRecords",
                    ErrorPrinter.SEVERITY_INFO, "Context element not found.
                    Creating context element...");
451              attr = new RegularAttribute("numNodes","int");
452              attr.setValue("numNodes", Integer.toString(1));
453              elem.addAttribute(attr);
```

```
454             attr = new RegularAttribute("description","string");
455             attr.setValue("description", ownNode.getAggregation(caseName).
                    getDescription());
456             elem.addAttribute(attr);
457          }
458          attr = new NodeAttribute(ownNode.getName(),"node");
459          attr.setValue("address", ownNode.getAddress());
460          attr.setValue("numcases", pair.getValue().toString());
461          attr.setValue("timestamp", ownNode.getLastUpdate(pair.getKey()));
462          elem.addAttribute(attr);
463          PublicationData pubData = generatePublicationData(elem,action);
464          boolean result = parseAnswer(publishData(pubData));
465          if(success)
466          {
467             success = result;
468             this.ownNode.addPublishedAggregation(caseName);
469          }
470       }
471       return success;
472    }
473
474    /**
475     * Removes a certain type of case aggregation from the context broker
476     * @param cases {@code Set} with the names of each case aggregation that is
              going to be unpublished
477     * @return {@code true} if the process was successful, {@code false}
            otherwise
478     */
479    public boolean deleteRecords(HashSet<String> cases)
480    {
481       boolean success = true;
482       Iterator<String> it = cases.iterator();
483       while(it.hasNext()) //Each case is a context element
484       {
485          String caseName = it.next();
486          if(!isNodeRegistered(ownNode,caseName))
487          {
488             continue;
489          }
490          Element elem = new Element("case",caseName);
491          String action = "DELETE";
492
493          Attribute attr = new NodeAttribute(ownNode.getName(),"node");
494          elem.addAttribute(attr);
495          PublicationData pubData = generatePublicationData(elem,action);
496          success = parseAnswer(publishData(pubData));
497
498          if(success)
499          {
500             //Now we modify the numNodes attribute
501             elem = new Element("case",caseName);
502             action = "UPDATE";
503             attr = new RegularAttribute("numNodes","int");
```

```
504             attr.setValue("numNodes", Integer.toString(getNumNodes(caseName)
                    -1));
505             elem.addAttribute(attr);
506             success = parseAnswer(publishData(generatePublicationData(elem,
                    action)));
507             this.ownNode.removePublishedAggregation(caseName);
508          }
509       }
510       return success;
511    }
512
513    /**
514     * Generates data to use the {@code updateContext} operation in the Orion
            context broker
515     * @param elem {@link Element} object that represents the Context Element
            that the operation targets
516     * @param action Action to do by the Context Broker
517     * @return {@code PublicationData} object that holds the JSON query of the
            operation
518     */
519    private PublicationData generatePublicationData(Element elem, String action
          )
520    {
521       PublicationData publishData = new PublicationData();
522       publishData.setElement(elem.getType(), elem.getID());
523       publishData.setAction(action);
524       HashSet<Attribute> attrs = elem.getAttributes();
525       Iterator<Attribute> it = attrs.iterator();
526       while(it.hasNext())
527       {
528          Attribute attr = it.next();
529          publishData.setAttribute(attr.getName(), attr.getType(),attr.getValue
                ());
530       }
531       return publishData;
532    }
533
534    /**
535     * Generates data to use the {@code queryContext} operation in the Orion
            context broker
536     * @param elem {@link Element} object that represents the Context Element
            that the operation targets
537     * @return {@code QueryData} object that holds the JSON query of the
            operation
538     */
539    private QueryData generateQueryData(Element elem)
540    {
541       QueryData queryData = new QueryData();
542       queryData.setEntity(elem.getType(), elem.getID());
543       HashSet<Attribute> attrs = elem.getAttributes();
544       Iterator<Attribute> it = attrs.iterator();
545       while(it.hasNext())
546       {
547          queryData.setAttribute(it.next().getName());
```

```
548         }
549         return queryData;
550     }
551 
552     /**
553      * Generates data to use the {@code subscribeContext} operation in the
           Orion context broker
554      * @param elem {@link Element} object that represents the Context Element
           that the operation targets
555      * @param responseURL URL to be included in the {@code subscribeContext}
           operation for notifications from the context broker
556      * @param attrs {@code Set} of {@link Attribute} objects that the {@code
           subscribeContext} operation will include
557      * @return {@code SubscriptionData} object that holds the JSON query of the
            operation
558      */
559     private SubscriptionData generateSubscriptionData(Element elem, String
           responseURL, HashSet<Attribute> attrs)
560     {
561         SubscriptionData subscribeData = new SubscriptionData();
562         subscribeData.setEntity(elem.getType(), elem.getID());
563         subscribeData.setDuration(1);
564         subscribeData.setReference(responseURL);
565         Iterator<Attribute> it = attrs.iterator();
566         while(it.hasNext())
567         {
568             subscribeData.setNotifyCondition("ONCHANGE", it.next().getName());
569         }
570         return subscribeData;
571     }
572 
573     /**
574      * Generates data to use the {@code unsubscribeContext} operation in the
           Orion context broker
575      * @param subscriptionID Subscription ID against which the {@code
           unsubscribeContext} operation is made
576      * @return {@code UnsubscriptionData} object that holds the JSON query of
           the operation
577      */
578     private UnsubscriptionData generateUnsubscriptionData(String subscriptionID
           )
579     {
580         UnsubscriptionData unsubscriptionData = new UnsubscriptionData();
581         unsubscriptionData.setSubscriptionToBeDeleted(subscriptionID);
582         return unsubscriptionData;
583     }
584 
585     /**
586      * Sends a JSON query to the Context Broker through the {@code
           updateContext} operation
587      * @param publishData {@code PublicationData} object that holds the JSON
           query
588      * @return A string representing the JSON-formatted answer of the Context
           Broker to that query
```

```
589      */
590     private String publishData(PublicationData publishData)
591     {
592        PublisherEntity publisherEntity = new PublisherEntity();
593        WebResource webResource = publisherEntity.connectEntity(this.url);
594        JSONObject contextPayload = publishData.getData();
595        ErrorPrinter.printLog("ContextBroker", "publishData", ErrorPrinter.
               SEVERITY_DEBUG, "Publish:\n" + contextPayload.toString());
596        String answer = publisherEntity.publish(webResource, "updateContext",
               contextPayload);
597        ErrorPrinter.printLog("ContextBroker", "publishData", ErrorPrinter.
               SEVERITY_DEBUG, "Answer:\n" + answer);
598        return answer;
599     }
600
601     /**
602      * Sends a JSON query to the Context Broker through the {@code queryContext
               } operation
603      * @param queryData {@code QueryData} object that holds the JSON query
604      * @return A string representing the JSON-formatted answer of the Context
               Broker to that query
605      */
606     private String queryData(QueryData queryData)
607     {
608        QueryEntity queryEntity = new QueryEntity();
609        WebResource webResource = queryEntity.connectEntity(this.url);
610        JSONObject contextPayload = queryData.getData();
611        ErrorPrinter.printLog("ContextBroker", "queryData", ErrorPrinter.
               SEVERITY_DEBUG, "Query:\n" + contextPayload.toString());
612        String answer = queryEntity.query(webResource, "queryContext",
               contextPayload);
613        ErrorPrinter.printLog("ContextBroker", "queryData", ErrorPrinter.
               SEVERITY_DEBUG, "Answer:\n" + answer);
614        return answer;
615     }
616
617     /**
618      * Sends a JSON query to the Context Broker through the {@code
               subscribeContext} operation
619      * @param subsData {@code SubscriptionData} object that holds the JSON
               query
620      * @return A string representing the JSON-formatted answer of the Context
               Broker to that query
621      */
622     private String subscribeData(SubscriptionData subsData)
623     {
624        SubscriberEntity subsEntity = new SubscriberEntity();
625        WebResource webResource = subsEntity.connectEntity(this.url);
626        JSONObject contextPayload = subsData.getData();
627        ErrorPrinter.printLog("ContextBroker", "subscribe", ErrorPrinter.
               SEVERITY_DEBUG, "Subscription:\n" + contextPayload.toString());
628        String answer = subsEntity.subscribe(webResource, "subscribeContext",
               contextPayload);
```

```
629         ErrorPrinter.printLog("ContextBroker", "subscribe", ErrorPrinter.
                SEVERITY_DEBUG, "Answer:\n" + answer);
630         return answer;
631     }
632 
633     /**
634      * Sends a JSON query to the Context Broker through the {@code
                unsubscribeContext} operation
635      * @param unsubData {@code UnsubscriptionData} object that holds the JSON
                query
636      * @return A string representing the JSON-formatted answer of the Context
                Broker to that query
637      */
638     private String unsubscribe(UnsubscriptionData unsubData)
639     {
640         SubscriberEntity subscriberEntity = new SubscriberEntity();
641         WebResource webResource = subscriberEntity.connectEntity(this.url);
642         JSONObject contextPayload = unsubData.getData();
643         return subscriberEntity.unsubscribe(webResource, "unsubscribeContext",
                contextPayload);
644     }
645 }
```

Element.java

```java
package ch.hevs.ISyPeM2.ContextBroker;

import java.util.HashSet;

/**
 * This class defines a Context Element from the Orion Context Broker
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class Element
{
   /**
    * Type of the {@code Element}
    */
   private String type;
   /**
    * ID of the {@code Element}
    */
   private String ID;
   /**
    * Set of {@link Attribute} objects that represent the Context Element's
       attributes
    */
   private HashSet<Attribute> attributes;

   /**
    * Class constructor
    * @param type Type of the {@code Element}
    * @param id ID of the {@code Element}
    */
   public Element(String type, String id)
   {
      this.ID = id;
      this.type = type;
      this.attributes = new HashSet<Attribute>();
   }

   /**
    * {@link #type} getter
    * @return Type of the {@code Element}
    */
   public String getType()
   {
      return type;
   }

   /**
    * {@link #ID} getter
    * @return ID of the {@code Element}
    */
   public String getID()
   {
      return ID;
```

```
53      }
54
55      /**
56       * Adds an attribute to the {@code Element}
57       * @param attr Attribute to be added
58       */
59      public void addAttribute(Attribute attr)
60      {
61          this.attributes.add(attr);
62      }
63
64      /**
65       * {@link #attributes} getter
66       * @return The collection of attributes the {@code Element} currently has
67       */
68      public HashSet<Attribute> getAttributes()
69      {
70          return attributes;
71      }
72
73  }
```

NodeAttribute.java

```
1  package ch.hevs.ISyPeM2.ContextBroker;
2
3  import java.text.ParseException;
4  import java.util.HashMap;
5  import java.util.Iterator;
6  import java.util.Map;
7  import java.util.Map.Entry;
8
9  import ch.hevs.ISyPeM2.Core.ErrorPrinter;
10
11 /**
12  * This class defines an {@code Attribute} which represents a node in the
        network
13  * @author Imanol-Mikel Barba Sabariego
14  */
15 public class NodeAttribute extends Attribute
16 {
17    /** Values of the {@code Attribute}
18     *
19     * <p>Since the original Orion Interface doesn't contemplate assigning a
           JSON object as value
20     * they are serialized using the {@link #getValue} method and read back
           using the {@link #parseValues} method.</p>
21     */
22    private HashMap<String,String> values;
23
24    /**
25     * Class constructor
26     * @param name Name of the {@code NodeAttribute}
27     * @param type Type of the {@code NodeAttribute}
28     */
29    public NodeAttribute(String name, String type)
30    {
31       super(name,type);
32       this.values = new HashMap<String,String>();
33    }
34
35    /**
36     * Adds a value to the {@code NodeAttribute}
37     * @param name Name of the attribute to set
38     * @param value Value to be set
39     */
40    @Override
41    public void setValue(String name, String value)
42    {
43       values.put(name, value);
44    }
45
46    /**
47     * Serializes the {@link #values} {@code HashMap} into a string to be
           stored by the Orion Context Broker as the attribute's
48     * value
```

```
49      * @return Serialized string containing the keys and values of the {@code
            Map}
50      */
51     @Override
52     public String getValue()
53     {
54        String str = "";
55        Iterator<Entry<String, String>> it = this.values.entrySet().iterator();
56        while(it.hasNext())
57        {
58           Map.Entry<String,String> pair = it.next();
59           str += pair.getKey();
60           str += "~"; //A more normal choice would be '=', but it turns out it'
                 s an illegal character in the latest versions of Orion
61           str += pair.getValue();
62           if(it.hasNext())
63           {
64              str += "|";
65           }
66        }
67        return str;
68     }
69
70     /**
71      * Parses a serialized string containing a set of values
72      *
73      * <p>It parses a string with the format {@code "key1~value1|key2~value2
             |[...]|keyN~valueN"} and it adds each key and its
74      * value to the {@link #values} {@code HashMap}.</p>
75      * @param valueString The string to be parsed
76      * @return {@code Map} containing the keys parsed with their respective
            values
77      * @throws ParseException If the string is malformed
78      */
79     public HashMap<String,String> parseValues(String valueString) throws
           ParseException
80     {
81        HashMap<String,String> localValues = new HashMap<String,String>();
82        int pos = 0;
83        String[] values = valueString.split("\\|");
84        for(int i = 0; i < values.length; i++)
85        {
86           pos += values[i].length();
87           String[] entry = values[i].split("~");
88           if(entry.length != 2)
89           {
90              ErrorPrinter.printLog("Attribute","parseValues",ErrorPrinter.
                    SEVERITY_ERROR, "Expected '~' in " + pos);
91              throw new ParseException("",0);
92           }
93           localValues.put(entry[0], entry[1]);
94        }
95        return localValues;
96     }
```

```
97 }
```

RegularAttribute.java

```java
package ch.hevs.ISyPeM2.ContextBroker;

/**
 * This is class represents a single-valued attribute of the Orion context broker
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class RegularAttribute extends Attribute
{
   /**
    * Value of the {@code RegularAttribute}
    */
   private String value;

   /**
    * Class constructor
    * @param name Name of the {@code RegularAttribute}
    * @param type Type of the {@code RegularAttribute}
    */
   public RegularAttribute(String name, String type)
   {
      super(name,type);
   }

   /**
    * Sets the value of the {@code RegularAttribute}
    * @param name Name of the value to be set. As {@code RegularAttribute} is single-valued, this parameter is ignored
    * @param value Value to be set
    */
   @Override
   public void setValue(String name, String value)
   {
      this.value = value;

   }

   /**
    * Retrieves the value of the {@code RegularAttribute}
    * @return the value this {@code RegularAttribute} has
    */
   @Override
   public String getValue()
   {
      return value;
   }

}
```

Subscription.java

```java
package ch.hevs.ISyPeM2.ContextBroker;

import java.util.HashMap;

import ch.hevs.ISyPeM2.Core.ErrorPrinter;
import ch.hevs.ISyPeM2.Node.Node;

/**
 * This class represents a subscription to the Orion context broker
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class Subscription
{
   /**
    * Subscription ID from the Orion context broker
    */
   private String ID;

   /**
    * {@code Map} which contains the nodes which we are subscribed to, using
       their names as key
    */
   private HashMap<String,Node> nodes;

   /**
    * {@code Map} which contains the number of cases each node has for this
       case collection we are subscribed to. It
    * uses the nodes's names as key
    */
   private HashMap<String,Integer> nodeCases;

   /**
    * {@code Map} which contains the timestamps of the last modification dates
        for the case collections from
    * the nodes we are subscribed to. It uses the nodes's names as key
    */
   private HashMap<String,String> nodeTimestamps;

   /**
    * {@link Element} object that represents the context element we are
       subscribed to
    */
   private Element contextElement;

   /**
    * Description of the case collection provided by the subscribing nodes
    */
   private String description;

   /**
    * Response URL sent to the Orion context broker to handle notifications
    */
```

```
50     private String responseURL;
51
52     /**
53      * Class constructor
54      * @param ID Subscription ID provided by the Orion context broker
55      * @param contextElement Context element to which the subscription is made
56      * @param responseURL URL where the node is running its {@link ch.hevs.
            ISyPeM2.NotificationServer.NotificationServer}
57      */
58     public Subscription(String ID, Element contextElement, String responseURL)
59     {
60        this.contextElement = contextElement;
61        this.ID = ID;
62        this.responseURL = responseURL;
63        nodes = new HashMap<String,Node>();
64        nodeCases = new HashMap<String,Integer>();
65        nodeTimestamps = new HashMap<String,String>();
66     }
67
68     /**
69      * This method checks whether or not the node subscription parameters
            passed by argument correspond to a node in this
70      * subscription and if so, checks if the node has made changes to its
            subscription
71      *
72      * @param node {@code Node} object containing the node's name and address.
73      * @param numCases Number of cases from the case collection regarding the
            subscription
74      * @param timestamp Timestamp with the last modification date of that node'
            s case collection
75      * @return {@code true} if the node is within the subscription and has
            changed any of the parameters of its subscription, {@code false}
            otherwise.
76      */
77     public boolean isModifiedNode(Node node, int numCases, String timestamp)
78     {
79        if(!(nodeCases.containsKey(node.getName()) && nodeTimestamps.containsKey
              (node.getName())))
80        {
81           ErrorPrinter.printLog("Subscription", "isModifiedNode", ErrorPrinter.
                 SEVERITY_WARNING, "The node being processed is NOT registered");
82           return false;
83        }
84        boolean isEqual = (numCases == nodeCases.get(node.getName()).intValue())
               && (timestamp.equals(nodeTimestamps.get(node.getName()))) && node.
              getAddress().equals(nodes.get(node.getName()).getAddress());
85        if(isEqual)
86        {
87           return false;
88        }
89        return true;
90     }
91
92     /**
```

```
93      * Updates the information about a node in this subscription with newer one
            from an incoming notification
94      * @param nodeName Name of the node whose information is going to be
            changed
95      * @param numCases New number of cases reported by the notification
96      * @param timestamp New timestamp reported by the notification
97      */
98     public void updateNode(String nodeName,int numCases, String timestamp)
99     {
100       this.nodeCases.remove(nodeName);
101       this.nodeTimestamps.remove(nodeName);
102       this.nodeCases.put(nodeName, new Integer(numCases));
103       this.nodeTimestamps.put(nodeName, timestamp);
104    }
105
106    /**
107     * This method returns the number of nodes we are subscribed to.
108     * @return Number of subscribed nodes
109     */
110    public int getNumNodes()
111    {
112       return nodes.size();
113    }
114
115    /**
116     * {@link #contextElement} getter
117     * @return The {@code Element} object representing the context element of
            this subscription
118     */
119    public Element getContextElement()
120    {
121       return contextElement;
122    }
123
124    /**
125     * {@link #ID} getter
126     * @return This subscription's ID, provided by the Orion context broker
127     */
128    public String getID()
129    {
130       return ID;
131    }
132
133    /**
134     * {@link #description} getter
135     * @return This subscription's description, provided by the subscribed
            nodes
136     */
137    public String getDescription()
138    {
139       return description;
140    }
141
142    /**
```

```
143      * {@link #responseURL} getter
144      * @return The response URL to handle notifications for this subscription
145      */
146     public String getResponseURL()
147     {
148        return responseURL;
149     }
150
151     /**
152      * {@link #description} setter
153      * @param description The new description for the subscribed case
             collection
154      */
155     public void setDescription(String description)
156     {
157        this.description = description;
158     }
159
160     /**
161      * Adds a {@code Node} to the list of subscribed nodes. <p>This method
             should NOT be used to add nodes dinamically
162      * as that is not supported. Nodes should only be added when the
             subscription is made. If any nodes get added or removed
163      * the subscription should be remade. This is already implemented in the {
             @link Node#notifyNode(String, Subscription, ContextBroker)} method from
              the {@link Node}
164      * class
165      *
166      * @param node {@code Node} object to add to the list of subscribed nodes
167      * @param numCases Number of cases the new node is offering
168      * @param timestamp Last modification date of the new node's case
             collection
169      */
170     public void addNode(Node node, int numCases, String timestamp)
171     {
172        nodes.put(node.getName(),node);
173        nodeCases.put(node.getName(), new Integer(numCases));
174        nodeTimestamps.put(node.getName(), timestamp);
175     }
176
177     /**
178      * Retrieves the number of cases a certain node is offering
179      * @param nodeName Name of the node which we want to inquiry the number of
             cases it offers
180      * @return The number of cases that node offers
181      */
182     public int getNumCases(String nodeName)
183     {
184        return nodeCases.get(nodeName).intValue();
185     }
186
187     /**
188      * Retrieves the last modification date timestamp of a certain node's case
             collection
```

```
189      * @param nodeName Name of the node which owns the case collection whose
             timestamp we're asking
190      * @return The timestamp of the last modification date of that node's case
             collection
191      */
192     public String getTimestamp(String nodeName)
193     {
194        return nodeTimestamps.get(nodeName);
195     }
196
197     /**
198      * Retrieves the list of subscribed nodes
199      * @return {@code Map} with the list of subscribed nodes, using their name
             as key
200      */
201     public HashMap<String,Node> getNodes()
202     {
203        return nodes;
204     }
205 }
```

D.2 ch.hevs.ISyPeM2.Core package

ErrorPrinter.java

```
1  package ch.hevs.ISyPeM2.Core;
2
3  import java.util.Calendar;
4
5  /**
6   * This class provides the means to log messages to the console according to a
        specified verbosity level
7   * @author Imanol-Mikel Barba Sabariego
8   *
9   */
10 public class ErrorPrinter
11 {
12    /**
13     * Current verbosity level. This variable is usually modified from the main
          class when certain commandline arguments are supplied
14     */
15    private static int verbosityLevel = 1;
16
17    public static final int SEVERITY_ERROR = 0;
18    public static final int SEVERITY_WARNING = 1;
19    public static final int SEVERITY_INFO = 2;
20    public static final int SEVERITY_DEBUG = 3;
21
22    /**
23     * {@link #verbosityLevel} setter
24     * @param v New verbosity level
25     */
26    public static void setVerbosity(int v)
27    {
28       verbosityLevel = v;
29    }
30
31    /**
32     * Prints a log message to stderr
33     * @param classFilename Name of the Java {@code Class} that calls this
          method
34     * @param methodName Name of the {@code Method} from which this method is
          called
35     * @param messageLevel Verbosity level of the log message
36     * @param message Log message to be printed
37     */
38    public static void printLog(String classFilename, String methodName, int
          messageLevel, String message)
39    {
40       if(messageLevel > verbosityLevel)
41       {
42          return;
43       }
44       String errorMsg = new java.sql.Timestamp(Calendar.getInstance().getTime
             ().getTime()).toString() + " [" + classFilename + ".java: " +
```

```
            methodName + "()] ";
45        switch(messageLevel)
46        {
47           case SEVERITY_DEBUG:
48              errorMsg += "DEBUG";
49              break;
50           case SEVERITY_INFO:
51              errorMsg += "INFO";
52              break;
53           case SEVERITY_WARNING:
54              errorMsg += "WARNING";
55              break;
56           case SEVERITY_ERROR:
57              errorMsg += "ERROR";
58              break;
59           default:
60              errorMsg += "UNKNOWN";
61              break;
62        }
63
64        errorMsg += ": " + message;
65        System.err.println(errorMsg);
66     }
67
68
69
70
71  }
```

MainClass.java

```java
package ch.hevs.ISyPeM2.Core;
import org.kohsuke.args4j.CmdLineException;
import org.kohsuke.args4j.CmdLineParser;
import org.kohsuke.args4j.Option;

/**
 * This class is a runnable class that serves as a template for a main class
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class MainClass
{
   @Option(name = "-v", aliases = { "--verbose" }, required = false, usage = "
      Increase verbosity to show INFO messages")
   private boolean verbose = false;

   @Option(name = "-vv", aliases = { "--verbose-debug" }, required = false,
      usage = "Increase verbosity to show DEBUG messages")
   private boolean debug = false;

   public static void main(String[] args)
   {
      MainClass arguments = new MainClass();
      CmdLineParser parser = new CmdLineParser(arguments);
      try
      {
         parser.parseArgument(args);
      }
      catch(CmdLineException exc)
      {
         parser.setUsageWidth(Integer.MAX_VALUE);
         parser.printUsage(System.err);
         return;
      }

      if(arguments.verbose)
      {
         ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_INFO);
         System.setProperty("org.eclipse.jetty.LEVEL","INFO");
      }
      if(arguments.debug)
      {
         ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_DEBUG);
         System.setProperty("org.eclipse.jetty.LEVEL","DEBUG");
      }

      final long startTime = System.currentTimeMillis();

      try
      {

         //DO STUFF HERE...

```

```
52          }
53          catch(Exception e)
54          {
55             ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                   SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
56          }
57
58          final long duration = System.currentTimeMillis() - startTime;
59          long seconds = duration/1000;
60          long ms = duration - seconds*1000;
61          System.out.println("Execution time: " + seconds + "s " + ms + "ms ");
62       }
63    }
```

D.3 ch.hevs.ISyPeM2.DB package

CaseProvisioner.java

```
1  package ch.hevs.ISyPeM2.DB;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  import ch.hevs.ISyPeM2.Node.CaseAggregation;
7  import ch.hevs.ISyPeM2.Node.CaseCollection;
8
9  /**
10  * This interfaces represents a storage backend to store/retrieve case
       collections and aggregations from some type of
11  * database. {@link ch.hevs.ISyPeM2.Node.Node} objects use provisioners
       following this interface to manage case storage.
12  *
13  * @author Imanol-Mikel Barba Sabariego
14  *
15  */
16 public interface CaseProvisioner
17 {
18    /**
19     * Writes a {@link CaseAggregation} object into storage
20     *
21     * @param aggregation {@link CaseAggregation} object to be saved
22     * @throws Exception If the operation is unsuccessful
23     */
24    public void writeCaseAggregation(CaseAggregation aggregation) throws
          Exception;
25
26    /**
27     * Reads a case aggregation from storage and returns a {@link
          CaseAggregation} object holding a copy of said data.
28     *
29     * <p>Please note that modifying the {@link CaseAggregation} object
          returned will not have any effect on
30     * the data stored.</p>
31     *
32     * @param name Name of the case aggregation to fetch
33     * @return {@link CaseAggregation} object holding a copy of the actual data
            stored, if the aggregation doesn't exist
34     * it will return {@code null}
35     * @throws Exception If the operation is unsuccessful
36     */
37    public CaseAggregation readCaseAggregation(String name) throws Exception;
38
39    /**
40     * Deletes a case aggregation from storage
41     * @param name Name of the case aggregation to remove
42     * @throws Exception If the operation is unsuccessful
43     */
44    public void removeCaseAggregation(String name) throws Exception;
```

```
45
46      /**
47       * Writes a {@link CaseCollection} object into a case aggregation already
             into storage
48       *
49       * @param collection {@link CaseCollection} object to be saved
50       * @throws Exception If the operation is unsuccessful
51       */
52      public void writeCaseCollection(CaseCollection collection) throws Exception
             ;
53
54      /**
55       * Reads a case collection from a case aggregation in storage and returns a
             {@link CaseCollection} object
56       * holding a copy of said data.
57       *
58       * <p>Please note that modifying the {@link CaseCollection} object returned
             will not have any effect on
59       * the data stored.</p>
60       *
61       * @param ID ID of the case collection to fetch
62       * @return {@link CaseCollection} object holding a copy of the actual data
             stored, if the collection doesn't exist
63       * it will return {@code null}
64       * @throws Exception If the operation is unsuccessful
65       */
66      public CaseCollection readCaseCollection(String ID) throws Exception;
67
68      /**
69       * Deletes a case collection from storage
70       * @param collectionID ID of the case collection to remove
71       * @throws Exception If the operation is unsuccessful
72       */
73      public void removeCaseCollection(String collectionID) throws Exception;
74
75      /**
76       * Queries the storage media for case collections matching a certain
             criteria and returns a {@code Set} of matching collections
77       *
78       * @param caseType Name of the aggregation the case collections are part of
79       * @param keywords Keywords that have to be present in the case collections
             being searched
80       * @param attributes Attributes that have to be present in the case
             collections being searched
81       * @return {@code Set} of case collections matching the criteria in the
             parameters
82       * @throws Exception If the operation is unsuccessful
83       */
84      public HashSet<CaseCollection> queryDatabase(String caseType, HashSet<
             String> keywords, Set<String> attributes) throws Exception;
85  }
```

LocalCaseProvisioner.java

```
1  package ch.hevs.ISyPeM2.DB;
2
3  import java.util.HashMap;
4  import java.util.HashSet;
5  import java.util.Iterator;
6  import java.util.Set;
7
8  import ch.hevs.ISyPeM2.Core.ErrorPrinter;
9  import ch.hevs.ISyPeM2.Node.Case;
10 import ch.hevs.ISyPeM2.Node.CaseAggregation;
11 import ch.hevs.ISyPeM2.Node.CaseAttribute;
12 import ch.hevs.ISyPeM2.Node.CaseCollection;
13
14 /**
15  * This case provisioner stores and reads the collections and aggregations
       directly into memory as Java objects.
16  *
17  * <p>When storing or fetching an object, it will first make a copy of it so
       further modifications to the reference
18  * being stored or read from storage will not have any effect on the data
       stored.</p>
19  *
20  * @author Imanol-Mikel Barba Sabariego
21  *
22  */
23 public class LocalCaseProvisioner implements CaseProvisioner
24 {
25    /**
26     * {@code Map} holding the {@link CaseAggregation} objects the node holds
27     *
28     * <p>The case aggregation's name is used as key for faster lookup and
          retrieval of a certain case aggregation</p>
29     */
30    private HashMap <String,CaseAggregation> caseAggregations;
31
32    /**
33     * Class constructor
34     */
35    public LocalCaseProvisioner()
36    {
37       this.caseAggregations = new HashMap<String,CaseAggregation>();
38    }
39
40    @Override
41    public void writeCaseAggregation(CaseAggregation aggregation) throws
          Exception
42    {
43       CaseAggregation newAggregation = new CaseAggregation(aggregation.getName
             (),aggregation.getDescription());
44       newAggregation.setTimestamp(aggregation.getTimestamp());
45       this.caseAggregations.put(newAggregation.getName(), newAggregation);
46       Iterator<CaseCollection> itCol = aggregation.getCollections();
47       while(itCol.hasNext())
```

```
48          {
49              this.writeCaseCollection(itCol.next());
50          }
51      }
52
53      @Override
54      public CaseAggregation readCaseAggregation(String name)
55      {
56          CaseAggregation aggregation = this.caseAggregations.get(name);
57          CaseAggregation newAggregation = null;
58          if(aggregation != null)
59          {
60              newAggregation = new CaseAggregation(aggregation.getName(),
                    aggregation.getDescription());
61              newAggregation.setTimestamp(aggregation.getTimestamp());
62              Iterator<CaseCollection> itCol = aggregation.getCollections();
63              while(itCol.hasNext())
64              {
65                  try
66                  {
67                      /* Imanol Barba
68                       *
69                       * Okay, now this seems stupid. I know that with the iterator
                            we ALREADY have access to the
70                       * CaseCollection object BUT using the readCaseCollection()
                            method gives us a COPY of the object,
71                       * not a REFERENCE, and since just getting object references
                            from a Map is a tireless effort, we
72                       * may as well pay the performance price to get our VERY OWN
                            COPY which we may freely screw up as
73                       * much as it is desired.
74                       */
75                      newAggregation.addCollection(this.readCaseCollection(itCol.next
                            ().getID()));
76                  }
77                  catch (Exception e)
78                  {
79                      //This shouldn't happen AT ALL, if it does I'll punch a kitten
                            and kick a puppy
80                      ErrorPrinter.printLog("LocalCaseProvisioner", "
                            readCaseAggregation", ErrorPrinter.SEVERITY_ERROR, e.
                            getClass() + ": " + e.getMessage());
81                  }
82              }
83          }
84          return newAggregation;
85      }
86
87      @Override
88      public void removeCaseAggregation(String name) throws Exception
89      {
90          if(this.caseAggregations.containsKey(name))
91          {
92              this.caseAggregations.remove(name);
```

```
93          }
94          else
95          {
96             throw new Exception ("Aggregation " + name + " not found");
97          }
98       }
99
100      //TODO: if RSDB
101      @Override
102      public void removeCaseCollection(String collectionID) throws Exception
103      {
104         Iterator<CaseAggregation> itAggr = this.caseAggregations.values().
               iterator();
105         while(itAggr.hasNext())
106         {
107            CaseAggregation aggregation = itAggr.next();
108            aggregation.startAtomicOperation();
109            if(aggregation.hasCollection(collectionID))
110            {
111               aggregation.removeCollection(collectionID);
112               aggregation.stopAtomicOperation();
113               return;
114            }
115            aggregation.stopAtomicOperation();
116         }
117         throw new Exception("Collection " + collectionID + " not found");
118      }
119
120      //TODO: if RSDB
121      @Override
122      public void writeCaseCollection(CaseCollection collection) throws Exception
123      {
124         if(this.caseAggregations.containsKey(collection.getType()))
125         {
126            CaseCollection newCollection = new CaseCollection(collection.getID(),
                  collection.getType(),collection.getDescription(),collection.
                  getOriginNode());
127            Iterator<String> itKw = collection.getKeywords();
128            while(itKw.hasNext())
129            {
130               newCollection.addKeyword(itKw.next());
131            }
132            Iterator<CaseAttribute> itAttr = collection.getAttributes();
133            while(itAttr.hasNext())
134            {
135               CaseAttribute attr = itAttr.next();
136               newCollection.setAttribute(CaseAttribute.parseCaseAttribute(attr.
                     getName(), attr.getValue()));
137            }
138            Iterator<Case> itCase = collection.getCases();
139            while(itCase.hasNext())
140            {
141               Case c = itCase.next();
142               Case newCase = new Case(c.getPS(),c.getType(),c.getHealthData());
```

```
143             newCase.setID(c.getID());
144             Iterator<CaseAttribute> itCaseAttr = c.getAttributes();
145             while(itCaseAttr.hasNext())
146             {
147                CaseAttribute attr = itAttr.next();
148                newCase.addAttribute(CaseAttribute.parseCaseAttribute(attr.
                      getName(), attr.getValue()));
149             }
150             newCollection.addCase(newCase);
151          }
152          this.caseAggregations.get(collection.getType()).addCollection(
                newCollection);
153       }
154       else
155       {
156          throw new Exception ("There is no case aggregation for that
                collection");
157       }
158    }
159
160    //TODO: if RSDB
161    @Override
162    public CaseCollection readCaseCollection(String ID)
163    {
164       CaseCollection newCollection = null;
165       Iterator<CaseAggregation> itAggr = this.caseAggregations.values().
             iterator();
166       while(itAggr.hasNext())
167       {
168          CaseAggregation aggregation = itAggr.next();
169          aggregation.startAtomicOperation();
170          if(aggregation.hasCollection(ID))
171          {
172             CaseCollection collection = aggregation.getCollection(ID);
173             newCollection = new CaseCollection(collection.getID(),collection.
                   getType(),collection.getDescription(),collection.getOriginNode
                   ());
174             Iterator<String> itKw = collection.getKeywords();
175             while(itKw.hasNext())
176             {
177                newCollection.addKeyword(itKw.next());
178             }
179             Iterator<Case> itCase = collection.getCases();
180             while(itCase.hasNext())
181             {
182                Case c = itCase.next();
183                Case newCase = new Case(c.getPS(),c.getType(),c.getHealthData()
                      );
184                newCase.setID(c.getID());
185                Iterator<CaseAttribute> itCaseAttr = c.getAttributes();
186                while(itCaseAttr.hasNext())
187                {
188                   CaseAttribute attr = itCaseAttr.next();
```

```
189                     newCase.setAttribute(CaseAttribute.parseCaseAttribute(attr.
                            getName(), attr.getValue()));
190                 }
191                 try
192                 {
193                     newCollection.addCase(newCase);
194                 }
195                 catch (Exception e)
196                 {
197                     ErrorPrinter.printLog("LocalCaseProvisioner", "
                            readCaseCollection", ErrorPrinter.SEVERITY_ERROR, "Error
                            while adding case: attribute mismatch");
198                 }
199             }
200         }
201         aggregation.stopAtomicOperation();
202     }
203     return newCollection;
204 }
205
206 @Override
207 public HashSet<CaseCollection> queryDatabase(String caseType, HashSet<
        String> keywords, Set<String> attributes)
208 {
209     HashSet<CaseCollection> collections = new HashSet<CaseCollection>();
210     Iterator<CaseCollection> itCol = this.readCaseAggregation(caseType).
            getCollections();
211     while(itCol.hasNext())
212     {
213         boolean matchesKeywords = true;
214         boolean matchesAttributes = true;
215         CaseCollection collection = itCol.next();
216         if(attributes.size() == 0)
217         {
218             /* Imanol Barba
219              *
220              * We don't allow requests with no attributes
221              */
222             return collections;
223         }
224         if((keywords.size() > collection.getNumKeywords()) || (attributes.
                size() > collection.getNumAttributes()))
225         {
226             /* Imanol Barba
227              *
228              * We are asking more attributes/keywords than the collection
                    actually has, so there's
229              * no point in searching through the current collection
230              */
231             continue;
232         }
233         Iterator<String> itKw = keywords.iterator();
234         while(itKw.hasNext())
235         {
```

```
236                 if(!collection.hasKeyword(itKw.next()))
237                 {
238                     matchesKeywords = false;
239                 }
240             }
241             if(!matchesKeywords)
242             {
243                 continue;
244             }
245             Iterator<String> itAttr = attributes.iterator();
246             while(itAttr.hasNext())
247             {
248                 if(!collection.hasAttribute(itAttr.next()))
249                 {
250                     matchesAttributes = false;
251                 }
252             }
253             if(!matchesAttributes)
254             {
255                 continue;
256             }
257             collections.add(collection);
258         }
259         return collections;
260     }
261 }
```

MySQLCaseProvisioner.java

```
1  package ch.hevs.ISyPeM2.DB;
2
3  import java.sql.DriverManager;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  /**
8   * This class extends the {@link SQLCaseProvisioner} to interact with a MySQL
        database
9   *
10  * @author Imanol-Mikel Barba Sabariego
11  *
12  */
13 public class MySQLCaseProvisioner extends SQLCaseProvisioner
14 {
15    /**
16     * Class constructor, it will initialize the JDBC driver and make a
          connection with the MySQL database
17     *
18     * @param host Hostname where the MySQL instance is running
19     * @param user Username with permission to CREATE, SELECT, INSERT, REPLACE,
           DELETE, DROP in the database
20     * @param pass Password of the username
21     * @param DBName Name of the MySQL database
22     * @throws SQLException If connection to the SQL database fails
23     * @throws ClassNotFoundException If the JDBC MySQL connector is not found
24     */
25    public MySQLCaseProvisioner(String host, String user, String pass, String
          DBName) throws SQLException, ClassNotFoundException
26    {
27       Class.forName("com.mysql.jdbc.Driver");
28       this.connection = DriverManager.getConnection("jdbc:mysql://" + host + "
             /" + DBName, user, pass);
29       this.createDatabase(DBName);
30    }
31
32    /**
33     * Creates the initial schema for the MySQL database if it doesn't exist or
           it's empty
34     *
35     * @param DBName Name of the database to create
36     * @throws SQLException if the operation fails
37     */
38    public void createDatabase(String DBName) throws SQLException
39    {
40       String createStatement = "CREATE DATABASE IF NOT EXISTS `" + DBName + "`
             ";
41       Statement SQLStatement = connection.createStatement();
42       SQLStatement.executeUpdate(createStatement);
43       SQLStatement.close();
44
45       createStatement = "CREATE TABLE IF NOT EXISTS `Aggregation_index` (`name
             ` varchar(45) NOT NULL,`description` "
```

```
46              + "mediumtext,`timestamp` varchar(45) DEFAULT NULL,PRIMARY KEY (`
                    name`)) ENGINE=InnoDB DEFAULT "
47              + "CHARSET=utf8";
48          SQLStatement = connection.createStatement();
49          SQLStatement.executeUpdate(createStatement);
50          SQLStatement.close();
51
52          createStatement = "CREATE TABLE IF NOT EXISTS `Collection_index` (`
                DBName` varchar(64) NOT NULL, `KW` tinytext, "
53              + "`type` varchar(45) DEFAULT NULL, `description` varchar(45)
                    DEFAULT NULL, `origin` varchar(45) DEFAULT "
54              + "NULL, PRIMARY KEY (`DBName`), KEY `aggr_foreign_idx` (`type`),
                    CONSTRAINT `aggr_foreign` FOREIGN KEY "
55              + "(`type`) REFERENCES `Aggregation_index` (`name`) ON DELETE NO
                    ACTION ON UPDATE NO ACTION) ENGINE=InnoDB "
56              + "DEFAULT CHARSET=utf8";
57          SQLStatement = connection.createStatement();
58          SQLStatement.executeUpdate(createStatement);
59          SQLStatement.close();
60
61          createStatement = "CREATE TABLE IF NOT EXISTS `Collection_attributes` (`
                DBName` varchar(64) NOT NULL, `name` "
62              + "varchar(45) DEFAULT NULL,`value` varchar(45) DEFAULT NULL, KEY
                    `index1` (`DBName`), CONSTRAINT "
63              + "`collection_name` FOREIGN KEY (`DBName`) REFERENCES `
                    Collection_index` (`DBName`) ON DELETE NO ACTION "
64              + "ON UPDATE NO ACTION) ENGINE=InnoDB DEFAULT CHARSET=utf8";
65          SQLStatement = connection.createStatement();
66          SQLStatement.executeUpdate(createStatement);
67          SQLStatement.close();
68      }
69
70      public void startTransaction() throws SQLException
71      {
72          Statement SQLStatement = connection.createStatement();
73          SQLStatement.executeUpdate("START TRANSACTION");
74          SQLStatement.close();
75      }
76
77      public void endTransaction() throws SQLException
78      {
79          Statement SQLStatement = connection.createStatement();
80          SQLStatement.executeUpdate("COMMIT");
81          SQLStatement.close();
82      }
83  }
```

SQLCaseProvisioner.java

```
1  package ch.hevs.ISyPeM2.DB;
2
3  import java.sql.Connection;
4  import java.sql.DatabaseMetaData;
5  import java.sql.ResultSet;
6  import java.sql.ResultSetMetaData;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9  import java.util.HashSet;
10 import java.util.Iterator;
11 import java.util.Set;
12
13 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
14 import ch.hevs.ISyPeM2.Node.Case;
15 import ch.hevs.ISyPeM2.Node.CaseAggregation;
16 import ch.hevs.ISyPeM2.Node.CaseAttribute;
17 import ch.hevs.ISyPeM2.Node.CaseCollection;
18 import ch.hevs.ISyPeM2.Node.Node;
19
20 /**
21  * This class provides case provisioning using a generic SQL database
22  *
23  * <p>This class is abstract because it's meant to be subclassed with each
       specific SQL database because they use
24  * different JDBC connectors, credentials and such. The methods here
       implemented can be overriden in the subclasses
25  * for more efficient/specific implementations for different SQL databases</p>
26  *
27  * @author Imanol-Mikel Barba Sabariego
28  *
29  */
30 public abstract class SQLCaseProvisioner implements CaseProvisioner
31 {
32    /**
33     * {@link java.sql.Connection} object that holds the connection with the
          remote SQL database
34     */
35    protected Connection connection;
36
37    /**
38     * Starts a SQL Transaction
39     * @throws SQLException if the operation fails
40     */
41    public abstract void startTransaction() throws SQLException;
42
43    /**
44     * Ends a SQL Transaction
45     * @throws SQLException if the operation fails
46     */
47    public abstract void endTransaction() throws SQLException;
48
49    public void writeCaseAggregation(CaseAggregation aggregation) throws
          Exception
```

```
50      {
51          String checkStatement = "SELECT COUNT(*) FROM Aggregation_index WHERE
                name = '" + aggregation.getName() + "'";
52          Statement SQLStatement = connection.createStatement();
53          ResultSet result = SQLStatement.executeQuery(checkStatement);
54          result.next();
55          if(result.getInt(1) != 0)
56          {
57              this.removeCaseAggregation(aggregation.getName());
58          }
59          SQLStatement.close();
60          String insertStatement = "REPLACE INTO Aggregation_index VALUES ('" +
                aggregation.getName() + "','" + aggregation.getDescription() + "','"
                + aggregation.getTimestamp() + "')";
61          SQLStatement = connection.createStatement();
62          SQLStatement.executeUpdate(insertStatement);
63          SQLStatement.close();
64          Iterator<CaseCollection> itCol = aggregation.getCollections();
65          while(itCol.hasNext())
66          {
67              this.writeCaseCollection(itCol.next());
68          }
69      }
70
71      public CaseAggregation readCaseAggregation(String name) throws SQLException
72      {
73          String queryStatement = "SELECT * FROM Aggregation_index WHERE name = '"
                + name + "'";
74          Statement SQLStatement = connection.createStatement();
75          ResultSet result = SQLStatement.executeQuery(queryStatement);
76          if(!result.next())
77          {
78              return null;
79          }
80          CaseAggregation aggregation = new CaseAggregation(result.getString(1),
                result.getString(2));
81          aggregation.setTimestamp(result.getString(3));
82          SQLStatement.close();
83          queryStatement = "SELECT * FROM Collection_index WHERE type = '" +
                aggregation.getName() + "'";
84          SQLStatement = connection.createStatement();
85          result = SQLStatement.executeQuery(queryStatement);
86          HashSet<String> collections = new HashSet<String>();
87          while(result.next())
88          {
89              collections.add(result.getString(1));
90          }
91          SQLStatement.close();
92          Iterator<String> itCol = collections.iterator();
93          while(itCol.hasNext())
94          {
95              CaseCollection collection = this.readCaseCollection(itCol.next());
96              if(collection != null)
97              {
```

```
98              aggregation.addCollection(collection);
99           }
100       }
101       return aggregation;
102    }
103
104    public void removeCaseAggregation(String name) throws Exception
105    {
106       String checkStatement = "SELECT COUNT(*) FROM Aggregation_index WHERE
              name = '" + name + "'";
107       Statement SQLStatement = connection.createStatement();
108       ResultSet result = SQLStatement.executeQuery(checkStatement);
109       result.next();
110       if(result.getInt(1) == 0)
111       {
112          throw new Exception("Aggregation " + name + " not found");
113       }
114       SQLStatement.close();
115       String selectStatement = "SELECT DBName FROM Collection_index WHERE type
              = '" + name + "'";
116       SQLStatement = connection.createStatement();
117       result = SQLStatement.executeQuery(selectStatement);
118       HashSet<String> collections = new HashSet<String>();
119       while(result.next())
120       {
121          collections.add(result.getString(1));
122       }
123       SQLStatement.close();
124       Iterator<String> itCol = collections.iterator();
125       while(itCol.hasNext())
126       {
127          this.removeCaseCollection(itCol.next());
128       }
129
130       String removeStatement = "DELETE FROM Aggregation_index WHERE name = '"
              + name + "'";
131       SQLStatement = connection.createStatement();
132       SQLStatement.executeUpdate(removeStatement);
133       SQLStatement.close();
134    }
135
136    /**
137     * Creates the table for a case collection in the SQL database
138     *
139     * @param collection {@link CaseCollection} which will be stored into the
              database
140     * @throws SQLException If the operation is unsuccessful
141     */
142    private void createCaseCollection(CaseCollection collection) throws
           SQLException
143    {
144       String createStatement = "CREATE TABLE `" + collection.getID() + "` (`ID
              ` varchar(45) NOT NULL, `PS` varchar(45) NOT NULL, ";
145       Iterator<CaseAttribute> itAttr = collection.getAttributes();
```

```
146         while(itAttr.hasNext())
147         {
148             String attrName = itAttr.next().getName();
149             createStatement += "`" + attrName + "` varchar(45) DEFAULT NULL, ";
150         }
151         createStatement += "`health_data` text, PRIMARY KEY (`ID`))";
152         Statement SQLStatement = connection.createStatement();
153         SQLStatement.executeUpdate(createStatement);
154         SQLStatement.close();
155     }
156 
157     //TODO: if RSDB
158     public void removeCaseCollection(String collectionID) throws Exception
159     {
160         String checkStatement = "SELECT COUNT(*) FROM Collection_index WHERE
                DBName = '" + collectionID + "'";
161         Statement SQLStatement = connection.createStatement();
162         ResultSet result = SQLStatement.executeQuery(checkStatement);
163         result.next();
164         if(result.getInt(1) == 0)
165         {
166             throw new Exception("Collection " + collectionID + " not found");
167         }
168         SQLStatement.close();
169 
170         String removeStatement = "DELETE FROM Collection_attributes WHERE DBName
                = '" + collectionID + "'";
171         SQLStatement = connection.createStatement();
172         SQLStatement.executeUpdate(removeStatement);
173         SQLStatement.close();
174 
175         removeStatement = "DELETE FROM Collection_index WHERE DBName = '" +
                collectionID + "'";
176         SQLStatement = connection.createStatement();
177         SQLStatement.executeUpdate(removeStatement);
178         SQLStatement.close();
179 
180         String dropStatement = "DROP TABLE IF EXISTS `" + collectionID + "`";
181         SQLStatement = connection.createStatement();
182         SQLStatement.executeUpdate(dropStatement);
183         SQLStatement.close();
184     }
185 
186     //TODO: if RSDB
187     public void writeCaseCollection(CaseCollection collection) throws
            SQLException
188     {
189         String keywords = "";
190         Iterator<String> itKey = collection.getKeywords();
191         while(itKey.hasNext())
192         {
193             keywords += itKey.next();
194             if(itKey.hasNext())
195             {
```

```
196             keywords += ",";
197          }
198       }
199       startTransaction();
200       String indexStatement = "REPLACE INTO Collection_index VALUES ('" +
             collection.getID() + "','" + keywords + "','"
201             + collection.getType() + "','" + collection.getDescription() + "
                ','" + collection.getOriginNode().getName() + "|" + collection.
                getOriginNode().getAddress() + "')";
202       Statement SQLStatement = connection.createStatement();
203       SQLStatement.executeUpdate(indexStatement);
204       SQLStatement.close();
205       String insertAttributesStatement = "REPLACE INTO Collection_attributes
             VALUES ";
206       Iterator<CaseAttribute> itAttr = collection.getAttributes();
207       while(itAttr.hasNext())
208       {
209          CaseAttribute attr = itAttr.next();
210          insertAttributesStatement += "('" + collection.getID() + "','" + attr
                .getName() + "','" + attr.getValue() + "')";
211          if(itAttr.hasNext())
212          {
213             insertAttributesStatement += ", ";
214          }
215       }
216       SQLStatement = connection.createStatement();
217       SQLStatement.executeUpdate(insertAttributesStatement);
218       SQLStatement.close();
219       DatabaseMetaData meta = connection.getMetaData();
220       ResultSet result = meta.getTables(null, null, collection.getID(), new
             String[] {"TABLE"});
221       if(!result.next())
222       {
223          this.createCaseCollection(collection);
224       }
225       result.close();
226       Iterator<Case> itCase = collection.getCases();
227       while(itCase.hasNext())
228       {
229          Case c = itCase.next();
230          String insertStatement = "REPLACE INTO " + collection.getID() + "
                VALUES ('" + c.getID() + "','" + c.getPS() + "',";
231          itAttr = collection.getAttributes();
232          while(itAttr.hasNext())
233          {
234             CaseAttribute caseAttr = c.getAttribute(itAttr.next().getName());
235             insertStatement += "'" + caseAttr.getValue() + "',";
236          }
237          insertStatement += "'" + c.getHealthData() + "')";
238
239          SQLStatement = connection.createStatement();
240          SQLStatement.executeUpdate(insertStatement);
241          SQLStatement.close();
242       }
```

```
243         endTransaction();
244     }
245 
246     //TODO: if RSDB
247     public CaseCollection readCaseCollection(String ID) throws SQLException
248     {
249         String queryStatement = "SELECT * FROM Collection_index WHERE DBName = '
                " + ID + "'";
250         Statement SQLStatement = connection.createStatement();
251         ResultSet result = SQLStatement.executeQuery(queryStatement);
252         if(!result.next())
253         {
254             return null;
255         }
256         String nodeData = result.getString(5);
257         Node n = new Node(nodeData.substring(0, nodeData.indexOf("|")),nodeData.
                substring(nodeData.indexOf("|")+1));
258         CaseCollection collection = new CaseCollection(ID,result.getString(3),
                result.getString(4),n);
259         SQLStatement.close();
260         String queryTable = "SELECT * FROM " + ID;
261         SQLStatement = connection.createStatement();
262         result = SQLStatement.executeQuery(queryTable);
263         ResultSetMetaData metadata = result.getMetaData();
264         while(result.next())
265         {
266             Case c = new Case(result.getString(2),collection.getType(),result.
                    getString(metadata.getColumnCount()));
267             c.setID(result.getString(1));
268             for(int i = 1; i < metadata.getColumnCount(); i++)
269             {
270                 String columnName = metadata.getColumnName(i);
271                 if(!(columnName.equals("PS") || columnName.equals("health_data")
                        || columnName.equalsIgnoreCase("ID")))
272                 {
273                     c.setAttribute(CaseAttribute.parseCaseAttribute(columnName,
                            result.getString(i)));
274                 }
275             }
276             try
277             {
278                 collection.addCase(c);
279             }
280             catch (Exception e)
281             {
282                 ErrorPrinter.printLog("SQLCaseProvisioner", "readCaseCollection",
                        ErrorPrinter.SEVERITY_ERROR, "Error while adding case:
                        attribute mismatch");
283             }
284         }
285         SQLStatement.close();
286         String queryIndex = "SELECT KW FROM Collection_index WHERE DBName = '" +
                collection.getID() + "'";
287         SQLStatement = connection.createStatement();
```

```
288         result = SQLStatement.executeQuery(queryIndex);
289         result.next();
290         String[] keywords = result.getString(1).split(",");
291         for(int i = 0; i < keywords.length; i++)
292         {
293            collection.addKeyword(keywords[i]);
294         }
295         SQLStatement.close();
296         return collection;
297      }
298
299      public HashSet<CaseCollection> queryDatabase(String caseType, HashSet<
            String> keywords, Set<String> attributes) throws SQLException
300      {
301         /* CASE 1: Multiple attributes and multiple keywords
302          *
303          * SELECT Q1.DBName FROM Collection_index Q1 INNER JOIN (SELECT DISTINCT
                T1.DBName AS DBName FROM
304          * Collection_attributes T1 INNER JOIN Collection_attributes T2 ON (T1.
               DBName = T2.DBName) WHERE
305          * T1.name = 'weight' AND T2.name = 'gender') AS Q2 ON Q1.DBName = Q2.
               DBName WHERE type = 'Malaria'
306          * AND (KW LIKE 'chronic,%' || KW LIKE '%,chronic' || KW LIKE '%,chronic
               ,%' || KW LIKE 'chronic')
307          * AND (KW LIKE 'adults,%' || KW LIKE '%,adults' || KW LIKE '%,adults,%'
                || KW LIKE 'adults');
308          */
309
310         /* CASE 2: Single attribute and single keyword
311          *
312          * SELECT Q1.DBName FROM Collection_index Q1 INNER JOIN (SELECT DISTINCT
                T1.DBName AS DBName FROM
313          * Collection_attributes T1 WHERE T1.name = 'age') AS Q2 ON Q1.DBName =
               Q2.DBName WHERE
314          * type = 'Malaria' AND (KW LIKE 'chronic,%' || KW LIKE '%,chronic' ||
               KW LIKE '%,chronic,%' || KW LIKE 'chronic')
315          */
316
317         HashSet<CaseCollection> collections = new HashSet<CaseCollection>();
318         String query = "SELECT Q1.DBName FROM Collection_index Q1 INNER JOIN (
               SELECT DISTINCT T1.DBName AS DBName "
319               + "FROM Collection_attributes T1";
320
321         if(attributes.size() == 0)
322         {
323            /* Imanol Barba
324             *
325             * We don't allow requests with no attributes
326             */
327            return collections;
328         }
329         else if(attributes.size() == 1)
330         {
331            query += " WHERE T1.name = '" + attributes.iterator().next() + "'";
```

```
332         }
333         else
334         {
335            String innerJoinString = "";
336            String onString = "";
337            String columnString = "";
338            Iterator<String> itAttr = attributes.iterator();
339            int i = 2;
340            while(itAttr.hasNext())
341            {
342               String num = Integer.toString(i++);
343               innerJoinString += " INNER JOIN Collection_attributes T" + num;
344               onString += "T" + num + ".DBName";
345               columnString += "T" + num + ".name = '" + itAttr.next() + "'";
346               if(itAttr.hasNext())
347               {
348                  onString += " AND T" + num + ".DBName = ";
349                  columnString += " AND ";
350               }
351            }
352            query += innerJoinString + " ON(T1.DBName = " + onString + ") WHERE "
                   + columnString;
353         }
354         query += ") AS Q2 ON Q1.DBName = Q2.DBName WHERE type = '" + caseType +
               "'";
355         Iterator<String> itKw = keywords.iterator();
356         while(itKw.hasNext())
357         {
358            String kw = itKw.next();
359            query += " AND (KW LIKE '" + kw + ",%' || KW LIKE '%," + kw + "' ||
                  KW LIKE '%," + kw + ",%' || KW LIKE '" + kw + "')";
360         }
361         Statement SQLStatement = connection.createStatement();
362         ResultSet result = SQLStatement.executeQuery(query);
363         while(result.next())
364         {
365            collections.add(this.readCaseCollection(result.getString(1)));
366         }
367         return collections;
368      }
369   }
```

SQLiteCaseProvisioner.java

```java
package ch.hevs.ISyPeM2.DB;

import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * This class extends the {@link SQLCaseProvisioner} to interact with a SQLite database
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class SQLiteCaseProvisioner extends SQLCaseProvisioner
{
   /**
    * Class constructor, it will initialize the JDBC driver and create or read the SQLite database
    *
    * @param file Path of the SQLite file
    * @throws SQLException If connection to the SQL database fails
    * @throws ClassNotFoundException If the JDBC MySQL connector is not found
    */
   public SQLiteCaseProvisioner(String file) throws SQLException, ClassNotFoundException
   {
      Class.forName("org.sqlite.JDBC");
      this.connection = DriverManager.getConnection("jdbc:sqlite:" + file);
      this.createDatabase();
   }

   /**
    * Creates the initial schema for the SQLite database if it's empty
    *
    * @throws SQLException if the operation fails
    */
   public void createDatabase() throws SQLException
   {
      String createStatement = "CREATE TABLE IF NOT EXISTS `Aggregation_index` (`name` varchar(45) NOT NULL,`description` "
            + "mediumtext,`timestamp` varchar(45) DEFAULT NULL,PRIMARY KEY (`name`))";
      Statement SQLStatement = connection.createStatement();
      SQLStatement.executeUpdate(createStatement);
      SQLStatement.close();

      createStatement = "CREATE TABLE IF NOT EXISTS `Collection_attributes` (`DBName` varchar(64) NOT NULL, `name` "
            + "varchar(45) DEFAULT NULL,`value` varchar(45) DEFAULT NULL)";
      SQLStatement = connection.createStatement();
      SQLStatement.executeUpdate(createStatement);
      SQLStatement.close();

```

```
48          createStatement = "CREATE TABLE IF NOT EXISTS `Collection_index` (`
               DBName` varchar(64) NOT NULL, `KW` tinytext, "
49                + "`type` varchar(45) DEFAULT NULL, `description` varchar(45)
                     DEFAULT NULL, `origin` varchar(45) DEFAULT "
50                + "NULL, PRIMARY KEY (`DBName`))";
51          SQLStatement = connection.createStatement();
52          SQLStatement.executeUpdate(createStatement);
53          SQLStatement.close();
54       }
55
56       public void startTransaction() throws SQLException
57       {
58          Statement SQLStatement = connection.createStatement();
59          SQLStatement.executeUpdate("BEGIN TRANSACTION");
60          SQLStatement.close();
61       }
62
63       public void endTransaction() throws SQLException
64       {
65          Statement SQLStatement = connection.createStatement();
66          SQLStatement.executeUpdate("END TRANSACTION");
67          SQLStatement.close();
68       }
69    }
```

D.4 ch.hevs.ISyPeM2.Generalization package

AttributeTree.java

```
1  package ch.hevs.ISyPeM2.Generalization;
2
3
4  import java.util.TreeMap;
5
6  import ch.hevs.ISyPeM2.Node.NumericalAttribute;
7  /**
8   * Class extends TreeMap and represents a generalization tree
9   * @author Alevtina
10  *
11  * @param <T> Type of the Vertice (Range or Category)
12  */
13
14 public class AttributeTree<T> extends TreeMap<String, Vertice<T>> {
15    private static final long serialVersionUID = 1L;
16    private final boolean isRange;
17  /**
18   * Class constructor
19   * @param isRange boolean parameter indicating whether the tree (for which
       this instance of AttributeTree is being created) is numerical or not (
       categorical)
20   */
21    public AttributeTree(boolean isRange) {
22       super();
23       this.isRange = isRange;
24 }
25  /**
26   * Method for getting binary representation of a non-generalized value (always
       a leaf for the numerical value), could be any vertice for categorical
27   * @param element  non-generalized attribute
28   * @return binary representation of the non-generalized attribute with respect
       to the corresponding tree (and therefore path from the root of the tree)
29   */
30    public String getPathToLeaf(T element){
31   //first check among the leaves
32       int depth=(int) (Math.log((this.size()+1))/Math.log(2) - 1);
33       for (int i = 0; i<Math.pow(2,depth) ; i++){
34          String k=normkey(Integer.toBinaryString(i),depth );
35
36          if(this.get(k).contains(element)){
37             return k;
38          }
39
40       }
41       //did not find categorical value among the leaves, checking all along
          the tree
42          if (element.getClass().equals(String.class)){
43          System.out.println("looking for cat");
44          for (java.util.Map.Entry<String, Vertice<T>> entry: this.entrySet())
              {
```

```
45              if (entry.getValue().contains(element)){
46                 return entry.getKey();
47              }
48           }
49           }
50        return "";
51     }
52     /**
53      * Method for generalization of the attribute
54      * @param  db DB or a VIEW we are using
55      * @param  G_attr currently being generalized attribute
56      * @param  leaf binary representation of an attribute wrt to the tree
57      * @return generalized attribute wrt to current state of RSDB
58      */
59     //maybe to move this method to the ISyPem2.Node.Case ?
60     public String getG_Value(String db, String G_attr, String leaf)
61     {
62        int i=0;
63        String currgenerVal = "";
64        String TgenerVal= "";
65        String FgenerVal= "";
66        boolean tres;
67        boolean fres;
68
69        while (i<leaf.length() ){
70           TgenerVal=(currgenerVal+leaf.substring(i, i+1));
71           if (leaf.substring(i, i+1).equals("0" ) ){
72              FgenerVal=leaf.substring(0, i)+"1";
73              }
74           else FgenerVal=(currgenerVal+"0");
75
76     //TODO randomize query order
77           tres=queries.SearchForGen (db,G_attr, TgenerVal);
78           fres=queries.SearchForGen(db,G_attr, FgenerVal);
79
80     // TODO mask the case when there is only entries corresponding to FgenerVal
          that means that we stop and the DB knows that there is no enries with
       the gen value we r looking for.
81     // TODO introduce the parameter that specifies the number of fake quiries
       after we decide to stop
82           if (tres && fres){
83              currgenerVal=TgenerVal;
84              i++;
85              }
86           else i=leaf.length();
87
88        }
89        if (currgenerVal.length()>0){
90           return currgenerVal;
91        }
92        else
93           return "*";
94        }
95  /**
```

```
96   *
97   * @return true if the attribute is numerical, false if categorical
98   */
99     public boolean isRange(){
100       return this.isRange;
101    }
102
103    @SuppressWarnings("unchecked")
104    public String getPathToLeaf(NumericalAttribute element) {
105        //for(java.util.Map.Entry<String, Vertice<T>> entry : this.entrySet()){
106       int depth=(int) (Math.log((this.size()+1))/Math.log(2) - 1);
107       for (int i = 0; i<Math.pow(2,depth) ; i++){
108          String k=normkey(Integer.toBinaryString(i),depth );
109
110          if(this.get(k).contains((T) element)){
111             return k;
112          }
113
114       }
115       return "";
116    }
117 /**
118  * Normalizes the key value of the AttributeTree (used in the phase of
       constructing the tree from the file)
119  * @param str binary string, representation of an int
120  * @param count current depth of the tree
121  * @return normalized string (with the number of 0s added such that the length
       is equal to the current depth of the tree)
122  */
123    public static String normkey(String str, int count){
124       while (count!=str.length()){
125       str = "0"+str;
126       }
127       return str;
128    }
129
130 }
```

Category.java

```java
package ch.hevs.ISyPeM2.Generalization;
/**
 * Implements interface Vertice for a categorical (de-)generalization tree
 * @author Alevtina
 *
 */
public class Category implements Vertice<String>{

   private final String value;
    /**
     * Class constructor
     *
     * @param value string, that represents generalized value (corresponding to the different value of generalization)
     */
   public Category(String value){
      this.value = value;
   }

   public String value(){
      return this.value;
   }

   public String toString(){
      return "["+value+"]";
   }

   @Override
   /**
    * Method checks whether the @element belongs to the to the leaf (=equal to the leaf)
    */
   public boolean contains(String element) {
      if(element.equals(value)){
         return true;
      }
      else{
         return false;
      }
   }
}
```

Main.java

```
1  package ch.hevs.ISyPeM2.Generalization;
2
3
4  import ch.hevs.ISyPeM2.Tests.CaseGeneralizationTest;
5
6
7
8
9
10 public class Main {
11
12
13
14
15    public static void main(String[] args) throws Exception {
16
17       CaseGeneralizationTest.testCaseGen();
18
19        }
20    }
```

Range.java

```java
package ch.hevs.ISyPeM2.Generalization;
/**
 * Implements interface Vertice for a numerical (de-)generalization tree
 * @author Alevtina
 *
 */
public class Range implements Vertice<Double>{

   private final Double min;
   private final Double max;
    /**
     * Class constructor
     *
     * @param min lower bound of a range that represents a vertice of a tree
        corresponding to the numerical attribute
     * @param max upper bound of a range
     */
   public Range(Double min, Double max){
      this.min = min;
      this.max = max;
   }

   public Double min(){
      return this.min;
   }

   public Double max(){
      return this.max;
   }

   public String toString(){
      return "["+min+", "+max+"]";
   }

   @Override
   public boolean contains(Double element) {
      if(min <= element && element < max){
         return true;
      }
      else{
         return false;
      }
   }
}
```

TreeCollection.java

```
1  package ch.hevs.ISyPeM2.Generalization;
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileInputStream;
6  import java.io.FileNotFoundException;
7  import java.io.FileReader;
8  import java.io.IOException;
9  import java.io.InputStream;
10 import java.io.InputStreamReader;
11 import java.util.Iterator;
12 import java.util.Scanner;
13 import java.util.TreeMap;
14
15 import org.json.simple.JSONArray;
16 import org.json.simple.JSONObject;
17 import org.json.simple.parser.JSONParser;
18 import org.json.simple.parser.ParseException;
19
20
21 /**
22  * Class represents collection of generalization trees for different
        attributes for an instance of RSDB
23  * @author Alevtina
24  *
25  */
26
27 public class TreeCollection extends TreeMap<String, AttributeTree<?>> {
28    private static final long serialVersionUID = 2L;
29
30    @SuppressWarnings("unchecked")
31    public AttributeTree<Double> getRange(String attribute) {
32       AttributeTree<?> at = get(attribute);
33       if(at.isRange()){
34          return (AttributeTree<Double>) at;
35       }
36
37       return null;
38    }
39
40    @SuppressWarnings("unchecked")
41    public AttributeTree<String> getCategory(String attribute) {
42       AttributeTree<?> at = get(attribute);
43       if(at.isRange()){
44          return (AttributeTree<String>) at;
45       }
46
47       return null;
48    }
49    /**
50     * TODO PARSE FROM the single FILE
51     * Parses configuration file with the description of generalization trees.
          The trees are stores as a map <name of the attribute, tree for the
```

```
52      attribute>
53     * @param filePath  uri of the description
54     * @return  map <attribute, treeCat>, where treeCat is true if attribute is
           a category, false if numerical value
55     * @throws IOException
56     */
57
58    public TreeMap<String, Boolean> JsonParse(String filePath) throws
          IOException, ParseException {
59
60       TreeMap<String, Boolean> treeIsCat = new TreeMap<String, Boolean>();
61       try {
62          FileReader reader = new FileReader(filePath);
63
64          JSONParser jsonParser = new JSONParser();
65          JSONObject jsonObject = (JSONObject) jsonParser.parse(reader);
66
67
68          JSONArray trees= (JSONArray) jsonObject.get("trees");
69          Iterator j = trees.iterator();
70          while (j.hasNext()) {
71             JSONObject attrTree = (JSONObject) j.next();
72
73             String attrName = (String) attrTree.get("attrName");
74             String category = (String) attrTree.get("property");
75
76             if (category.equals("categorical")){
77                treeIsCat.put(attrName, true);
78                AttributeTree<String> at = new AttributeTree<String>(true);
79
80                JSONArray vertices= (JSONArray) attrTree.get("vertices");
81                Iterator i = vertices.iterator();
82
83                while (i.hasNext()) {
84                   JSONObject innerObj = (JSONObject) i.next();
85                   at.put((String)innerObj.get("code"), new Category((String)
                         innerObj.get("value")));
86
87                }
88
89                this.put(attrName, at);
90
91             }
92
93             else {
94                AttributeTree<Double> at = new AttributeTree<Double>(true);
95                treeIsCat.put(attrName, false);
96                JSONArray vertices= (JSONArray) attrTree.get("vertices");
97                Iterator i = vertices.iterator();
98                while (i.hasNext()) {
99                   JSONObject innerObj = (JSONObject) i.next();
100                  at.put((String)innerObj.get("code"), new Range((Double.
                         parseDouble((String)innerObj.get("valueMin"))), (Double.
```

```
                        parseDouble((String)innerObj.get("valueMax"))));
101                 }
102                 System.out.println(at);
103                 this.put(attrName, at);
104              }
105           }
106           System.out.println(this);
107
108           } catch (FileNotFoundException ex) {
109              ex.printStackTrace();
110           } catch (IOException ex) {
111              ex.printStackTrace();
112           } catch (ParseException ex) {
113              ex.printStackTrace();
114           } catch (NullPointerException ex) {
115              ex.printStackTrace();
116           }
117           return treeIsCat;
118
119
120     }
121  }
```

Vertice.java

```
1  package ch.hevs.ISyPeM2.Generalization;
2  /**
3   *
4   * @author Alevtina
5   *
6   *This interface represents a vertice of a (de-)generalization tree (both for
        categorical and numerical values)
7   */
8  public interface Vertice<T> {
9
10    public boolean contains(T element);
11 }
```

queries.java

```
1  package ch.hevs.ISyPeM2.Generalization;
2  import java.sql.Connection;
3  import java.sql.DriverManager;
4  import java.sql.PreparedStatement;
5  import java.sql.ResultSet;
6  import java.sql.SQLException;
7  import java.sql.Statement;
8
9
10 //TODO document and move to the methods of the case or SQLinterface
       realization
11 public class queries {
12
13    public static void connection() {
14
15       try {
16          Class.forName("com.mysql.jdbc.Driver");
17          // System.out.println("Worked");
18       } catch (ClassNotFoundException e) {
19          System.out.println("no driver found");
20          // TODO Auto-generated catch block
21          e.printStackTrace();
22       }
23    }
24
25
26 public static void ConnectionToMySql1() {
27
28       try{
29             connection();
30       String host="jdbc:mysql://localhost/testgen";
31       String username="root";
32       String password="root";
33
34       Connection connect = DriverManager.getConnection(host, username,
             password);
35       System.out.println("connection established");
36
37       // mysql insert statement (hardcoded)
38       String LDB_1_Update = " insert into LDB_1 (PS, age, gender, G_age,
             G_gender, health_data) values (?, ?, ?, ?, ?, ?)";
39
40     // create the mysql insert preparedstatement
41 /*    PreparedStatement preparedStmtUpd = connect.prepareStatement(
       LDB_1_Update);
42       preparedStmtUpd.setString(1, "Pseudo1");
43       preparedStmtUpd.setLong(2, 39);
44       preparedStmtUpd.setString(3, "female");
45       preparedStmtUpd.setString(4, "");
46       preparedStmtUpd.setString(5, "");
47       preparedStmtUpd.setString(6, "healthy");
48
49 //    preparedStmtUpd.execute();
```

```
50
51        preparedStmtUpd.setString(1, "Pseudo2999");
52        preparedStmtUpd.setLong(2, 26);
53        preparedStmtUpd.setString(3, "male");
54        preparedStmtUpd.setString(4, "");
55        preparedStmtUpd.setString(5, "");
56        preparedStmtUpd.setString(6, "healthy");
57
58        preparedStmtUpd.execute();
59  */
60        // preparedStmt.close();
61        String searchOverLdb = "select * from LDB_1";
62
63       // PreparedStatement preparedStmtSearch = connect
64       // PreparedStatement preparedStmtSearch = connect.prepareStatement(
             searchOverLdb);
65       // preparedStmtSearch.execute();
66       // preparedStmtSearch.close();
67       // System.out.println("query result" );
68
69       Statement stmtSelect=connect.createStatement();
70       ResultSet rsltSelect=stmtSelect.executeQuery(searchOverLdb);
71
72       //Deleting records
73      /* String delete = "delete from LDB_1";
74       Statement stmtDel=connect.createStatement();
75       stmtDel.executeUpdate(delete);
76     */
77    while (rsltSelect.next()) {
78           String PS = rsltSelect.getString("PS");
79           int LDB_age = rsltSelect.getInt("age");
80           String gender = rsltSelect.getString("gender");
81           String G___age = rsltSelect.getString("G_age");
82           System.out.println("PS: " + PS +", LDB_age: " + LDB_age
83                 + ", gender: " + gender + ", G_age: " +G___age);
84       }
85        connect.close();
86
87        }
88        catch (SQLException e) {
89           System.out.println("connection is not established or quiry is not
                successfull");
90           // TODO Auto-generated catch block
91           e.printStackTrace();
92        }
93
94  }
95
96  public static void Select_updView(String db, String clause) { //query view
97
98     try{
99           connection();
100
101          String host="jdbc:mysql://localhost/testgen";
```

```
102           String username="root";
103           String password="root";
104
105           Connection connect = DriverManager.getConnection(host, username,
                 password);
106           System.out.println("connection established");
107 //        String checkstateRSDB_1 = "SELECT * FROM "+db+" WHERE '"+clause+"'";
108
109           String select = "SELECT * FROM stateRSDB_1 WHERE "+clause;
110
111           Statement selectState=connect.createStatement();
112           ResultSet result=selectState.executeQuery(select);
113
114           while (result.next()) {
115
116              int PsNumber = result.getInt("PsNumber");
117              String G_age = result.getString("G_age");
118              String G_gender = result.getString("G_gender");
119
120              System.out.println("PsNumber: " + PsNumber + " G_age: " + G_age +"
                     genGender: " + G_gender );
121           }
122           connect.close();
123
124        }
125        catch (SQLException e) {
126           System.out.println("Select_updView() is not established or quiry is
                 not successfull");
127           // TODO Auto-generated catch block
128           e.printStackTrace();
129        }
130
131     }
132
133 public static boolean SearchForGen (String db, String attr, String gen) {//
     checking statRSDB_1 or view looking for the prefixes of future generalized
     value
134    int PsNumber=0;
135    try{
136           connection();
137           String host="jdbc:mysql://localhost/testgen";
138           String username="root";
139           String password="root";
140           Connection connect = DriverManager.getConnection(host, username,
                 password);
141           Statement stmtSelect=connect.createStatement();
142           //System.out.println("G_attr is "+G_attr);
143           ResultSet rsltSelect=stmtSelect.executeQuery("SELECT PsNumber FROM "+
                 db+" WHERE G_"+attr+" LIKE '"+gen+"%'");
144           while (rsltSelect.next()) {
145              PsNumber = rsltSelect.getInt("PsNumber");
146              //System.out.println("PsNumber...."+PsNumber);
147              }
148           connect.close();
```

```
149
150         }
151         catch (SQLException e) {
152             System.out.println("SearchForGen() is not established or quiry is not
                    successfull");
153             // TODO Auto-generated catch block
154             e.printStackTrace();
155         }
156
157     if (PsNumber > 0){
158         return true;
159         }
160     else return false;
161 }
162
163 public static void Drop_View() {// drop view
164     try{
165             connection();
166             String host="jdbc:mysql://localhost/testgen";
167             String username="root";
168             String password="root";
169             Connection connect = DriverManager.getConnection(host, username,
                    password);
170
171             //System.out.println("G_attr is "+G_attr);
172             PreparedStatement preparedStmtUpd = connect.prepareStatement("DROP
                    VIEW updView");
173             preparedStmtUpd.execute();
174 //    ResultSet rsltCreateVIEW=stmtSelect.executeQuery("create VIEW viewFor_"+
    nextattr+"as SELECT * FROM "+db+" WHERE "+G_attrPrev+"='"+genPrevVal+"'");
175
176             connect.close();
177
178         }
179         catch (SQLException e) {
180             System.out.println("Init_View() is not established or quiry is not
                    successfull");
181             // TODO Auto-generated catch block
182             e.printStackTrace();
183         }
184
185
186 }
187
188
189 public static void Init_View() {// initialize view
190     try{
191             connection();
192             String host="jdbc:mysql://localhost/testgen";
193             String username="root";
194             String password="root";
195             Connection connect = DriverManager.getConnection(host, username,
                    password);
196
```

```
197             //System.out.println("G_attr is "+G_attr);
198             PreparedStatement preparedStmtUpd = connect.prepareStatement("create
                    VIEW updView AS SELECT * FROM stateRSDB_1");
199             preparedStmtUpd.execute();
200 //    ResultSet rsltCreateVIEW=stmtSelect.executeQuery("create VIEW viewFor_"+
        nextattr+"as SELECT * FROM "+db+" WHERE "+G_attrPrev+"='"+genPrevVal+"'");
201
202             connect.close();
203
204         }
205         catch (SQLException e) {
206             System.out.println("Init_View() is not established or quiry is not
                    successfull");
207             // TODO Auto-generated catch block
208             e.printStackTrace();
209         }
210
211
212 }
213
214 public static String CreateClause(String attrPrev, String genPrevVal){
215     return "G_"+attrPrev+"='"+genPrevVal+"'";
216 }
217
218 public static boolean Create_View(String db, String clause) {// checking
        statRSDB_1 looking for the prefixes of future generalized value
219     try{
220             connection();
221             String host="jdbc:mysql://localhost/testgen";
222             String username="root";
223             String password="root";
224             Connection connect = DriverManager.getConnection(host, username,
                    password);
225
226             //System.out.println("G_attr is "+G_attr);
227             PreparedStatement preparedStmtUpd = connect.prepareStatement("create
                    or replace VIEW updView AS SELECT * FROM "+db+" WHERE "+clause);
228             preparedStmtUpd.execute();
229 //    ResultSet rsltCreateVIEW=stmtSelect.executeQuery("create VIEW viewFor_"+
        nextattr+"as SELECT * FROM "+db+" WHERE "+G_attrPrev+"='"+genPrevVal+"'");
230
231             connect.close();
232
233         }
234         catch (SQLException e) {
235             System.out.println("Create_View() is not established or quiry is not
                    successfull");
236             // TODO Auto-generated catch block
237             e.printStackTrace();
238         }
239     return true;
240
241 }
242
```

```
243
244 public static void ConnectionToMySql1upd(String Pseudo, String generalised_age
        ) {
245
246    try{
247           connection();
248           String host="jdbc:mysql://localhost/testgen";
249           String username="root";
250           String password="root";
251
252           Connection connect = DriverManager.getConnection(host, username,
                password);
253           System.out.println("connection established");
254
255           String LDB_1_Update = " update LDB_1 SET G_age='"+generalised_age+"'
                where PS='"+Pseudo+"'";
256
257           PreparedStatement preparedStmtUpd = connect.prepareStatement(
                LDB_1_Update);
258           preparedStmtUpd.execute();
259           preparedStmtUpd.close();
260           connect.close();
261       }
262       catch (SQLException e) {
263           System.out.println("connection is not established or quiry is not
                successfull");
264           // TODO Auto-generated catch block
265           e.printStackTrace();
266       }
267
268 }
269
270 }
```

D.5 ch.hevs.ISyPeM2.Node package

Case.java

```
1  package ch.hevs.ISyPeM2.Node;
2
3  import java.io.IOException;
4  import java.math.BigInteger;
5  import java.security.MessageDigest;
6  import java.util.Calendar;
7  import java.util.HashMap;
8  import java.util.Iterator;
9  import java.util.TreeMap;
10 import java.util.Map.Entry;
11
12 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
13 import ch.hevs.ISyPeM2.Generalization.TreeCollection;
14 import ch.hevs.ISyPeM2.Generalization.queries;
15
16 /**
17  * This class describes a medical case
18  * @author Imanol-Mikel Barba
19  *
20  */
21 public class Case
22 {
23    /**
24     * Case ID
25     */
26    private String ID;
27
28    /**
29     * Case patient Pseudo-identifier
30     */
31    private String PS;
32
33    /**
34     * Case type
35     */
36    private String type;
37
38    /**
39     * {@code Map} with the current case's attributes, using their name as key
40     */
41    private HashMap<String, CaseAttribute> attributes;
42
43    /**
44     * Health data for this case
45     */
46    private String healthData;
47
48    /**
49     * Class constructor
50     * @param PS Case Pseudo-identifier
```

```
51      * @param type Case type
52      * @param healthData Health data of the medical record
53      */
54     public Case(String PS, String type, String healthData)
55     {
56        this.ID = this.hashMD5(Integer.toHexString(System.identityHashCode(this)
              ) +
57              new java.sql.Timestamp(Calendar.getInstance().getTime().getTime())
                  .toString());
58        this.PS = PS;
59        this.type = type;
60        this.attributes = new HashMap<String, CaseAttribute>();
61        this.healthData = healthData;
62     }
63
64     /**
65      * Generates the MD5 hash of a string passed as parameter. This is used to
           generate the case ID.
66      * @param input Input string for the hash function
67      * @return MD5 hash of the input string
68      */
69     private String hashMD5(String input)
70     {
71        String hashtext = "";
72        try
73        {
74           MessageDigest md = MessageDigest.getInstance("MD5");
75           byte[] messageDigest = md.digest(input.getBytes());
76           BigInteger number = new BigInteger(1, messageDigest);
77           hashtext = number.toString(16);
78           while (hashtext.length() < 32)
79           {
80              hashtext = "0" + hashtext;
81           }
82        }
83        catch(Exception e)
84        {
85           ErrorPrinter.printLog("Case", "hashMD5", ErrorPrinter.SEVERITY_ERROR,
                 e.getClass() + ": " + e.getMessage());
86        }
87        return hashtext;
88     }
89
90     /**
91      * {@link #healthData} getter
92      * @return Health data for this medical case
93      */
94     public String getHealthData()
95     {
96        return this.healthData;
97     }
98
99     /**
100     * Adds an attribute to this case. It also adds another one with the same
```

```
         name and a "G_" prefix, denoting it is the
101      * generalized version of the new attribute being added.
102      *
103      * @param attr {@link CaseAttribute} object containing the attribute being
             added and its value
104      */
105     public void addAttribute(CaseAttribute attr)
106     {
107        this.attributes.put(attr.getName(), attr);
108        this.attributes.put("G_" + attr.getName(),new CategoricalAttribute("G_"
               + attr.getName(),null));
109     }
110
111     /**
112      * Adds an attribute to this case. It doesn't add a generalized version of
             the new attribute being added.
113      *
114      * @param attr {@link CaseAttribute} object containing the attribute being
             added and its value
115      */
116     public void setAttribute(CaseAttribute attr)
117     {
118        this.removeAttribute(attr.getName());
119        this.attributes.put(attr.getName(), attr);
120     }
121
122     /**
123      * Retrieves a certain attribute from this case
124      *
125      * @param name Name of the attribute to retrieve
126      * @return {@link CaseAttribute} object containing the requested attribute
             and its value
127      */
128     public CaseAttribute getAttribute(String name)
129     {
130        return attributes.get(name);
131     }
132
133     /**
134      * Retrieves an iterator to read all the attributes of this case
135      * @return Iterator for the {@link #attributes} {@code Map}
136      */
137     public Iterator<CaseAttribute> getAttributes()
138     {
139        return this.attributes.values().iterator();
140     }
141
142     /**
143      * Removes an attribute from this case
144      *
145      * @param name Name of the attribute to remove
146      */
147     public void removeAttribute(String name)
148     {
```

```
149         attributes.remove(name);
150     }
151 
152     /**
153      * {@link #PS} getter
154      * @return Pseudo-identifier of the case
155      */
156     public String getPS()
157     {
158         return PS;
159     }
160 
161     /**
162      * {@link #ID} getter
163      * @return ID of the case
164      */
165     public String getID()
166     {
167         return ID;
168     }
169 
170     /**
171      * {@link #ID} getter
172      * @param ID new ID for the case
173      */
174     public void setID(String ID)
175     {
176         this.ID = ID;
177     }
178 
179     /**
180      * {@link #type} getter
181      * @return Type of the case
182      */
183     public String getType()
184     {
185         return type;
186     }
187     /**
188      * TODO add update of case collection with the generalized values
189      * TODO add links in description
190      * Method for generalizing the case
191      * @param treeColl  Generalization trees
192      * @param treeCat Map <attributes, type(categorical or numerical)> for all
           trees; returned be the method parseConfig from TreeCollection}
193      * @throws IOException
194      */
195     public void generalizeCase(TreeCollection treeColl, TreeMap<String, Boolean
           > treeCat) throws IOException{
196 
197 //      TreeCollection treeColl = new TreeCollection();
198 //      TreeMap<String, Boolean> treeCat = new TreeMap<String, Boolean>();
199 //      treeCat=treeColl.parseCofig("/Users/uadmin/Documents/gits/project/
       inputfiles/");
```

```
200
201         String leaf="";
202         String genleaf="";
203         String Clause="";
204         queries.Drop_View();
205         queries.Init_View();
206
207 for(Entry<String, Boolean> entry :treeCat.entrySet()){
208    if(entry.getValue()){//categorical
209    leaf=treeColl.getCategory(entry.getKey()).getPathToLeaf(this.getAttribute(
           entry.getKey()).getValue()); //getting binary repr of the leaf
210    System.out.println("catAttr: "+this.getAttribute(entry.getKey()).getValue()
           );
211    genleaf=treeColl.getCategory(entry.getKey()).getG_Value("updView",entry.
           getKey(), leaf);
212    System.out.println("leaf "+leaf);
213    System.out.println("genleaf "+genleaf);
214    if (Clause.length()==0){
215       Clause=Clause+queries.CreateClause(entry.getKey(), genleaf);
216    }
217    else{
218       Clause=Clause+" AND "+queries.CreateClause(entry.getKey(), genleaf);
219    }
220    System.out.println("Clause categ "+Clause);
221    queries.Create_View("stateRSDB_1", Clause);
222    queries.Select_updView("stateRSDB_1", Clause);
223
224    }
225
226    else{//range
227
228
229       leaf=treeColl.getRange(entry.getKey()).getPathToLeaf(this.getAttribute(
              entry.getKey()).getActualValue()); //getting binary repr of the leaf
230       System.out.println("rangeAttr: "+this.getAttribute(entry.getKey()).
              getActualValue());
231       genleaf=treeColl.getRange(entry.getKey()).getG_Value("updView",entry.
              getKey(), leaf);
232       System.out.println("leaf "+leaf);
233       System.out.println("genleaf "+genleaf);
234       if (Clause.length()==0){
235          Clause=Clause+queries.CreateClause(entry.getKey(), genleaf);
236       }
237       else{
238          Clause=Clause+" AND "+queries.CreateClause(entry.getKey(), genleaf);
239       }
240       System.out.println("Clause categ "+Clause);
241       queries.Create_View("stateRSDB_1", Clause);
242       queries.Select_updView("stateRSDB_1", Clause);
243       }
244
245 }
246
247
```

```
248         System.out.println(treeColl);
249         System.out.println(treeCat);
250 
251 
252          }
253 
254 }
```

CaseAggregation.java

```
1  package ch.hevs.ISyPeM2.Node;
2
3  import java.util.Calendar;
4  import java.util.HashMap;
5  import java.util.Iterator;
6  import java.util.concurrent.locks.Lock;
7  import java.util.concurrent.locks.ReentrantLock;
8
9  /**
10  * This class represents an aggregation of case collections with the same type
11  *
12  * @author Imanol-Mikel Barba Sabariego
13  *
14  */
15 public class CaseAggregation
16 {
17    /**
18     * Name of the case aggregation. It's the type of medical cases it
          comprises
19     */
20    private String name;
21
22    /**
23     * Timestamp with the last modification date to any of the collections in
          this aggregation
24     */
25    private String timestamp;
26
27    /**
28     * Description of the type of case collections this aggregation holds
29     */
30    private String description;
31
32    /**
33     * {@code Map} holding all the case collections in the aggregation, using
          their ID as key
34     */
35    private HashMap<String,CaseCollection> collections;
36
37    /**
38     * {@code Lock} to perform mutually exclusive operations in the collection
39     */
40    private Lock lock;
41
42    /**
43     * Class constructor
44     *
45     * @param name Name of the aggregation
46     * @param description Description of the cases in the aggregation
47     */
48    public CaseAggregation(String name, String description)
49    {
50       this.name = name;
```

```
51          this.description = description;
52          this.collections = new HashMap<String,CaseCollection>();
53          this.lock = new ReentrantLock();
54          this.updateTimestamp();
55      }
56
57      /**
58       * Acquires the {@code Lock} to ensure mutual exclusion between critical
             methods.
59       *
60       * <p>Due to the fact that {@link #lock} is instantiated as a {@code
             ReentrantLock}, this method may be called repeatedly
61       * as long as the lock is owned by the current thread. In each critical
             method, the lock is locked and unlocked using
62       * the {@link #startAtomicOperation()} and {@link #stopAtomicOperation()}
             methods, but if we start an atomic operation
63       * before, we can perform multiple critical operations without releasing
             the lock</p>
64       */
65      public synchronized void startAtomicOperation()
66      {
67          lock.lock();
68      }
69
70      /**
71       * Frees the {@code Lock} to finish mutual exclusion situations between
             methods.
72       */
73      public void stopAtomicOperation()
74      {
75          lock.unlock();
76      }
77
78      /**
79       * {@link #name} getter
80       * @return Name of the case aggregation
81       */
82      public String getName()
83      {
84          startAtomicOperation();
85          String t = new String(this.name);
86          stopAtomicOperation();
87          return t;
88      }
89
90      /**
91       * Retrieves the last modification date timestamp for this case aggregation
92       * @return Timestamp with the last modification date
93       */
94      public String getTimestamp()
95      {
96          startAtomicOperation();
97          String time = new String(this.timestamp);
98          stopAtomicOperation();
```

```
99          return time;
100     }
101
102     /**
103      * Sets the last modification date timestamp for this case aggregation
104      * @param timestamp New timestamp for the last modification date
105      */
106     public void setTimestamp(String timestamp)
107     {
108         startAtomicOperation();
109         this.timestamp = timestamp;
110         stopAtomicOperation();
111     }
112
113     /**
114      * Updates the timestamp of a the current case aggregation to the present
            moment
115      */
116     public void updateTimestamp()
117     {
118         this.setTimestamp(new java.sql.Timestamp(Calendar.getInstance().getTime
                ().getTime()).toString());
119     }
120
121     /**
122      * {@link #description} getter
123      * @return Description of the case aggregation
124      */
125     public String getDescription()
126     {
127         startAtomicOperation();
128         String d = new String(this.description);
129         stopAtomicOperation();
130         return d;
131     }
132
133     /**
134      * Changes the description of this case aggregation
135      * @param description New description to be set
136      */
137     public void setDescription(String description)
138     {
139         startAtomicOperation();
140         this.description = description;
141         stopAtomicOperation();
142     }
143
144     /**
145      * Adds a case collection to the current aggregation
146      * @param collection Case collection to be added
147      */
148     public void addCollection(CaseCollection collection)
149     {
150         startAtomicOperation();
```

```
151         this.removeCollection(collection.getID());
152         this.collections.put(collection.getID(), collection);
153         stopAtomicOperation();
154     }
155 
156     /**
157      * Removes a case collection from this aggregation
158      * @param collectionID ID of the case collection to be removed
159      */
160     public void removeCollection(String collectionID)
161     {
162         startAtomicOperation();
163         this.collections.remove(collectionID);
164         stopAtomicOperation();
165     }
166 
167     /**
168      * Checks if this aggregation holds a certain case collection
169      * @param collectionID ID of the collection to be checked
170      * @return {@code true} if the aggregation contains a collection with the
         same ID, {@code false} otherwise
171      */
172     public boolean hasCollection(String collectionID)
173     {
174         startAtomicOperation();
175         boolean hasIt = this.collections.containsKey(collectionID);
176         stopAtomicOperation();
177         return hasIt;
178     }
179 
180     /**
181      * Retrieves a certain case collection from this aggregation
182      * @param collectionID ID of the collection to be fetched
183      * @return {@link CaseCollection} object with the collection requested
184      */
185     public CaseCollection getCollection(String collectionID)
186     {
187         startAtomicOperation();
188         CaseCollection collection = this.collections.get(collectionID);
189         stopAtomicOperation();
190         return collection;
191     }
192 
193     /**
194      * Retrieve the case collections contained in this case aggregation
195      * @return {@code Iterator} to access all the case collections forming this
           aggregation
196      */
197     public Iterator<CaseCollection> getCollections()
198     {
199         return collections.values().iterator();
200     }
201 
202     /**
```

```
203      * Counts the number of cases the collections in this aggregation have.
204      *
205      * @return Total number of cases in this aggregation
206      */
207     public int getNumCases()
208     {
209        int num = 0;
210        Iterator<CaseCollection> it = this.collections.values().iterator();
211        while(it.hasNext())
212        {
213           num += it.next().numCases();
214        }
215        return num;
216     }
217
218  }
```

CaseAttribute.java

```java
package ch.hevs.ISyPeM2.Node;

import java.util.regex.Pattern;

/**
 * This abstract class represents an attribute or field of a certain medical
    case. It must be subclassed into other classes
 * according to the type of attribute intending to represent
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public abstract class CaseAttribute
{
   /**
    * Name of the attribute
    */
   private String name;

   /**
    * Class constructor
    * @param name Name of the attribute
    */
   public CaseAttribute(String name)
   {
      this.name = name;
   }

   /**
    * {@link #name} getter
    * @return The name of the attribute
    */
   public String getName()
   {
      return this.name;
   }

   /**
    * This abstract method returns the value of the attribute which depends on
       the type of attribute the object is
    * @return Value of the attribute
    */
   public abstract String getValue();

   //TODO: Fix this
   /**
    * For the numerical non-generalized value
    * @return numerical value (one double)
    */
   public abstract double getActualValue();


    /**
```

```
52      * Overriden {@code equals(Object)} method. This method checks if two
            attributes are the same by identity or content
53      */
54     @Override
55     public boolean equals(Object obj)
56     {
57        if(!(obj instanceof CaseAttribute))
58        {
59           return false;
60        }
61        if(obj == this)
62        {
63           return true;
64        }
65        CaseAttribute attr = (CaseAttribute)obj;
66        return (attr.getName().equals(this.getName()) && ((attr.getValue() ==
              null) && (this.getValue() == null)) || ((attr.getValue().equals(this.
              getValue()))));
67     }
68
69     /**
70      * This method parses the value of an attribute to determine which subclass
            of attribute it is and returns a new
71      * instance of that {@code CaseAttribute} subclass with the value set
72      *
73      * @param name Name of the attribute
74      * @param val String representing the value of the attribute
75      * @return Instance of the {@code CaseAttribute} subclass that the value
            represents
76      */
77     public static CaseAttribute parseCaseAttribute(String name, String val)
78     {
79        CaseAttribute attr;
80        if(name.startsWith("'") && name.endsWith("'"))
81        {
82           name = name.substring(1, name.length()-1);
83        }
84        if(val.startsWith("'") && val.endsWith("'"))
85        {
86           val = val.substring(1, val.length()-1);
87        }
88        if(Pattern.compile("\\[\\d+(\\.\\d+)*,\\d+(\\.\\d+)*\\]").matcher(val).
              matches())
89        {
90           //NumericalAttribute case
91           int commaPos = val.indexOf(",");
92           attr = new NumericalAttribute(name,Double.parseDouble(val.substring
                 (1,commaPos)),Double.parseDouble(val.substring(commaPos + 1, val.
                 length()-1)));
93        }
94        else
95        {
96           //CategoricalAttribute case
97           if(val.equals("null"))
```

```
98              {
99                 val = null;
100             }
101             attr = new CategoricalAttribute(name,val);
102          }
103          return attr;
104       }
105
106       /**
107        * Method that takes a {@link CaseAttribute} as argument and returns a
              generalised version of it
108        * @param attr Case attribute to be generalised
109        * @return {@link CaseAttribute} object with a generalised version of the
              case attribute passed as argument
110        */
111       public static CaseAttribute generalizeAttribute(CaseAttribute attr, String
              tree, int K)
112       {
113          //TODO: Implement generalization (Alevtina)
114          //will probably have to pass K and binary tree as parameter
115          // Alevtina: binary trees will be passed as a URI of the directory where
                the description of the trees is stored
116          // Alevtina: for updating RSDB with one single case we base
                generalization on the current state of the statRSDB_1,
117          //        meaning that we don't need pass K, however for initialization
                of RSDB (generalization locally we need)
118          if(attr instanceof NumericalAttribute)
119          {
120             NumericalAttribute numAttr = (NumericalAttribute)attr;
121             return new NumericalAttribute("G_" + numAttr.getName(),numAttr.getMin
                   (),numAttr.getMax());
122          }
123          else
124          {
125             CategoricalAttribute catAttr = (CategoricalAttribute)attr;
126             return new CategoricalAttribute("G_" + catAttr.getName(), catAttr.
                   getValue());
127          }
128       }
129    }
```

CaseCollection.java

```
1  package ch.hevs.ISyPeM2.Node;
2
3  import java.util.HashMap;
4  import java.util.HashSet;
5  import java.util.Iterator;
6  import java.util.Set;
7  import java.util.concurrent.locks.Lock;
8  import java.util.concurrent.locks.ReentrantLock;
9
10 /**
11  * This class represents a collection of medical cases with the same type and
        attributes
12  * @author Imanol-Mikel Barba Sabariego
13  *
14  */
15 public class CaseCollection
16 {
17    /**
18     * {@code Map} of cases, using their name as the key
19     */
20    private HashMap<String,Case> collection;
21
22    /**
23     * ID of the case collection
24     */
25    private String ID;
26
27    /**
28     * Type of medical condition of this case collection
29     */
30    private String type;
31
32    /**
33     * Description of the type of medical cases this collection holds
34     */
35    private String description;
36
37    /**
38     * {@code Lock} to perform mutually exclusive operations in the collection
39     */
40    private Lock lock;
41
42    /**
43     * {@code Set} with the case attributes that all the medical cases in this
          collection have in common
44     */
45    private HashMap<String,CaseAttribute> attributes;
46
47    /**
48     * {@code Set} with the keywords this set has
49     */
50    private HashSet<String> keywords;
51
```

```
52      /**
53       * Node from which the current case collection originated
54       */
55      private Node originNode;
56
57      /**
58       * Class constructor
59       * @param ID Name of the case collection
60       * @param type Type of the case collection
61       * @param description Description of the case collection
62       * @param originNode Node from which the case collection was retrieved
63       */
64      public CaseCollection(String ID, String type, String description, Node
             originNode)
65      {
66         this.ID = ID;
67         this.type = type;
68         this.description = description;
69         this.originNode = originNode;
70         this.collection = new HashMap<String,Case>();
71         this.attributes = new HashMap<String,CaseAttribute>();
72         this.keywords = new HashSet<String>();
73         this.lock = new ReentrantLock();
74      }
75
76      /**
77       * Acquires the {@code Lock} to ensure mutual exclusion between critical
            methods.
78       *
79       * <p>Due to the fact that {@link #lock} is instantiated as a {@code
            ReentrantLock}, this method may be called repeatedly
80       * as long as the lock is owned by the current thread. In each critical
            method, the lock is locked and unlocked using
81       * the {@link #startAtomicOperation()} and {@link #stopAtomicOperation()}
            methods, but if we start an atomic operation
82       * before, we can perform multiple critical operations without releasing
            the lock</p>
83       */
84      public synchronized void startAtomicOperation()
85      {
86         lock.lock();
87      }
88
89      /**
90       * Frees the {@code Lock} to finish mutual exclusion situations between
            methods.
91       */
92      public void stopAtomicOperation()
93      {
94         lock.unlock();
95      }
96
97      /**
98       * Adds a case to the collection.
```

```
99      *
100     * <p>If the case is already included in the collection, it replaces it</p>
101     * @param c {@link Case} object to be added
102     * @throws Exception If the operation fails due to attribute mismatch
103     */
104    public void addCase(Case c) throws Exception
105    {
106       startAtomicOperation();
107
108       /* Imanol Barba
109        *
110        * It's the first time a case is added to this collection, so we should
              add the attributes to the collection.
111        *
112        * The attribute's values will be dynamically updated, but we use the
              new case's as the initial value.
113        */
114       if(collection.isEmpty())
115       {
116          Iterator<CaseAttribute> itAttr = c.getAttributes();
117          while(itAttr.hasNext())
118          {
119             CaseAttribute attr = itAttr.next();
120             this.setAttribute(attr);
121          }
122          collection.put(c.getID(), c);
123       }
124       else
125       {
126          /* Imanol Barba
127           *
128           * Sanity check before we add the case: We check that the case's
                attributes are the same of the collection's
129           */
130          Iterator<CaseAttribute> itAttr = c.getAttributes();
131          while(itAttr.hasNext())
132          {
133             CaseAttribute attr = itAttr.next();
134             CaseAttribute colAttr = this.getAttribute(attr.getName());
135             if(colAttr.getClass().equals(attr.getClass()) && colAttr.getName()
                   .equals(attr.getName()))
136             {
137                this.updateAttribute(attr);
138                continue;
139             }
140             else
141             {
142                stopAtomicOperation();
143                throw new Exception("Tried to add a case whose attributes do
                      not match the collection's");
144             }
145          }
146          removeCase(c.getID());
147          collection.put(c.getID(),c);
```

```
148         }
149         stopAtomicOperation();
150     }
151 
152     /**
153      * Removes a case from the case collection
154      * @param ID ID of the case to be removed
155      */
156     public void removeCase(String ID)
157     {
158         startAtomicOperation();
159         collection.remove(ID);
160         stopAtomicOperation();
161     }
162 
163     /**
164      * Method to obtain the number of cases the collection has
165      * @return Number of cases of the collection
166      */
167     public int numCases()
168     {
169         startAtomicOperation();
170         int num = this.collection.size();
171         stopAtomicOperation();
172         return num;
173     }
174 
175     //TODO: Document method
176     public RSDBCollection toRSDB(String tree, int K)
177     {
178         RSDBCollection RSDB = new RSDBCollection(this,tree,K);
179         return RSDB;
180     }
181 
182     /**
183      * Joins another case collection's cases with this one, excluding those
          that have the same ID (duplicates)
184      *
185      * @param collection Collection to be merged
186      * @return Number of cases obtained from the merging
187      */
188     public int unify(CaseCollection collection)
189     {
190         Set<String> nameSet = this.attributes.keySet();
191         Iterator<CaseAttribute> itCaseB = collection.getAttributes();
192         while(itCaseB.hasNext())
193         {
194             if(!nameSet.contains(itCaseB.next().getName()))
195             {
196                 // Attribute mismatch between the two collections
197                 return 0;
198             }
199         }
200         Set<String> caseIDSet = this.collection.keySet();
```

```
201         Iterator<Case> itCase = collection.getCases();
202         int casesAdded = 0;
203         while(itCase.hasNext())
204         {
205            Case c = itCase.next();
206            if(!caseIDSet.contains(c.getID()))
207            {
208               try
209               {
210                  this.addCase(c);
211                  casesAdded++;
212               }
213               catch(Exception e)
214               {
215                  //This should not happen because we checked before that both
                       collections share the same attributes
216               }
217            }
218         }
219         return casesAdded;
220      }
221
222      /**
223       * {@link #ID} getter
224       * @return ID of the case collection
225       */
226      public String getID()
227      {
228         startAtomicOperation();
229         String n = new String(this.ID);
230         stopAtomicOperation();
231         return n;
232      }
233
234      /**
235       * {@link #type} getter
236       * @return Type of the case collection
237       */
238      public String getType()
239      {
240         startAtomicOperation();
241         String n = new String(this.type);
242         stopAtomicOperation();
243         return n;
244      }
245
246      /**
247       * {@link #description} getter
248       * @return Description of the case collection
249       */
250      public String getDescription()
251      {
252         startAtomicOperation();
253         String d = new String(this.description);
```

```
254         stopAtomicOperation();
255         return d;
256      }
257
258      /**
259       * Changes the description of this case collection
260       * @param description New description to be set
261       */
262      public void setDescription(String description)
263      {
264         startAtomicOperation();
265         this.description = description;
266         stopAtomicOperation();
267      }
268
269      /**
270       * Retrieves an {@code Iterator} for the case current case collection
271       * @return Iterator for this case collection
272       */
273      public Iterator<Case> getCases()
274      {
275         return collection.values().iterator();
276      }
277
278      /**
279       * Adds an attribute to the set of case attributes that all the cases in
             this collection have in common. It won't add
280       * a generalized version of the new attribute being added.
281       *
282       * @param attr String with the name of the attribute to be added
283       */
284      public void setAttribute(CaseAttribute attr)
285      {
286         this.attributes.remove(attr.getName());
287         attributes.put(attr.getName(),attr);
288      }
289
290      /**
291       * Retrieves all the case attributes that all the cases in this collection
             have in common
292       * @return {@code Iterator} to access all of the attributes
293       */
294      public Iterator<CaseAttribute> getAttributes()
295      {
296         return attributes.values().iterator();
297      }
298
299      /**
300       * Retrieves a collection attribute
301       * @param attrName Name of the attribute to retrieve
302       * @return The specified attribute
303       */
304      public CaseAttribute getAttribute(String attrName)
305      {
```

```
306         return this.attributes.get(attrName);
307     }
308 
309     /**
310      * This method is called when a case is added to the collection to see if
             any of its attributes is beyond the boundaries of the one from
311      * its collection.
312      *
313      * @param attr Attribute from a case being added
314      */
315     private void updateAttribute(CaseAttribute attr)
316     {
317         CaseAttribute colAttr = this.getAttribute(attr.getName());
318         if(attr instanceof NumericalAttribute)
319         {
320             double newMin = ((NumericalAttribute)colAttr).getMin();
321             double newMax = ((NumericalAttribute)colAttr).getMax();
322             if(((NumericalAttribute) attr).getMin() < ((NumericalAttribute)
                    colAttr).getMin())
323             {
324                 newMin = ((NumericalAttribute) attr).getMin();
325             }
326             if(((NumericalAttribute) attr).getMax() > ((NumericalAttribute)
                    colAttr).getMax())
327             {
328                 newMax = ((NumericalAttribute) attr).getMax();
329             }
330             this.attributes.remove(colAttr.getName());
331             this.attributes.put(attr.getName(),new NumericalAttribute(attr.
                    getName(), newMin, newMax));
332         }
333         else
334         {
335             if(attr.getValue() != null)
336             {
337                 if( (!colAttr.getValue().equals("*")) && (!colAttr.getValue().
                        equals(attr.getValue())) )
338                 {
339                     this.attributes.remove(colAttr.getName());
340                     this.attributes.put(attr.getName(),new CategoricalAttribute(
                            attr.getName(), "*"));
341                 }
342             }
343         }
344     }
345 
346     /**
347      * Adds a keyword to the {@link #keywords} set
348      *
349      * @param kw String with the name of the attribute to be added
350      */
351     public void addKeyword(String kw)
352     {
353         this.keywords.add(kw);
```

```
354     }
355
356     /**
357      * Gets an iterator to read all the keywords in this collection
358      *
359      * @return Iterator for the {@link #keywords} set
360      */
361     public Iterator<String> getKeywords()
362     {
363        return this.keywords.iterator();
364     }
365
366     /**
367      * Counts the number of keywords this collection has
368      *
369      * @return Number of elements of the {@link #keywords} set
370      */
371     public int getNumKeywords()
372     {
373        return this.keywords.size();
374     }
375
376     /**
377      * Checks if the {@link #keywords} set has a certain keyword
378      *
379      * @param kw Keyword to be searched
380      * @return {@code true} if this collection contains the specified keyword,
           {@code false} otherwise
381      */
382     public boolean hasKeyword(String kw)
383     {
384        return this.keywords.contains(kw);
385     }
386
387     /**
388      * Counts the number of attributes this collection has
389      *
390      * @return Number of elements of the {@link #attributes} set
391      */
392     public int getNumAttributes()
393     {
394        return this.attributes.size();
395     }
396
397     /**
398      * Checks if the {@link #attributes} set has a certain attribute name
399      *
400      * @param attrName attribute name to be searched
401      * @return {@code true} if this collection contains the specified attribute
           , {@code false} otherwise
402      */
403     public boolean hasAttribute(String attrName)
404     {
405        return this.attributes.containsKey(attrName);
```

```
406     }
407
408     /**
409      * Returns the node of origin for this case collection
410      * @return Node from which all the cases in this case collection originated
411      */
412     public Node getOriginNode()
413     {
414        return originNode;
415     }
416     /**
417      * TODO implement queries (first organize implemented ones)
418      * Method for initializing RSDB by applying modified Mondrian algorithm for
             local anonymization
419      */
420     public void localGeneralization(){
421
422     }
423 }
```

CategoricalAttribute.java

```java
package ch.hevs.ISyPeM2.Node;

/**
 * This class represents a {@link CaseAttribute} whose value is a plain string
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class CategoricalAttribute extends CaseAttribute
{
   /**
    * String containing the value of this attribute
    */
   private String value;

   /**
    * Class constructor
    *
    * @param name Name of the attribute
    * @param value Value of the attribute
    */
   public CategoricalAttribute(String name, String value)
   {
      super(name);
      this.value = value;
   }

   /**
    * {@link #value} getter
    * @return The value of the attribute
    */
   @Override
   public String getValue()
   {
      return this.value;
   }


    // Maybe it is not optimal, but we need this method for numerical values (
        ranges)
   //TODO: Fix this
   @Override
   public double getActualValue() {
      // TODO Auto-generated method stub
      return 0;
   }
}
```

Node.java

```
1  package ch.hevs.ISyPeM2.Node;
2
3  import java.util.HashMap;
4  import java.util.HashSet;
5  import java.util.Iterator;
6  import java.util.concurrent.locks.Lock;
7  import java.util.concurrent.locks.ReentrantLock;
8
9  import org.json.simple.JSONArray;
10 import org.json.simple.JSONObject;
11 import org.json.simple.parser.JSONParser;
12
13 import alice.logictuple.exceptions.InvalidLogicTupleException;
14 import alice.tucson.api.exceptions.TucsonInvalidAgentIdException;
15 import alice.tucson.api.exceptions.TucsonInvalidTupleCentreIdException;
16 import alice.tucson.api.exceptions.TucsonOperationNotPossibleException;
17 import alice.tucson.api.exceptions.UnreachableNodeException;
18 import alice.tuplecentre.api.exceptions.OperationTimeOutException;
19 import ch.hevs.ISyPeM2.ContextBroker.Attribute;
20 import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
21 import ch.hevs.ISyPeM2.ContextBroker.NodeAttribute;
22 import ch.hevs.ISyPeM2.ContextBroker.Subscription;
23 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
24 import ch.hevs.ISyPeM2.DB.CaseProvisioner;
25 import ch.hevs.ISyPeM2.NotificationServer.NotificationServer;
26 import ch.hevs.ISyPeM2.TuCSoN.NegotiatingAgent;
27 import ch.hevs.ISyPeM2.TuCSoN.SpawningAgent;
28 import ch.hevs.ISyPeM2.TuCSoN.TuCSoNService;
29
30 /**
31  * This class describes a node in the network
32  * @author Imanol-Mikel Barba Sabariego
33  *
34  */
35 public class Node
36 {
37    /**
38     * Node name
39     */
40
41    private String name;
42    /**
43     * IP address of the node
44     */
45    private String address;
46
47    /**
48     * Backend that manages the Node's read/write operations on the medical
          case database
49     */
50    private CaseProvisioner provisioner;
51
52    /**
```

```
53      * {@code Set} with all the case aggregations that have been published
54      */
55     private HashSet<String> publishedAggregations;
56
57     /**
58      * {@code Map} with all the active host negotiating agents within this node
            using the case type they are negotiating as key
59      */
60     private HashMap<String,HashSet<NegotiatingAgent>> activeNegotiatingAgents;
61
62     /**
63      * TuCSoN service running in this node
64      */
65     private TuCSoNService tucsonService;
66
67     /**
68      * Spawning agent residing in this node to process incoming negotiation
           requests
69      */
70     private SpawningAgent spawningAgent;
71
72     /**
73      * {@code Map} with all the {@link NotificationServer} instances running in
            this node, using the {@link ContextBroker}
74      * instances they are listening to as key
75      */
76     private HashMap<ContextBroker,NotificationServer> notifyServers;
77
78     /**
79      * {@code Map} containing all the active subscriptions of this node using
           the {@link NegotiatingAgent}
80      * that made started them as key. This is used to keep track of active
           subcriptions and terminate
81      * them when needed
82      */
83     private HashMap<NegotiatingAgent,ContextBroker> activeSubscriptions;
84
85     /**
86      * {@code Lock} to perform mutually exclusive operations on the {@code Node
           }
87      */
88     private Lock lock;
89
90     /**
91      * Class constructor. It also starts a Spawning Agent in the local node.
92      * @param name Name of the node
93      * @param address IP address of the node
94      */
95     public Node(String name, String address)
96     {
97        this.name = name;
98        this.address = address;
99        this.publishedAggregations = new HashSet<String>();
```

```
100         this.activeNegotiatingAgents = new HashMap<String,HashSet<
                NegotiatingAgent>>();
101         this.activeSubscriptions = new HashMap<NegotiatingAgent,ContextBroker>()
                ;
102         this.notifyServers = new HashMap<ContextBroker,NotificationServer>();
103         this.lock = new ReentrantLock();
104     }
105 
106     /**
107      * Acquires the {@code Lock} to ensure mutual exclusion between critical
                methods.
108      *
109      * <p>Due to the fact that {@link #lock} is instantiated as a {@code
                ReentrantLock}, this method may be called repeatedly
110      * as long as the lock is owned by the current thread. In each critical
                method, the lock is locked and unlocked using
111      * the {@link #startAtomicOperation()} and {@link #stopAtomicOperation()}
                methods, but if we start an atomic operation
112      * before, we can perform multiple critical operations without releasing
                the lock</p>
113      */
114     public void startAtomicOperation()
115     {
116         lock.lock();
117     }
118 
119     /**
120      * Frees the {@code Lock} to finish mutual exclusion situations between
                methods.
121      */
122     public void stopAtomicOperation()
123     {
124         lock.unlock();
125     }
126 
127     /**
128      * Starts the {@link SpawningAgent} in this node
129      * @throws TucsonInvalidAgentIdException
130      */
131     public void startTuCSoN() throws TucsonInvalidAgentIdException
132     {
133         this.tucsonService = new TuCSoNService(this.getAddress(),TuCSoNService.
                DEFAULT_PORT,this);
134         this.spawningAgent = new SpawningAgent(this.tucsonService);
135         this.spawningAgent.go();
136     }
137 
138     /**
139      * Stops the {@link SpawningAgent} in this node, waiting for active
                negotiations to finish before
140      * doing so
141      */
142     public void stopTuCSoN()
143     {
```

```
144         try
145         {
146            while(!this.activeNegotiatingAgents.isEmpty())
147            {
148               Thread.sleep(5000);
149            }
150            this.tucsonService = null;
151            this.spawningAgent.terminateAgent();
152            this.spawningAgent = null;
153         }
154         catch(InterruptedException e)
155         {
156            ErrorPrinter.printLog("Node", "stopTuCSoN", ErrorPrinter.
                  SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
157         }
158      }
159
160      /**
161       * Removes a subscription to a Context Broker.
162       *
163       * <p>This method is called once a negotiation finishes to shutdown the
             notification servers</p>
164       * @param caseName Name of the {@code Context Element} the subscription was
               made to
165       * @param cb Context Broker instance where the subscription was set
166       */
167      public void removeSubscription(String caseName, ContextBroker cb)
168      {
169         Iterator<Subscription> itSub = cb.getActiveSubscriptions().values().
               iterator();
170         while(itSub.hasNext())
171         {
172            Subscription sub = itSub.next();
173            if(sub.getContextElement().getID().equals(caseName))
174            {
175               cb.unsubscribeContext(sub.getID());
176               break;
177            }
178         }
179         if(cb.getActiveSubscriptions().size() == 0)
180         {
181            if(this.notifyServers.get(cb) != null)
182            {
183               this.notifyServers.get(cb).stopServer();
184               this.notifyServers.remove(cb);
185            }
186         }
187      }
188
189      /**
190       * Handles a notification received from the context broker. It commits
             changes to the case aggregation's description
191       * and redoes the subscription if it detects that nodes have been added or
             removed. If it detects that a node has
```

```
192      * changed his published data, it will call {@link #processChanges(HashMap,
             Subscription)} to query the nodes.
193      *
194      * @param notification JSON-formatted string containing the notification
             from the context broker
195      * @param subscription {@link Subscription} object that the notification is
             referring to
196      * @param broker {@link ContextBroker} instance representing the Context
             Broker that sent the notification in order
197      * to redo the subscription using this object if necessary
198      */
199     public synchronized void notifyNode(String notification, Subscription
            subscription, ContextBroker broker)
200     {
201        JSONParser parser = new JSONParser();
202        JSONArray attrArray;
203        try
204        {
205           attrArray = (JSONArray)((JSONObject)((JSONObject)((JSONArray)((
                  JSONObject)parser.parse(notification)).get("contextResponses")).
                  get(0)).get("contextElement")).get("attributes");
206           if(subscription.getNumNodes() == 0)
207           {
208              ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                     SEVERITY_INFO, "First notification received, setting
                     subscription...");
209              int numNodes = -1;
210              for(int i = 0; i < attrArray.size(); i++)
211              {
212                 JSONObject attr = (JSONObject)attrArray.get(i);
213                 if(((String)attr.get("name")).equals("numNodes"))
214                 {
215                    numNodes = Integer.parseInt(((String)attr.get("value")));
216                 }
217                 else if(((String)attr.get("name")).equals("description"))
218                 {
219
220                    subscription.setDescription((String)attr.get("value"));
221                 }
222                 else
223                 {
224                    if(!((String)attr.get("type")).equals("node"))
225                    {
226                       //Unknown attribute
227                       ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                              SEVERITY_WARNING, "Unknown attribute found in
                              notification");
228                       continue;
229                    }
230                    String name = (String)attr.get("name");
231                    Attribute nodeAttr = new NodeAttribute(name,"node");
232                    try
233                    {
```

```
234                     HashMap<String,String> values = ((NodeAttribute)nodeAttr)
                            .parseValues((String)attr.get("value"));
235                     subscription.addNode(new Node(name,values.get("address"))
                            , Integer.parseInt(values.get("numcases")),values.get(
                            "timestamp"));
236                 }
237                 catch(java.text.ParseException exc)
238                 {
239                     //Error is already printed in parseValues if exception
                            occurs
240                 }
241             }
242         }
243         if(numNodes != subscription.getNumNodes())
244         {
245             ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                    SEVERITY_WARNING, "Number of nodes in notification and
                    number of registered nodes differ");
246         }
247     }
248     else
249     {
250         HashMap<Node,HashMap<String,String>> changedNodes = new HashMap<
                Node,HashMap<String,String>>();
251         for(int i = 0; i < attrArray.size(); i++)
252         {
253             JSONObject attr = (JSONObject)attrArray.get(i);
254             if(((String)attr.get("name")).equals("numNodes"))
255             {
256                 if(Integer.parseInt(((String)attr.get("value"))) !=
                        subscription.getNumNodes())
257                 {
258                     /*
259                      * Imanol Barba:
260                      * Here we assume that if the number of nodes changes,
                            redoing the whole subscription
261                      * is prioritary rather than checking for node or
                            description changes because no other nodes
262                      * besides the one being added/removed will be modified
                            upon adding/removing a node and if
263                      * we redo the subscription, the new description will get
                             added either way. That's why we stop
264                      * checking the rest of the attributes and break the loop
                            .
265                      *
266                      * This holds true UNLESS someone deliberately and
                            externally messes up with the context broker.
267                      */
268                     if(!changedNodes.isEmpty())
269                     {
270                         ErrorPrinter.printLog("Node", "notifyNode",
                                ErrorPrinter.SEVERITY_WARNING, "Besides the number
                                of nodes, " + changedNodes.size() + " nodes were
                                found as having been modified");
```

```
271                         }
272                         ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                                SEVERITY_INFO, "numNodes modified. Redoing
                                subscription...");
273                         broker.unsubscribeContext(subscription.getID());
274                         broker.subscribeContext(subscription.getContextElement().
                                getID(), subscription.getResponseURL());
275                         return;
276                      }
277                   }
278                   else if(((String)attr.get("name")).equals("description"))
279                   {
280                      String description = (String)attr.get("value");
281                      if(!description.equals(subscription.getDescription()))
282                      {
283                         /*
284                          * Imanol Barba:
285                          * The description only gets changed whenever a context
                                element is created or a separate operation
286                          * with the sole objective of changing the description
                                and no other nodes occurs. Thus, if we detect
287                          * a change in the description, no other fields could've
                                been modified. That's why we stop checking
288                          * the rest of the attributes and break the loop.
289                          *
290                          * This holds true UNLESS someone deliberately and
                                externally messes up with the context broker.
291                          */
292                         if(!changedNodes.isEmpty())
293                         {
294                            ErrorPrinter.printLog("Node", "notifyNode",
                                   ErrorPrinter.SEVERITY_WARNING, "Besides the case
                                   description, " + changedNodes.size() + " nodes were
                                    found as having been modified");
295                         }
296                         ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                                SEVERITY_INFO, "description modified. Changing
                                description...");
297                         subscription.setDescription(description);
298                         return;
299                      }
300                   }
301                   else
302                   {
303                      if(!((String)attr.get("type")).equals("node"))
304                      {
305                         //Unknown attribute
306                         ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                                SEVERITY_WARNING, "Unknown attribute found in
                                notification");
307                         continue;
308                      }
309                      String name = (String)attr.get("name");
310                      Attribute nodeAttr = new NodeAttribute(name,"node");
```

```
311                     try
312                     {
313                         HashMap<String,String> values = ((NodeAttribute)nodeAttr)
                                .parseValues((String)attr.get("value"));
314                         Node node = new Node(name,values.get("address"));
315                         if(subscription.isModifiedNode(node, Integer.parseInt(
                                values.get("numcases")),values.get("timestamp")))
316                         {
317                             changedNodes.put(node,values);
318                         }
319                     }
320                     catch(java.text.ParseException exc)
321                     {
322                         //Error is already printed in parseValues if exception
                                occurs
323                     }
324                 }
325             }
326             processChanges(changedNodes,subscription);
327         }
328     }
329     catch(org.json.simple.parser.ParseException exc)
330     {
331         ErrorPrinter.printLog("Node", "notifyNode", ErrorPrinter.
                SEVERITY_ERROR, exc.getMessage());
332     }
333 }
334
335 /**
336  * This method is responsible for updating the case aggregation from a node
        we are subscribed to when it notifies us
337  * that it has been modified.
338  * @param changedNodes List of nodes that notified changes in their case
        collection and their value published in
339  * the context broker
340  * @param subscription {@link Subscription} object representing the
        subscription whose changes we are processing. This
341  * is used as a way to consult the list of subscribed nodes from the
        current subscription we are processing previous
342  * to any modifications
343  */
344 private void processChanges(HashMap<Node,HashMap<String,String>>
        changedNodes,Subscription subscription)
345 {
346     if(changedNodes.size() == 0)
347     {
348         ErrorPrinter.printLog("processChanges", "Node", ErrorPrinter.
                SEVERITY_INFO, "Received a notification with no changes");
349         return;
350     }
351     Iterator<Node> itNode = changedNodes.keySet().iterator();
352     Iterator<HashMap<String,String>> itVal = changedNodes.values().iterator
            ();
353     while(itNode.hasNext())
```

```
354         {
355             Node changedNode = itNode.next();
356             Node subscriptionNode = subscription.getNodes().get(changedNode.
                    getName());
357             HashMap<String,String> changedValue = itVal.next();
358             if(!subscriptionNode.getAddress().equals(changedNode.getAddress()))
359             {
360                 //Address changed
361                 subscriptionNode.setAddress(changedNode.getAddress());
362                 ErrorPrinter.printLog("Node", "processChanges", ErrorPrinter.
                        SEVERITY_INFO, "Node " + changedNode.getName() + "changed his
                        address to " + changedNode.getAddress());
363             }
364             else
365             {
366                 System.out.println("Node " + changedNode.getName() + " at " +
                        changedNode.getAddress() + " changed");
367                 subscription.updateNode(subscriptionNode.getName(),Integer.
                        parseInt(changedValue.get("numcases")) ,changedValue.get("
                        timestamp"));
368                 HashSet<NegotiatingAgent> activeAgents = this.
                        activeNegotiatingAgents.get(subscription.getContextElement().
                        getID());
369                 if(activeAgents != null)
370                 {
371                     if(activeAgents.isEmpty())
372                     {
373                         /* Imanol Barba
374                          *
375                          * Here we should notify the user through the interface, so
                                he decides to start a negotiation
376                          * or not and specifying parameters and keywords.
377                          */
378                     }
379                     else
380                     {
381                         Iterator<NegotiatingAgent> itAgent = this.
                                activeNegotiatingAgents.get(subscription.
                                getContextElement().getID()).iterator();
382                         while(itAgent.hasNext())
383                         {
384                             try
385                             {
386                                 itAgent.next().addPeerToActiveNegotiation(changedNode)
                                        ;
387                             }
388                             catch (InvalidLogicTupleException
389                                    | TucsonInvalidTupleCentreIdException
390                                    | TucsonOperationNotPossibleException
391                                    | UnreachableNodeException
392                                    | OperationTimeOutException
393                                    | InterruptedException e)
394                             {
```

```
395                         ErrorPrinter.printLog("Node", "processChanges",
                                ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " +
                                e.getMessage());
396                     }
397                 }
398             }
399         }
400     }
401   }
402 }
403
404 /**
405  * Adds an aggregation name to the set of published aggregations
406  *
407  * @param name Name of the aggregation that was published
408  */
409 public void addPublishedAggregation(String name)
410 {
411   this.publishedAggregations.add(name);
412 }
413
414 /**
415  * Removes an aggregation name to the set of published aggregations
416  *
417  * @param name Name of the aggregation that was unpublished
418  */
419 public void removePublishedAggregation(String name)
420 {
421   this.publishedAggregations.remove(name);
422 }
423
424 /**
425  * Adds a case aggregation to the node's database
426  * @param aggregation {@link CaseAggregation} object containing the
       aggregation to add
427  * @return {@code true} if the operation was successful, {@code false}
       otherwise
428  */
429 public boolean addAggregation(CaseAggregation aggregation)
430 {
431   boolean success = true;
432   startAtomicOperation();
433   try
434   {
435     this.provisioner.writeCaseAggregation(aggregation);
436   }
437   catch (Exception e)
438   {
439     ErrorPrinter.printLog("Node", "addAggregation", ErrorPrinter.
          SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
440     success = false;
441   }
442   stopAtomicOperation();
443   return success;
```

```
444     }
445
446     /**
447      * Retrieves a case aggregation from the node's database
448      * @param type Type of the case collection to retrieve
449      * @return {@link CaseAggregation} object being requested, {@code null} if
           it doesn't exist
450      */
451     public CaseAggregation getAggregation(String type)
452     {
453        startAtomicOperation();
454        CaseAggregation aggr = null;
455        try
456        {
457           aggr = this.provisioner.readCaseAggregation(type);
458        }
459        catch (Exception e)
460        {
461           ErrorPrinter.printLog("Node", "getAggregation", ErrorPrinter.
                SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
462        }
463        stopAtomicOperation();
464        return aggr;
465     }
466
467     /**
468      * Removes a case aggregation from the node's database
469      * @param name {@link CaseAggregation} name of the case aggregation to
           delete
470      * @return {@code true} if the operation was successful, {@code false}
           otherwise
471      */
472     public boolean removeAggregation(String name)
473     {
474        boolean success = true;
475        startAtomicOperation();
476        try
477        {
478           this.provisioner.removeCaseAggregation(name);
479        }
480        catch (Exception e)
481        {
482           ErrorPrinter.printLog("Node", "removeAggregation", ErrorPrinter.
                SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
483           success = false;
484        }
485        stopAtomicOperation();
486        return success;
487     }
488
489     /**
490      * Adds a case collection to their respective case aggregations in the node
491      * @param collection {@link CaseCollection} object containing the case
           collection to add
```

```
492      * @return {@code true} if the operation was successful, {@code false}
             otherwise
493      */
494     public boolean addCollection(CaseCollection collection)
495     {
496        boolean success = true;
497        startAtomicOperation();
498        try
499        {
500           this.provisioner.writeCaseCollection(collection);
501        }
502        catch (Exception e)
503        {
504           ErrorPrinter.printLog("Node", "addCollection", ErrorPrinter.
                SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
505           success = false;
506        }
507        stopAtomicOperation();
508        return success;
509     }
510
511     /**
512      * Retrieves a case collection from the node's database
513      * @param collectionID ID of the case collection to retrieve
514      * @return {@link CaseCollection} object being requested, {@code null} if
             it doesn't exist
515      */
516     public CaseCollection getCollection(String collectionID)
517     {
518        startAtomicOperation();
519        CaseCollection collection = null;
520        try
521        {
522           collection = this.provisioner.readCaseCollection(collectionID);
523        }
524        catch (Exception e)
525        {
526           ErrorPrinter.printLog("Node", "getCollection", ErrorPrinter.
                SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
527        }
528        stopAtomicOperation();
529        return collection;
530     }
531
532     /**
533      * Removes a case collection from the node's database
534      * @param collectionID {@link CaseAggregation} ID of the case collection to
             delete
535      * @return {@code true} if the operation was successful, {@code false}
             otherwise
536      */
537     public boolean removeCollection(String collectionID)
538     {
539        boolean success = true;
```

```
540         startAtomicOperation();
541         try
542         {
543            this.provisioner.removeCaseCollection(collectionID);
544         }
545         catch (Exception e)
546         {
547            ErrorPrinter.printLog("Node", "removeCollection", ErrorPrinter.
                SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
548            success = false;
549         }
550         stopAtomicOperation();
551         return success;
552      }
553
554      /**
555       * Searches in the database for case collections matching a certain
             criteria. If partial matches are found, the cases
556       * that don't satisfy the query will be excluded from the results
557       *
558       * @param caseName Type of case aggregation in which to search
559       * @param keywords Keywords that the resulting case collections must
             contain
560       * @param attributes Attributes that the resulting case collections must
             contain and match their values
561       * @return {@code Set} with all the case collections that satisfy the query
562       */
563      public HashSet<CaseCollection> searchCollections(String caseName, HashSet<
             String> keywords, HashMap<String,CaseAttribute> attributes)
564      {
565         HashSet<CaseCollection> matchingCollections = null;
566         HashSet<CaseCollection> wantedCollections = new HashSet<CaseCollection
                >();
567         try
568         {
569            matchingCollections = this.provisioner.queryDatabase(caseName,
                keywords, attributes.keySet());
570            Iterator<CaseCollection> itCol = matchingCollections.iterator();
571            while(itCol.hasNext())
572            {
573               boolean isWanted = true;
574               CaseCollection collection = itCol.next();
575               Iterator<String> itAttr = attributes.keySet().iterator();
576               while(itAttr.hasNext())
577               {
578                  String attrName = itAttr.next();
579                  CaseAttribute requestedAttr = attributes.get(attrName);
580                  CaseAttribute colAttr = collection.getAttribute(attrName);
581                  if(!requestedAttr.equals(colAttr))
582                  {
583                     if(requestedAttr.getClass() != colAttr.getClass())
584                     {
585                        isWanted = false;
586                        break;
```

```
587                 }
588                 if(requestedAttr.getClass() == NumericalAttribute.class)
589                 {
590                    NumericalAttribute attr1 = (NumericalAttribute)
                          requestedAttr;
591                    NumericalAttribute attr2 = (NumericalAttribute) colAttr;
592                    //Is subset
593                    if((attr1.getMin() <= attr2.getMin()) && (attr1.getMax()
                          >= attr2.getMax()))
594                    {
595                       //Nothing to do, we're happy with this
596                    }
597                    //Disjoint sets
598                    else if((attr1.getMin() > attr2.getMax()) || (attr1.
                          getMax() < attr2.getMin()))
599                    {
600                       isWanted = false;
601                       break;
602                    }
603                    else
604                    {
605                       HashSet<Case> unwantedCases = new HashSet<Case>();
606                       Iterator<Case> itCase = collection.getCases();
607                       while(itCase.hasNext())
608                       {
609                          Case c = itCase.next();
610                          NumericalAttribute attr3 = (NumericalAttribute) c.
                                getAttribute(attrName);
611                          if(!((attr1.getMin() <= attr3.getMin()) && (attr1.
                                getMax() >= attr3.getMax())))
612                          {
613                             unwantedCases.add(c);
614                          }
615                       }
616                       Iterator<Case> itUnwanted = unwantedCases.iterator();
617                       while(itUnwanted.hasNext())
618                       {
619                          collection.removeCase(itUnwanted.next().getID());
620                       }
621                    }
622                 }
623                 else
624                 {
625                    if(!requestedAttr.getValue().equals("*"))
626                    {
627                       if(colAttr.getValue().equals("*"))
628                       {
629                          HashSet<Case> unwantedCases = new HashSet<Case>();
630                          Iterator<Case> itCase = collection.getCases();
631                          while(itCase.hasNext())
632                          {
633                             Case c = itCase.next();
634                             if(!c.getAttribute(attrName).getValue().equals(
                                   requestedAttr.getValue()))
```

```
635                                 {
636                                     unwantedCases.add(c);
637                                 }
638                             }
639                             Iterator<Case> itUnwanted = unwantedCases.iterator
                                    ();
640                             while(itUnwanted.hasNext())
641                             {
642                                 collection.removeCase(itUnwanted.next().getID())
                                        ;
643                             }
644                         }
645                         else if(!requestedAttr.getValue().equals(colAttr.
                                getValue()))
646                         {
647                             isWanted = false;
648                             break;
649                         }
650                     }
651                 }
652             }
653         }
654         if(isWanted)
655         {
656             wantedCollections.add(collection);
657         }
658     }
659   }
660   catch(Exception e)
661   {
662     ErrorPrinter.printLog("Node", "searchCollections", ErrorPrinter.
          SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
663   }
664   return wantedCollections;
665 }
666
667 /**
668  * Starts a {@link NegotiatingAgent} as host to begin a negotiation process
669  *
670  * @param caseType Type of cases that will be negotiated
671  * @param cb Context Broker instance where to look for peers
672  * @param attributes Attributes wanted for the cases
673  * @param keywords Keywords wanted for the cases
674  * @param numCases Number of cases needed
675  *
676  * @throws Exception If the TuCSoN service hasn't been started in this node
        instance
677  */
678 public void startNegotiation(String caseType, ContextBroker cb, HashSet<
      CaseAttribute> attributes, HashSet<String> keywords, int numCases,
      String tree, int K) throws Exception
679 {
680   if(this.tucsonService == null)
681   {
```

```
682             throw new Exception("TuCSoN not started in this node: " + this.
                    getName());
683         }
684         try
685         {
686             NegotiatingAgent newAgent = new NegotiatingAgent(this.tucsonService,
                    caseType,true);
687             newAgent.setNumCases(numCases);
688             if(this.notifyServers.get(cb) == null)
689             {
690                 NotificationServer notifySrv = new NotificationServer(cb,8000 +
                        this.notifyServers.size());
691                 notifySrv.start();
692                 this.notifyServers.put(cb, notifySrv);
693             }
694             cb.subscribeContext(caseType, "http://" + this.getAddress() + ":" +
                    notifyServers.get(cb).getPort());
695             HashMap<Node,Integer> nodeList = cb.searchRecords(caseType);
696             Iterator<Node> itPeers = nodeList.keySet().iterator();
697             Iterator<CaseAttribute> itAttr = attributes.iterator();
698             Iterator<String> itKw = keywords.iterator();
699             while(itPeers.hasNext())
700             {
701                 Node peer = itPeers.next();
702                 if(!peer.getName().equals(this.name))
703                 {
704                     newAgent.addPeer(peer);
705                 }
706             }
707             while(itAttr.hasNext())
708             {
709                 newAgent.addAttribute(itAttr.next());
710             }
711             while(itKw.hasNext())
712             {
713                 newAgent.addKeyword(itKw.next());
714             }
715             newAgent.setTree(tree);
716             newAgent.setK(K);
717             this.addNegotiatingAgent(newAgent, caseType);
718             this.activeSubscriptions.put(newAgent, cb);
719             newAgent.go();
720         }
721         catch (TucsonInvalidAgentIdException e)
722         {
723             ErrorPrinter.printLog("Node", "startNegotiation", ErrorPrinter.
                    SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
724         }
725     }
726
727     /**
728      * Adds an active host negotiating agent to this node
729      *
730      * @param agent {@link NegotiatingAgent} to be added
```

```
731      * @param caseType Type of cases that the agent is negotiating
732      */
733     private void addNegotiatingAgent(NegotiatingAgent agent, String caseType)
734     {
735        HashSet<NegotiatingAgent> agentSet = this.activeNegotiatingAgents.get(
              caseType);
736        if(agentSet == null)
737        {
738           this.activeNegotiatingAgents.put(caseType, new HashSet<
                 NegotiatingAgent>());
739        }
740        this.activeNegotiatingAgents.get(caseType).add(agent);
741     }
742
743     /**
744      * Removes an active host negotiating agent from this node
745      *
746      * @param agent {@link NegotiatingAgent} to be removed
747      * @param caseType Type of cases that the agent was negotiating
748      */
749     public void finishedNegotiation(NegotiatingAgent agent, String caseType)
750     {
751        this.activeNegotiatingAgents.get(caseType).remove(agent);
752        if(this.activeNegotiatingAgents.get(caseType).isEmpty())
753        {
754           this.activeNegotiatingAgents.remove(caseType);
755           ContextBroker cb = this.activeSubscriptions.get(agent);
756           this.activeSubscriptions.remove(agent);
757           this.removeSubscription(caseType, cb);
758        }
759     }
760
761     /**
762      * Creates a representation of a case aggregation in the node
763      *
764      * <p>This is used to perform operations (publish, unpublish) in the Orion
           Context Broker with a certain
765      * case aggregation</p>
766      * @param aggregationTypes {@code Set} of case aggregation names to include
           in the resulting information
767      * @return {@code Map} describing each case aggregation and the number of
           cases the aggregation holds
768      */
769     public synchronized HashMap<String, Integer> getCollectionInfo(HashSet<
           String> aggregationTypes)
770     {
771        HashMap<String,Integer> info = new HashMap<String,Integer>();
772        Iterator<String> it = aggregationTypes.iterator();
773        while(it.hasNext())
774        {
775           String type = it.next();
776           int numCases = getNumCases(type);
777           if(numCases != 0)
778           {
```

```
779             info.put(type, numCases);
780          }
781       }
782       return info;
783    }
784
785    /**
786     * Method to obtain the number of cases a certain case aggregation holds
787     * @param type Type of the case aggregation
788     * @return Number of cases the case aggregation holds
789     */
790    public int getNumCases(String type)
791    {
792       startAtomicOperation();
793       int num = 0;
794       CaseAggregation aggregation = this.getAggregation(type);
795       if(aggregation != null)
796       {
797          num = aggregation.getNumCases();
798       }
799       stopAtomicOperation();
800       return num;
801    }
802
803    /**
804     * Sets the case provisioner for the current node
805     *
806     * @param provisioner {@link CaseProvisioner} implementation that will be
          used to fetch and store medical case and/or
807     * medical case information in this node
808     */
809    public void setProvisioner(CaseProvisioner provisioner)
810    {
811       this.provisioner = provisioner;
812    }
813
814    /**
815     * Changes the node address and republishes its collections to the context
          broker so the change is notified to
816     * all possible subscribers
817     * @param newAddress New node address
818     * @param cb Context Broker instance to republish the collections
819     *
820     * @return {@code true} if the operation is successful, {@code false}
          otherwise
821     */
822    public boolean changeAddress(String newAddress, ContextBroker cb)
823    {
824       startAtomicOperation();
825       setAddress(newAddress);
826       boolean success = cb.publishRecords(getCollectionInfo(this.
             publishedAggregations));
827       stopAtomicOperation();
828       return success;
```

```
829     }
830
831     /**
832      * {@link #name} getter.
833      * @return Name of the node
834      */
835     public synchronized String getName()
836     {
837        return name;
838     }
839
840     /**
841      * {@link #address} getter.
842      * @return IP address of the node
843      */
844     public String getAddress()
845     {
846        startAtomicOperation();
847        String addr = new String(this.address);
848        stopAtomicOperation();
849        return addr;
850     }
851
852     /**
853      * {@link #address} setter.
854      * @param address New address of the node
855      */
856     private void setAddress(String address)
857     {
858        startAtomicOperation();
859        this.address = address;
860        stopAtomicOperation();
861     }
862
863     /**
864      * Method to obtain the last modification date of a case aggregation
865      * @param type Type of the case aggregation
866      * @return Timestamp with the last modification date
867      */
868     public synchronized String getLastUpdate(String type)
869     {
870        CaseAggregation aggregation;
871        try
872        {
873           aggregation = this.provisioner.readCaseAggregation(type);
874           if(aggregation != null)
875           {
876              return aggregation.getTimestamp();
877           }
878        }
879        catch (Exception e)
880        {
881           ErrorPrinter.printLog("Node", "getLastUpdate", ErrorPrinter.
               SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
```

```
882         }
883         return null;
884     }
885 }
```

NumericalAttribute.java

```java
package ch.hevs.ISyPeM2.Node;

/**
 * This class represents a {@link CaseAttribute} whose value is numerical, and is represented as a range of maximum value
 * and minimum value
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class NumericalAttribute extends CaseAttribute
{
   /**
    * Minimum value for this attribute
    */
   private double min;

   /**
    * Maximum value for this attribute
    */
   private double max;

   /**
    * Class constructor
    * @param name Name of the attribute
    * @param min Minimum value
    * @param max Maximum value
    */
   public NumericalAttribute(String name, double min, double max)
   {
      super(name);
      this.min = min;
      this.max = max;
   }

   /**
    * Returns a string representation of the attribute's value as in "[min,max]"
    * @return String representation of the attribute's value
    */
   @Override
   public String getValue()
   {
      return "[" + Double.toString(min) + "," + Double.toString(max) + "]";
   }

   /**
    * Returns the minimum value of this attribute's numerical range
    * @return The minimum value of the range
    */
   public double getMin()
   {
      return this.min;
```

```
52      }
53
54      /**
55       * Returns the maximum value of this attribute's numerical range
56       * @return The maximum value of the range
57       */
58      public double getMax()
59      {
60         return this.max;
61      }
62      //TODO: Fix this
63      /**
64       * Transforms the range (min, max) to a value if the min=max (non-
             generaliued value)
65       * @return the value corresponding to actual (non-generalized) number
66       */
67      public double getActualValue()
68      {
69         if (this.min==this.max){
70            return this.max;
71         }
72         else{System.out.println("it has been already generalized!");}
73         return -1;
74      }
75
76      /*public void asd()
77      {
78         CaseAttribute attr = whatever;
79         if(attr instanceof NumericalAttribute)
80         {
81            NumericalAttribute numAttr = (NumericalAttribute)attr;
82            if(numAttr.getMin() == numAttr.getMax())
83            {
84               //not generalised
85            }
86         }
87      }*/
88  }
```

RSDBCollection.java

```
1  package ch.hevs.ISyPeM2.Node;
2
3  import java.util.Iterator;
4
5  /**
6   * This class represents a Generalized case collection
7   * @author Imanol-Mikel Barba Sabariego
8   *
9   */
10 public class RSDBCollection extends CaseCollection
11 {
12    /* Imanol Barba
13     *
14     * ATM, this will be a String until the tree parsing algorithm is clear
15     *
16     * Alevtina: regardless from what we parse (now-multiple files, later-one
          file )there is a class AttributeTree that represents a single tree
17     *       and TreeCollection that stores all the trees for an instance of
          RSDB (we can discuss tmr)
18     *
19     */
20    /**
21     * Generalization tree for the parameters
22     */
23    private String tree;
24    /**
25     * K parameter from anonymization
26     */
27    private int K;
28
29    /**
30     * Class constructors
31     * @param collection
32     * @param tree
33     * @param K
34     */
35    protected RSDBCollection(CaseCollection collection,String tree, int K)
36    {
37       super(collection.getID(),collection.getType(),collection.getDescription
             (),collection.getOriginNode());
38       this.tree = tree;
39       this.K = K;
40       //this.convert();
41    }
42
43    /*private void convert()
44    {
45       Iterator<Case> itCase = this.getCases();
46       while(itCase.hasNext())
47       {
48          Case c = itCase.next();
49          Iterator<CaseAttribute> itAttr = c.getAttributes();
50          while(itAttr.hasNext())
```

```
51            {
52               CaseAttribute attr = itAttr.next();
53               if(!attr.getName().startsWith("G_"))
54               {
55                  c.removeAttribute(attr.getName());
56                  CaseAttribute G_attr = CaseAttribute.generalizeAttribute(attr,
                       this.tree, this.K);
57                  c.setAttribute(G_attr);
58               }
59            }
60         }
61
62      }*/
63
64
65
66   }
```

D.6 ch.hevs.ISyPeM2.NotificationServer package

NotificationHandler.java

```
1  package ch.hevs.ISyPeM2.NotificationServer;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5
6  import javax.servlet.ServletException;
7  import javax.servlet.http.HttpServletRequest;
8  import javax.servlet.http.HttpServletResponse;
9
10 import org.eclipse.jetty.server.Request;
11 import org.eclipse.jetty.server.handler.AbstractHandler;
12
13 /**
14  * This class is a handler for requests directed to the embedded HTTP server.
15  *
16  * @see NotificationServer
17  * @author Imanol-Mikel Barba Sabariego
18  *
19  */
20 public class NotificationHandler extends AbstractHandler
21 {
22    /**
23     * {@code NotificationServer} whose requests we are handling
24     */
25    private NotificationServer notifyServ;
26
27    /**
28     * Class constructor
29     * @param notifyServ {@code NotificationServer} whose requests we are
          handling
30     */
31    public NotificationHandler(NotificationServer notifyServ)
32    {
33       this.notifyServ = notifyServ;
34    }
35
36    /**
37     * This method gets called every time the Jetty embedded server receives a
          request. If the request is done using the POST
38     * method, addressed to the "/notify" URL path and the Content-Type header
          is set to "application/json", it will forward
39     * the body of the request to the {@code NotificationServer} instance,
          which will in turn send it back to the {@code
40     * ContextBroker} instance and finally to the node this notification is
          addressed to
41     */
42    public void handle(String target,Request baseRequest,HttpServletRequest
          request,HttpServletResponse response) throws IOException,
          ServletException
43    {
```

```
44          response.setContentType("text/html;charset=utf-8");
45          baseRequest.setHandled(true);
46          if(request.getMethod().equals("POST"))
47          {
48             if(target.equals("/notify"))
49             {
50                if(request.getContentType() != null)
51                {
52                   if(request.getContentType().equals("application/json"))
53                   {
54                      response.setStatus(HttpServletResponse.SC_OK);
55                      String requestBody = "";
56                      BufferedReader in = request.getReader();
57                      String line;
58                      while((line = in.readLine()) != null)
59                      {
60                         requestBody += line;
61                      }
62                      org.eclipse.jetty.util.log.Log.getLog().info("Received POST
                           request to " + target + ": " + requestBody);
63                      /*
64                       * Imanol Barba:
65                       * If the request body is empty, we reply with code 200 and
                            an empty body. This is used
66                       * for testing to check that the Jetty server is working
                            correctly
67                       */
68                      if(requestBody.isEmpty())
69                      {
70                         return;
71                      }
72                      else
73                      {
74                         notifyServ.pushResponse(requestBody);
75                      }
76                   }
77                   else
78                   {
79                      response.setStatus(HttpServletResponse.SC_NOT_ACCEPTABLE);
80                      response.setHeader("Accept", "application/json");
81                   }
82                }
83                else
84                {
85                   response.setStatus(HttpServletResponse.SC_NOT_ACCEPTABLE);
86                   response.setHeader("Accept", "application/json");
87                }
88             }
89             else
90             {
91                response.setStatus(HttpServletResponse.SC_FORBIDDEN);
92             }
93          }
94          else
```

```
95          {
96              response.setStatus(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
97          }
98      }
99  }
```

NotificationServer.java

```java
package ch.hevs.ISyPeM2.NotificationServer;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.ServerConnector;

import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
import ch.hevs.ISyPeM2.Core.ErrorPrinter;

/**
 * This class provides the node running the code with a minimal HTTP server using Jetty.
 *
 * <p>It runs on a separate thread and by default it only accepts a single connection at a time and can
 * enqueue up to 10 incoming connections</p>
 *
 * @see <a href="http://eclipse.org/jetty/">Jetty project page</a>
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class NotificationServer extends Thread
{
   /**
    * {@code ContextBroker} instance that originates the requests and thus, holds the {@code Node} object
    * which the notifications should be sent back to.
    */
   private ContextBroker context;

   /**
    * Jetty {@code Server} object that manages the incoming HTTP requests
    */
   private Server server;

   /**
    * TCP port the server is listening in
    */
   private int port;

   /**
    * Class constructor
    * @param context {@code ContextBroker} instance launched by the local node to which we are going to forward requests to
    * @param port TCP port the Jetty embedded HTTP server is going to listen for incoming requests
    */
   public NotificationServer(ContextBroker context, int port)
   {
         this.context = context;
         this.port = port;
         this.server = new Server(port);
   }

```

```
49      /**
50       * Checks if the Jetty server has started
51       * @return true if the server has started, false otherwise
52       */
53      public boolean isStarted()
54      {
55         return this.server.isStarted();
56      }
57
58      /**
59       * Thread start method. This will launch the Jetty embedded server and will
             end as the server does
60       */
61      public void run()
62      {
63         try
64         {
65            ServerConnector servConn = (ServerConnector)server.getConnectors()
                 [0];
66            servConn.setAcceptQueueSize(10);
67            servConn.setIdleTimeout(30000);
68            server.setHandler(new NotificationHandler(this));
69            server.start();
70            server.join();
71         }
72         catch (Exception e)
73         {
74            ErrorPrinter.printLog("NotificationServer", "run", ErrorPrinter.
                 SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
75         }
76      }
77
78      /**
79       * Forwards a notification from the {@link NotificationHandler} to the {
            @code ContextBroker} instance
80       * @param response JSON-encoded string containing the notification from the
             context broker
81       */
82      protected synchronized void pushResponse(String response)
83      {
84         context.processNotification(response);
85      }
86
87      /**
88       * Stops the currently running Jetty server. This will also make the {@code
             run()} method end and consequently
89       * end the thread the {@code NotificationServer} is running in
90       */
91      public void stopServer()
92      {
93         try
94         {
95            server.stop();
96         }
```

```
97          catch(Exception exc)
98          {
99             ErrorPrinter.printLog("NotificationServer", "stopServer",
                  ErrorPrinter.SEVERITY_ERROR, exc.getMessage());
100         }
101      }
102
103      /**
104       * {@link #port} getter.
105       * @return TCP port the current server is listening to
106       */
107      public int getPort()
108      {
109         return port;
110      }
111
112   }
```

D.7 ch.hevs.ISyPeM2.Tests package

AgentQueryDatabase.java

```
1  package ch.hevs.ISyPeM2.Tests;
2  import java.io.BufferedInputStream;
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.sql.Connection;
6  import java.sql.DriverManager;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9  import java.util.HashSet;
10 import java.util.Scanner;
11
12 import org.kohsuke.args4j.CmdLineException;
13 import org.kohsuke.args4j.CmdLineParser;
14 import org.kohsuke.args4j.Option;
15
16 import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
17 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
18 import ch.hevs.ISyPeM2.DB.MySQLCaseProvisioner;
19 import ch.hevs.ISyPeM2.Node.CaseAttribute;
20 import ch.hevs.ISyPeM2.Node.Node;
21 import ch.hevs.ISyPeM2.Node.NumericalAttribute;
22
23
24 /**
25  * This class is a runnable test to perform a test negotiation between 3
        agents
26  * @author Imanol-Mikel Barba Sabariego
27  *
28  */
29 public class AgentQueryDatabase
30 {
31    @Option(name = "-v", aliases = { "--verbose" }, required = false, usage = "
          Increase verbosity to show INFO messages")
32    private boolean verbose = false;
33
34    @Option(name = "-vv", aliases = { "--verbose-debug" }, required = false,
          usage = "Increase verbosity to show DEBUG messages")
35    private boolean debug = false;
36
37    @Option(name = "-r", aliases = { "--recreate" }, required = false, usage =
          "Recreate Nodes databases")
38    private boolean recreate = false;
39
40    private final static String MySQLHostname = "mysql.virtualbox";
41    private final static String MySQLUser = "root";
42    private final static String MySQLPass = "mysqladmin";
43
44    private final static String schemaFile = "ch/hevs/ISyPeM2/Files/
          negotiation_test.sql";
45
```

```
46     public static void importSQL(Connection conn, InputStream in) throws
           SQLException
47     {
48        Scanner s = new Scanner(in);
49        s.useDelimiter("(;(\r)?\n)|(--\n)");
50        Statement st = null;
51        try
52        {
53           st = conn.createStatement();
54           while (s.hasNext())
55           {
56              String line = s.next();
57              if (line.startsWith("/*!") && line.endsWith("*/"))
58              {
59                 int i = line.indexOf(' ');
60                 line = line.substring(i + 1, line.length() - " */".length());
61              }
62
63              if (line.trim().length() > 0)
64              {
65                 st.execute(line);
66              }
67           }
68        }
69        finally
70        {
71           if (st != null) st.close();
72        }
73        s.close();
74     }
75
76     public static void main(String[] args)
77     {
78        AgentQueryDatabase arguments = new AgentQueryDatabase();
79        CmdLineParser parser = new CmdLineParser(arguments);
80        try
81        {
82           parser.parseArgument(args);
83        }
84        catch(CmdLineException exc)
85        {
86           parser.setUsageWidth(Integer.MAX_VALUE);
87           parser.printUsage(System.err);
88           return;
89        }
90
91        if(arguments.verbose)
92        {
93           ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_INFO);
94           System.setProperty("org.eclipse.jetty.LEVEL","INFO");
95        }
96        if(arguments.debug)
97        {
98           ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_DEBUG);
```

```
99            System.setProperty("org.eclipse.jetty.LEVEL","DEBUG");
100       }
101       if(arguments.recreate)
102       {
103          try
104          {
105             Class.forName("com.mysql.jdbc.Driver");
106             Connection conn = DriverManager.getConnection("jdbc:mysql://" +
                   MySQLHostname + "/", MySQLUser, MySQLPass);
107             BufferedInputStream br = new BufferedInputStream(ClassLoader.
                   getSystemClassLoader().getResourceAsStream(schemaFile));
108             AgentQueryDatabase.importSQL(conn, br);
109             br.close();
110             conn.close();
111          }
112          catch (ClassNotFoundException e)
113          {
114             ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                   SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
115          }
116          catch (SQLException e)
117          {
118             ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                   SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
119          }
120          catch (IOException e)
121          {
122             ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                   SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
123          }
124       }
125       System.setProperty("org.eclipse.jetty.LEVEL","WARN");
126
127       System.out.println("Started at: " + System.currentTimeMillis());
128
129       try
130       {
131          Node n1 = new Node("Node 1", "tucson1.virtualbox");
132          Node n2 = new Node("Node 2", "tucson2.virtualbox");
133          Node n3 = new Node("Node 3", "tucson3.virtualbox");
134
135          n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname, MySQLUser,
                MySQLPass, "test1"));
136          n2.setProvisioner(new MySQLCaseProvisioner(MySQLHostname, MySQLUser,
                MySQLPass, "test2"));
137          n3.setProvisioner(new MySQLCaseProvisioner(MySQLHostname, MySQLUser,
                MySQLPass, "test3"));
138
139          n1.startTuCSoN();
140          n2.startTuCSoN();
141          n3.startTuCSoN();
142
143          HashSet<CaseAttribute> attributes = new HashSet<CaseAttribute>();
144          HashSet<String> keywords = new HashSet<String>();
```

```
145
146             CaseAttribute weight = new NumericalAttribute("BW",0,100000);
147             CaseAttribute age = new NumericalAttribute("GA",0,1000);
148
149             attributes.add(weight);
150             attributes.add(age);
151
152             keywords.add("gentamicin");
153             keywords.add("neonates");
154
155             ContextBroker cb1 = new ContextBroker("http://orion.virtualbox:1026/
                    NGSI10",n1);
156             ContextBroker cb2 = new ContextBroker("http://orion.virtualbox:1026/
                    NGSI10",n2);
157             ContextBroker cb3 = new ContextBroker("http://orion.virtualbox:1026/
                    NGSI10",n3);
158
159             HashSet<String> aggregationNames = new HashSet<String>();
160             aggregationNames.add("Gentamicin");
161
162             cb2.publishRecords(n2.getCollectionInfo(aggregationNames));
163             cb3.publishRecords(n3.getCollectionInfo(aggregationNames));
164
165             n1.startNegotiation("Gentamicin", cb1, attributes, keywords, 3,"
                    testTree",1);
166
167
168             //cb2.deleteRecords(aggregationNames);
169             //cb3.deleteRecords(aggregationNames);
170         }
171         catch(Exception e)
172         {
173             ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                    SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
174         }
175     }
176 }
```

CBPublishTest.java

```
1  package ch.hevs.ISyPeM2.Tests;
2
3  import static org.junit.Assert.assertTrue;
4
5  import java.util.HashSet;
6
7  import org.junit.Before;
8  import org.junit.Test;
9
10 import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
11 import ch.hevs.ISyPeM2.DB.LocalCaseProvisioner;
12 import ch.hevs.ISyPeM2.Node.Case;
13 import ch.hevs.ISyPeM2.Node.CaseAggregation;
14 import ch.hevs.ISyPeM2.Node.CaseCollection;
15 import ch.hevs.ISyPeM2.Node.Node;
16
17 /**
18  * This class is a JUnit test that checks if the code is able to publish a to
       publish some records to a certain
19  * instance of the Orion Context Broker
20  *
21  * @author Imanol-Mikel Barba Sabariego
22  *
23  */
24 public class CBPublishTest
25 {
26    final String contextBrokerURL = "http://orion.virtualbox:1026/NGSI10";
27
28    ContextBroker cb1;
29    ContextBroker cb2;
30    ContextBroker cb3;
31
32    Node n1;
33    Node n2;
34    Node n3;
35
36    HashSet<String> cn1;
37    HashSet<String> cn2;
38
39    @Before
40    public void setUp() throws Exception
41    {
42       n1 = new Node("Test Node 1","127.0.0.1");
43       n2 = new Node("Test Node 2","127.0.0.2");
44       n3 = new Node("Test Node 3","127.0.0.3");
45
46       n1.setProvisioner(new LocalCaseProvisioner());
47       n2.setProvisioner(new LocalCaseProvisioner());
48       n3.setProvisioner(new LocalCaseProvisioner());
49
50       cb1 = new ContextBroker(contextBrokerURL,n1);
51       cb2 = new ContextBroker(contextBrokerURL,n2);
52       cb3 = new ContextBroker(contextBrokerURL,n3);
```

```
53
54          CaseAggregation aggregationN1 = new CaseAggregation("Malaria", "People
                affected with malaria");
55          CaseAggregation aggregationN2 = new CaseAggregation("AIDS", "People
                affected with AIDS");
56          CaseAggregation aggregationN3 = new CaseAggregation("Malaria", "People
                affected with malaria");
57
58          CaseCollection collectionN1 = new CaseCollection("Malaria_ethnia_white",
                 "Malaria", "White people affected with malaria",n1);
59          CaseCollection collectionN2 = new CaseCollection("AIDS", "AIDS", "White
                people affected with malaria",n2);
60          CaseCollection collectionN3 = new CaseCollection("Malaria_ethnia_black",
                 "Malaria", "Black people affected with malaria",n3);
61
62          cn1 = new HashSet<String>();
63          cn2 = new HashSet<String>();
64
65          for(int i = 1; i <= 15; i++)
66          {
67             collectionN1.addCase(new Case("Case " + Integer.toString(i),"Malaria"
                   , "Malaria Case " + Integer.toString(i)));
68          }
69
70          for(int i = 1; i <= 45; i++)
71          {
72             collectionN2.addCase(new Case("Case " + Integer.toString(i),"AIDS", "
                   AIDS Case " + Integer.toString(i)));
73          }
74
75          for(int i = 1; i <= 20; i++)
76          {
77             collectionN3.addCase(new Case("Case " + Integer.toString(i),"Malaria"
                   , "Malaria Case " + Integer.toString(i)));
78          }
79
80          aggregationN1.addCollection(collectionN1);
81          aggregationN2.addCollection(collectionN2);
82          aggregationN3.addCollection(collectionN3);
83
84          cn1.add("Malaria");
85          cn2.add("AIDS");
86
87          assertTrue(n1.addAggregation(aggregationN1));
88          assertTrue(n2.addAggregation(aggregationN2));
89          assertTrue(n3.addAggregation(aggregationN3));
90       }
91
92       @Test
93       public void testPublish()
94       {
95          assertTrue(cb1.publishRecords(n1.getCollectionInfo(cn1)));
96          assertTrue(cb2.publishRecords(n2.getCollectionInfo(cn2)));
97          assertTrue(cb3.publishRecords(n3.getCollectionInfo(cn1)));
```

```
98
99          assertTrue(cb1.deleteRecords(cn1));
100         assertTrue(cb2.deleteRecords(cn2));
101         assertTrue(cb3.deleteRecords(cn1));
102     }
103 }
```

CBSearchTest.java

```
1  package ch.hevs.ISyPeM2.Tests;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertTrue;
5  import static org.junit.Assert.fail;
6
7  import java.util.HashMap;
8  import java.util.HashSet;
9  import java.util.Iterator;
10
11 import org.junit.Before;
12 import org.junit.Test;
13
14 import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
15 import ch.hevs.ISyPeM2.DB.LocalCaseProvisioner;
16 import ch.hevs.ISyPeM2.Node.Case;
17 import ch.hevs.ISyPeM2.Node.CaseAggregation;
18 import ch.hevs.ISyPeM2.Node.CaseCollection;
19 import ch.hevs.ISyPeM2.Node.Node;
20
21 /**
22  * This class is a JUnit test that checks if the code is able to publish a to
       publish some records to a certain
23  * instance of the Orion Context Broker and then search for published records
       within the context broker to see if they
24  * are visible by other nodes
25  *
26  * @author Imanol-Mikel Barba Sabariego
27  *
28  */
29 public class CBSearchTest
30 {
31    final String contextBrokerURL = "http://orion.virtualbox:1026/NGSI10";
32
33    ContextBroker cb1;
34    ContextBroker cb2;
35    ContextBroker cb3;
36
37    HashSet<Case> malariaSet1;
38    HashSet<Case> malariaSet2;
39    HashSet<Case> AIDSSet1;
40
41    HashSet<String> cn1;
42    HashSet<String> cn2;
43
44    @Before
45    public void setUp() throws Exception
46    {
47       Node n1 = new Node("Test Node 1","127.0.0.1");
48       Node n2 = new Node("Test Node 2","127.0.0.2");
49       Node n3 = new Node("Test Node 3","127.0.0.3");
50
51       n1.setProvisioner(new LocalCaseProvisioner());
```

```
52        n2.setProvisioner(new LocalCaseProvisioner());
53        n3.setProvisioner(new LocalCaseProvisioner());
54
55        cb1 = new ContextBroker(contextBrokerURL,n1);
56        cb2 = new ContextBroker(contextBrokerURL,n2);
57        cb3 = new ContextBroker(contextBrokerURL,n3);
58
59        CaseAggregation aggregationN1 = new CaseAggregation("Malaria", "People
              affected with malaria");
60        CaseAggregation aggregationN2 = new CaseAggregation("AIDS", "People
              affected with AIDS");
61        CaseAggregation aggregationN3 = new CaseAggregation("Malaria", "People
              affected with malaria");
62
63        CaseCollection collectionN1 = new CaseCollection("Malaria_ethnia_white",
               "Malaria", "White people affected with malaria",n1);
64        CaseCollection collectionN2 = new CaseCollection("AIDS", "AIDS", "White
              people affected with malaria",n2);
65        CaseCollection collectionN3 = new CaseCollection("Malaria_ethnia_black",
               "Malaria", "Black people affected with malaria",n3);
66
67        cn1 = new HashSet<String>();
68        cn2 = new HashSet<String>();
69
70        for(int i = 1; i <= 15; i++)
71        {
72           collectionN1.addCase(new Case("Case " + Integer.toString(i),"Malaria"
                 , "Malaria Case " + Integer.toString(i)));
73        }
74
75        for(int i = 1; i <= 45; i++)
76        {
77           collectionN2.addCase(new Case("Case " + Integer.toString(i),"AIDS", "
                 AIDS Case " + Integer.toString(i)));
78        }
79
80        for(int i = 1; i <= 20; i++)
81        {
82           collectionN3.addCase(new Case("Case " + Integer.toString(i),"Malaria"
                 , "Malaria Case " + Integer.toString(i)));
83        }
84
85        aggregationN1.addCollection(collectionN1);
86        aggregationN2.addCollection(collectionN2);
87        aggregationN3.addCollection(collectionN3);
88
89        cn1.add("Malaria");
90        cn2.add("AIDS");
91
92        assertTrue(n1.addAggregation(aggregationN1));
93        assertTrue(n2.addAggregation(aggregationN2));
94        assertTrue(n3.addAggregation(aggregationN3));
95
96        assertTrue(cb1.publishRecords(n1.getCollectionInfo(cn1)));
```

```
97          assertTrue(cb2.publishRecords(n2.getCollectionInfo(cn2)));
98          assertTrue(cb3.publishRecords(n3.getCollectionInfo(cn1)));
99      }
100
101     @Test
102     public void testSearch()
103     {
104         HashMap<Node,Integer> nodeList = cb1.searchRecords("Malaria");
105         assertEquals(nodeList.size(),2);
106         Iterator<Node> it = nodeList.keySet().iterator();
107         Node n;
108         while(it.hasNext())
109         {
110             n = it.next();
111             if(n.getName().equals("Test Node 1"))
112             {
113                 assertEquals(n.getAddress(),"127.0.0.1");
114                 assertEquals(nodeList.get(n),new Integer(15));
115             }
116             else if(n.getName().equals("Test Node 3"))
117             {
118                 assertEquals(n.getAddress(),"127.0.0.3");
119                 assertEquals(nodeList.get(n),new Integer(20));
120             }
121             else
122             {
123                 fail("Unidentified node found");
124             }
125         }
126
127         nodeList = cb1.searchRecords("AIDS");
128         assertEquals(nodeList.size(),1);
129         it = nodeList.keySet().iterator();
130         n = it.next();
131         assertEquals(n.getName(),"Test Node 2");
132         assertEquals(n.getAddress(),"127.0.0.2");
133         assertEquals(nodeList.get(n),new Integer(45));
134
135         nodeList = cb1.searchRecords("Cancer");
136         assertTrue(nodeList.isEmpty());
137
138         assertTrue(cb1.deleteRecords(cn1));
139         assertTrue(cb2.deleteRecords(cn2));
140         assertTrue(cb3.deleteRecords(cn1));
141
142         nodeList = cb1.searchRecords("Malaria");
143         assertTrue(nodeList.isEmpty());
144
145         nodeList = cb1.searchRecords("AIDS");
146         assertTrue(nodeList.isEmpty());
147     }
148
149 }
```

CBSubscribeTest.java

```java
package ch.hevs.ISyPeM2.Tests;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Random;

import org.junit.Before;
import org.junit.Test;

import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
import ch.hevs.ISyPeM2.ContextBroker.Subscription;
import ch.hevs.ISyPeM2.DB.LocalCaseProvisioner;
import ch.hevs.ISyPeM2.Node.Case;
import ch.hevs.ISyPeM2.Node.CaseAggregation;
import ch.hevs.ISyPeM2.Node.CaseCollection;
import ch.hevs.ISyPeM2.Node.Node;
import ch.hevs.ISyPeM2.NotificationServer.NotificationServer;

/**
 * This class is a JUnit test that checks if the code is able to publish some records to a certain
 * instance of the Orion Context Broker, subscribe to them and perform certain operations on the case
 * aggregation published to see if the changes are notified back through the subscription URL handled by the Jetty server
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class CBSubscribeTest
{
   final String contextBrokerURL = "http://orion.virtualbox:1026/NGSI10";

   NotificationServer notifySrv;
   final String responseURLNode = "http://host.virtualbox:8000/notify";

   ContextBroker cb1;
   ContextBroker cb2;

   Node n1;
   Node n2;

   CaseAggregation aggregationN1;
   CaseAggregation aggregationN2;

   HashSet<Case> malariaSet1;
   HashSet<String> cn1;
```

```
51
52      @Before
53      public void setUp() throws Exception
54      {
55          n1 = new Node("Test Node 1","127.0.0.1");
56          cb1 = new ContextBroker(contextBrokerURL,n1);
57          notifySrv = new NotificationServer(cb1,8000);
58          notifySrv.start();
59
60          n2 = new Node("Test Node 2","127.0.0.2");
61          cb2 = new ContextBroker(contextBrokerURL,n2);
62
63          n1.setProvisioner(new LocalCaseProvisioner());
64          n2.setProvisioner(new LocalCaseProvisioner());
65
66          malariaSet1 = new HashSet<Case>();
67          cn1 = new HashSet<String>();
68
69          CaseAggregation aggregationN1 = new CaseAggregation("Malaria", "People
                affected with malaria");
70          CaseAggregation aggregationN2 = new CaseAggregation("Malaria", "People
                affected with malaria");
71
72          CaseCollection collectionN1 = new CaseCollection("Malaria_ethnia_white",
                 "Malaria", "White people affected with malaria",n1);
73          CaseCollection collectionN2 = new CaseCollection("Malaria_ethnia_black",
                 "Malaria", "Black people affected with malaria",n2);
74
75          for(int i = 1; i <= 15; i++)
76          {
77              collectionN1.addCase(new Case("Case " + Integer.toString(i),"Malaria"
                    , "Malaria Case " + Integer.toString(i)));
78          }
79
80          aggregationN1.addCollection(collectionN1);
81          aggregationN2.addCollection(collectionN2);
82
83          cn1.add("Malaria");
84
85          assertTrue(n1.addAggregation(aggregationN1));
86          assertTrue(n2.addAggregation(aggregationN2));
87
88          assertTrue(cb1.publishRecords(n1.getCollectionInfo(cn1)));
89      }
90
91      @Test
92      public void test()
93      {
94          CaseCollection collectionN1 = n1.getAggregation("Malaria").
                getCollections().next();
95          CaseCollection collectionN2 = n2.getAggregation("Malaria").
                getCollections().next();
96
97          int retries = 0;
```

```
98          assertTrue(cb1.subscribeContext("Malaria", responseURLNode));
99          HashMap<String,Subscription> activeSubscriptions = cb1.
                getActiveSubscriptions();
100         assertEquals(activeSubscriptions.size(),1);
101         Subscription subs = activeSubscriptions.values().iterator().next();
102         String subsID = subs.getID();
103
104         //Add a case
105         try
106         {
107            collectionN1.addCase(new Case("Case " + Integer.toString(16),"Malaria
                ", "Malaria Case 16"));
108            n1.addCollection(collectionN1);
109            assertTrue(cb1.publishRecords(n1.getCollectionInfo(cn1)));
110            while((subs.getNumNodes() != 1) || (subs.getNumCases("Test Node 1")
                != 16))
111            {
112               retries++;
113               Thread.sleep(100);
114               if(retries == 100)
115               {
116                  assertEquals(subs.getNumNodes(),1);
117                  assertEquals(subs.getNumCases("Test Node 1"),16);
118               }
119            }
120            retries = 0;
121
122            //Delete a case
123
124            Iterator<Case> it = collectionN1.getCases();
125            Random randomGen = new Random();
126            int caseNum = randomGen.nextInt(16);
127            while(it.hasNext())
128            {
129               Case c = it.next();
130               if((caseNum--) == 0)
131               {
132                  collectionN1.removeCase(c.getID());
133                  break;
134               }
135            }
136            n1.addCollection(collectionN1);
137            assertTrue(cb1.publishRecords(n1.getCollectionInfo(cn1)));
138            while((subs.getNumNodes() != 1) || (subs.getNumCases("Test Node 1")
                != 15))
139            {
140               retries++;
141               Thread.sleep(100);
142               if(retries == 100)
143               {
144                  assertEquals(subs.getNumNodes(),1);
145                  assertEquals(subs.getNumCases("Test Node 1"),15);
146               }
147            }
```

```
148             retries = 0;
149
150             //Change case collection description
151
152             String newDescription = "Yayy! Malaria!";
153             CaseAggregation aggregation = n1.getAggregation("Malaria");
154             aggregation.setDescription(newDescription);
155             n1.addAggregation(aggregation);
156             assertTrue(cb1.updateRecordDescription("Malaria"));
157             while(!subs.getDescription().equals("Yayy! Malaria!"))
158             {
159                retries++;
160                Thread.sleep(100);
161                if(retries == 100)
162                {
163                   assertEquals(subs.getDescription(),"Yayy! Malaria!");
164                }
165             }
166             retries = 0;
167
168             //Change node address
169
170             assertTrue(n1.changeAddress("127.0.1.1",cb1));
171             while((subs.getNumNodes() != 1) || (!subs.getNodes().get("Test Node 1
                    ").getAddress().equals("127.0.1.1")))
172             {
173                retries++;
174                Thread.sleep(1000);
175                if(retries == 10)
176                {
177                   assertEquals(subs.getNumNodes(),1);
178                   assertEquals(subs.getNodes().values().iterator().next().
                          getAddress(),"127.0.1.1");
179                }
180             }
181             retries = 0;
182
183             //Add another node as published
184
185             collectionN2.addCase(new Case("Case 1", "Malaria", "Malaria Case 1"))
                    ;
186             n2.addCollection(collectionN2);
187             assertTrue(cb2.publishRecords(n2.getCollectionInfo(cn1)));
188             Thread.sleep(1000);
189             while(activeSubscriptions.values().iterator().next().getID().equals(
                    subsID))
190             {
191                retries++;
192                Thread.sleep(100);
193                if(retries == 100)
194                {
195                   assertFalse(activeSubscriptions.values().iterator().next().
                          getID().equals(subsID));
196                }
```

```
197             }
198             retries = 0;
199             subs = activeSubscriptions.values().iterator().next();
200             subsID = subs.getID();
201             while((subs.getNumNodes() != 2) || (subs.getNumCases("Test Node 2")
                    != 1))
202             {
203                retries++;
204                Thread.sleep(100);
205                if(retries == 100)
206                {
207                   assertEquals(subs.getNumNodes(),2);
208                   assertEquals(subs.getNumCases("Test Node 2"),1);
209                }
210             }
211             retries = 0;
212
213             //Second node adds a new case
214
215             collectionN2.addCase(new Case("Case 2", "Malaria", "Malaria Case 2"))
                    ;
216             n2.addCollection(collectionN2);
217             assertTrue(cb2.publishRecords(n2.getCollectionInfo(cn1)));
218             while((subs.getNumNodes() != 2) || (subs.getNumCases("Test Node 2")
                    != 2))
219             {
220                retries++;
221                Thread.sleep(100);
222                if(retries == 100)
223                {
224                   assertEquals(subs.getNumNodes(),2);
225                   assertEquals(subs.getNumCases("Test Node 2"),2);
226                }
227             }
228             retries = 0;
229
230             //Remove second node from subscription
231
232             assertTrue(cb2.deleteRecords(cn1));
233             Thread.sleep(1000);
234             while(activeSubscriptions.values().iterator().next().getID().equals(
                    subsID))
235             {
236                retries++;
237                Thread.sleep(100);
238                if(retries == 100)
239                {
240                   assertFalse(activeSubscriptions.values().iterator().next().
                          getID().equals(subsID));
241                }
242             }
243             subs = activeSubscriptions.values().iterator().next();
244             subsID = subs.getID();
245             assertEquals(subs.getNumNodes(),1);
```

```
246
247             //Unsubscribe and clean up
248
249             n1.getAggregation("Malaria").setDescription("People affected with
                    malaria");
250
251             assertTrue(cb1.unsubscribeContext(subs.getID()));
252             assertTrue(cb1.deleteRecords(cn1));
253             assertTrue(cb1.updateRecordDescription("Malaria"));
254
255             notifySrv.stopServer();
256         }
257         catch(Exception e)
258         {
259             fail(e.getMessage());
260         }
261     }
262 }
```

CaseGeneralizationTest.java

```java
package ch.hevs.ISyPeM2.Tests;

import java.util.TreeMap;

import ch.hevs.ISyPeM2.Generalization.TreeCollection;
import ch.hevs.ISyPeM2.Node.Case;
import ch.hevs.ISyPeM2.Node.CaseCollection;
import ch.hevs.ISyPeM2.Node.CategoricalAttribute;
import ch.hevs.ISyPeM2.Node.Node;
import ch.hevs.ISyPeM2.Node.NumericalAttribute;
/**
 * Test generalization of the case created as an example giver the state of RSDB
 * TODO ADD SQL FILE
 * @author Alevtina
 *
 */
public class CaseGeneralizationTest {
   public static void testCaseGen () throws Exception{
   Node node=new Node("NODE", "NODEadress");
   Case example = new Case("PS_e", "example case", "healthy");

   CaseCollection eColl = new CaseCollection("Collection", "example ", "example ", node);


   NumericalAttribute numAttr = new NumericalAttribute("age",99.0,99.0);
   CategoricalAttribute catAttr = new CategoricalAttribute("gender","female");
   example.addAttribute(numAttr);
   example.addAttribute(catAttr);

   eColl.addCase(example);

   TreeCollection treeColl = new TreeCollection();
   TreeMap<String, Boolean> treeIsCat = new TreeMap<String, Boolean>();

   treeIsCat=treeColl.JsonParse("/Users/uadmin/Documents/gits/ISyPeM2/ch.hevs.ISyPeM2.Node/src/ch/hevs/ISyPeM2/Generalization/inputfiles/trees.json");

   example.generalizeCase(treeColl, treeIsCat);

   }
}
```

DirectQueryDatabase.java

```java
package ch.hevs.ISyPeM2.Tests;
import java.util.HashSet;
import java.util.Iterator;

import org.kohsuke.args4j.CmdLineException;
import org.kohsuke.args4j.CmdLineParser;
import org.kohsuke.args4j.Option;

import ch.hevs.ISyPeM2.Core.ErrorPrinter;
import ch.hevs.ISyPeM2.DB.MySQLCaseProvisioner;
import ch.hevs.ISyPeM2.Node.CaseCollection;

/**
 * This class is a runnable class that serves as a template for a main class
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class DirectQueryDatabase
{
   @Option(name = "-v", aliases = { "--verbose" }, required = false, usage = "
      Increase verbosity to show INFO messages")
   private boolean verbose = false;

   @Option(name = "-vv", aliases = { "--verbose-debug" }, required = false,
      usage = "Increase verbosity to show DEBUG messages")
   private boolean debug = false;

   public static void main(String[] args)
   {
      DirectQueryDatabase arguments = new DirectQueryDatabase();
      CmdLineParser parser = new CmdLineParser(arguments);
      try
      {
         parser.parseArgument(args);
      }
      catch(CmdLineException exc)
      {
         parser.setUsageWidth(Integer.MAX_VALUE);
         parser.printUsage(System.err);
         return;
      }

      if(arguments.verbose)
      {
         ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_INFO);
         System.setProperty("org.eclipse.jetty.LEVEL","INFO");
      }
      if(arguments.debug)
      {
         ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_DEBUG);
         System.setProperty("org.eclipse.jetty.LEVEL","DEBUG");
      }

```

```
52          //DO STUFF
53
54          final long startTime = System.currentTimeMillis();
55
56          HashSet<String> keywords = new HashSet<String>();
57          HashSet<String> attributes = new HashSet<String>();
58
59          keywords.add("gentamicin");
60          keywords.add("neonates");
61
62          attributes.add("BW");
63
64          try
65          {
66             int numCases = 0;
67
68             MySQLCaseProvisioner provisioner1 = new MySQLCaseProvisioner("mysql.
                   virtualbox", "root", "mysqladmin", "test2");
69             HashSet<CaseCollection> collections1 = provisioner1.queryDatabase("
                   Gentamicin", keywords, attributes);
70             Iterator<CaseCollection> itCol1 = collections1.iterator();
71             while(itCol1.hasNext())
72             {
73                numCases += itCol1.next().numCases();
74             }
75
76             MySQLCaseProvisioner provisioner2 = new MySQLCaseProvisioner("mysql.
                   virtualbox", "root", "mysqladmin", "test3");
77             HashSet<CaseCollection> collections2 = provisioner2.queryDatabase("
                   Gentamicin", keywords, attributes);
78             Iterator<CaseCollection> itCol2 = collections2.iterator();
79             while(itCol2.hasNext())
80             {
81                numCases += itCol2.next().numCases();
82             }
83             System.out.println(numCases + " cases found");
84          }
85          catch(Exception e)
86          {
87             ErrorPrinter.printLog("MainClass", "main", ErrorPrinter.
                   SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
88          }
89          final long duration = System.currentTimeMillis() - startTime;
90          long seconds = duration/1000;
91          long ms = duration - seconds*1000;
92          System.out.println("Execution time: " + seconds + "s " + ms + "ms ");
93       }
94    }
```

DynamicTest.java

```java
package ch.hevs.ISyPeM2.Tests;
import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.InetAddress;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashSet;
import java.util.Random;
import java.util.Scanner;

import org.kohsuke.args4j.CmdLineException;
import org.kohsuke.args4j.CmdLineParser;
import org.kohsuke.args4j.Option;

import ch.hevs.ISyPeM2.ContextBroker.ContextBroker;
import ch.hevs.ISyPeM2.Core.ErrorPrinter;
import ch.hevs.ISyPeM2.DB.MySQLCaseProvisioner;
import ch.hevs.ISyPeM2.Node.CaseAttribute;
import ch.hevs.ISyPeM2.Node.Node;
import ch.hevs.ISyPeM2.Node.NumericalAttribute;


/**
 * This class is a runnable test to perform a test negotiation between 3 agents
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class DynamicTest
{
    @Option(name = "-v", aliases = { "--verbose" }, required = false, usage = "Increase verbosity to show INFO messages")
    private boolean verbose = false;

    @Option(name = "-vv", aliases = { "--verbose-debug" }, required = false, usage = "Increase verbosity to show DEBUG messages")
    private boolean debug = false;

    @Option(name = "-r", aliases = { "--recreate" }, required = false, usage = "Recreate Nodes databases")
    private boolean recreate = false;

    private final static String MySQLHostname = "mysql.virtualbox";
    private final static String MySQLUser = "root";
    private final static String MySQLPass = "mysqladmin";

    private final static String schemaFile = "ch/hevs/ISyPeM2/Files/dynamic_test.sql";

```

```
48    public static void importSQL(Connection conn, InputStream in) throws
          SQLException
49    {
50       Scanner s = new Scanner(in);
51       s.useDelimiter("(;(\r)?\n)|(--\n)");
52       Statement st = null;
53       try
54       {
55          st = conn.createStatement();
56          while (s.hasNext())
57          {
58             String line = s.next();
59             if (line.startsWith("/*!") && line.endsWith("*/"))
60             {
61                int i = line.indexOf(' ');
62                line = line.substring(i + 1, line.length() - " */".length());
63             }
64
65             if (line.trim().length() > 0)
66             {
67                st.execute(line);
68             }
69          }
70       }
71       finally
72       {
73          if (st != null) st.close();
74       }
75       s.close();
76    }
77
78    public static void main(String[] args)
79    {
80       DynamicTest arguments = new DynamicTest();
81       CmdLineParser parser = new CmdLineParser(arguments);
82       try
83       {
84          parser.parseArgument(args);
85       }
86       catch(CmdLineException exc)
87       {
88          parser.setUsageWidth(Integer.MAX_VALUE);
89          parser.printUsage(System.err);
90          return;
91       }
92
93       if(arguments.verbose)
94       {
95          ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_INFO);
96          //System.setProperty("org.eclipse.jetty.LEVEL","INFO");
97       }
98       if(arguments.debug)
99       {
100         ErrorPrinter.setVerbosity(ErrorPrinter.SEVERITY_DEBUG);
```

```
101             //System.setProperty("org.eclipse.jetty.LEVEL","DEBUG");
102         }
103         if(arguments.recreate)
104         {
105             try
106             {
107                 Class.forName("com.mysql.jdbc.Driver");
108                 Connection conn = DriverManager.getConnection("jdbc:mysql://" +
                        MySQLHostname + "/", MySQLUser, MySQLPass);
109                 BufferedInputStream br = new BufferedInputStream(ClassLoader.
                        getSystemClassLoader().getResourceAsStream(schemaFile));
110                 DynamicTest.importSQL(conn, br);
111                 br.close();
112                 conn.close();
113             }
114             catch (ClassNotFoundException e)
115             {
116                 ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                        SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
117             }
118             catch (SQLException e)
119             {
120                 ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                        SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
121             }
122             catch (IOException e)
123             {
124                 ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                        SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
125             }
126         }
127
128         Node n1 = null;
129
130         try
131         {
132             String hostname = InetAddress.getLocalHost().getHostName();
133             ErrorPrinter.printLog("DynamicTest", "main", ErrorPrinter.
                    SEVERITY_INFO, "Running from " + hostname);
134             switch(hostname)
135             {
136                 case "tucson1":
137                     n1 = new Node("Node 1", "tucson1.virtualbox");
138                     n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                            MySQLUser, MySQLPass, "test1"));
139                     break;
140                 case "tucson2":
141                     n1 = new Node("Node 2", "tucson2.virtualbox");
142                     n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                            MySQLUser, MySQLPass, "test2"));
143                     break;
144                 case "tucson3":
145                     n1 = new Node("Node 3", "tucson3.virtualbox");
```

```
146                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test3"));
147                 break;
148             case "tucson4":
149                 n1 = new Node("Node 4", "tucson4.virtualbox");
150                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test4"));
151                 break;
152             case "tucson5":
153                 n1 = new Node("Node 5", "tucson5.virtualbox");
154                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test5"));
155                 break;
156             case "tucson6":
157                 n1 = new Node("Node 6", "tucson6.virtualbox");
158                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test6"));
159                 break;
160             case "tucson7":
161                 n1 = new Node("Node 7", "tucson7.virtualbox");
162                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test7"));
163                 break;
164             case "tucson8":
165                 n1 = new Node("Node 8", "tucson8.virtualbox");
166                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test8"));
167                 break;
168             case "tucson9":
169                 n1 = new Node("Node 9", "tucson9.virtualbox");
170                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test9"));
171                 break;
172             case "tucson10":
173                 n1 = new Node("Node 10", "tucson10.virtualbox");
174                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test10"));
175                 break;
176             default:
177                 n1 = new Node("Host", "tucson1.virtualbox");
178                 n1.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,
                        MySQLUser, MySQLPass, "test1"));
179         }
180         n1.startTuCSoN();
181
182         HashSet<CaseAttribute> attributesN1 = new HashSet<CaseAttribute>();
183         HashSet<CaseAttribute> attributes = new HashSet<CaseAttribute>();
184         HashSet<String> keywords = new HashSet<String>();
185         HashSet<String> aggregationNames = new HashSet<String>();
186
187         keywords.add("gentamicin");
188         keywords.add("neonates");
189
190         aggregationNames.add("Gentamicin");
```

```
191
192         ContextBroker cb1 = new ContextBroker("http://orion.virtualbox:1026/
                NGSI10",n1);
193
194         cb1.publishRecords(n1.getCollectionInfo(aggregationNames));
195
196         Thread.sleep(10000);
197
198         attributes.add(new NumericalAttribute("BW",500,1500));
199         attributes.add(new NumericalAttribute("BW",1500,2500));
200         attributes.add(new NumericalAttribute("BW",2500,3500));
201         attributes.add(new NumericalAttribute("BW",3500,4500));
202         attributes.add(new NumericalAttribute("BW",750,1750));
203         attributes.add(new NumericalAttribute("BW",1750,2750));
204         attributes.add(new NumericalAttribute("BW",2750,3750));
205         attributes.add(new NumericalAttribute("BW",1000,2000));
206         attributes.add(new NumericalAttribute("BW",2000,3000));
207         attributes.add(new NumericalAttribute("BW",3000,4000));
208
209         attributes.add(new NumericalAttribute("GA",22,26));
210         attributes.add(new NumericalAttribute("GA",24,28));
211         attributes.add(new NumericalAttribute("GA",26,30));
212         attributes.add(new NumericalAttribute("GA",28,32));
213         attributes.add(new NumericalAttribute("GA",30,34));
214         attributes.add(new NumericalAttribute("GA",32,36));
215         attributes.add(new NumericalAttribute("GA",34,38));
216         attributes.add(new NumericalAttribute("GA",36,40));
217         attributes.add(new NumericalAttribute("GA",38,42));
218         attributes.add(new NumericalAttribute("GA",40,44));
219
220
221         int size = attributes.size();
222         int attrN1 = new Random().nextInt(size);
223
224         int i = 0;
225         for(CaseAttribute attr : attributes)
226         {
227            if (i == attrN1)
228            {
229               attributesN1.add(attr);
230            }
231            i++;
232         }
233         System.out.println("Started at: " + System.currentTimeMillis());
234         n1.startNegotiation("Gentamicin", cb1, attributesN1, keywords, 1,"
                testTree",1);
235
236         n1.stopTuCSoN();
237      }
238      catch(Exception e)
239      {
240         ErrorPrinter.printLog("NegotiationScenario", "main", ErrorPrinter.
                SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
241      }
```

```
242         }
243 }
```

JettyTest.java

```java
package ch.hevs.ISyPeM2.Tests;

import static org.junit.Assert.*;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.ProtocolException;
import java.net.URL;

import org.junit.Before;
import org.junit.Test;

import ch.hevs.ISyPeM2.NotificationServer.NotificationServer;

/**
 * This class is a JUnit test that checks if the code is able to successfully start a Jetty server instance and receive,
 * process and answer an empty request through the intended URL where the notifications will be sent to
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class JettyTest
{
   final String notificationServerURL = "http://localhost:8000/notify";
   final int notificationServerPort = 8000;
   NotificationServer notifySrv;

   @Before
   public void setUp() throws Exception
   {
      notifySrv = new NotificationServer(null, notificationServerPort);
      notifySrv.start();
      while (!notifySrv.isStarted())
      {
         Thread.sleep(20);
      }
   }

   @Test
   public void testServer()
   {
      URL obj;
      try
      {
         obj = new URL(notificationServerURL);
         HttpURLConnection con;
         try
```

```
52          {
53             con = (HttpURLConnection) obj.openConnection();
54             try
55             {
56                con.setRequestMethod("POST");
57             }
58             catch (ProtocolException e)
59             {
60                fail(e.getMessage());
61             }
62
63             con.setRequestProperty("Content-Type", "application/json");
64             String body = "";
65             con.setDoOutput(true);
66             DataOutputStream wr = new DataOutputStream(con.getOutputStream());
67             wr.writeBytes(body);
68             wr.flush();
69             wr.close();
70
71             int responseCode = con.getResponseCode();
72             BufferedReader in = new BufferedReader(new InputStreamReader(con.
                  getInputStream()));
73             String inputLine;
74             StringBuffer response = new StringBuffer();
75             while ((inputLine = in.readLine()) != null)
76             {
77                response.append(inputLine);
78             }
79             in.close();
80
81             assertEquals(responseCode, 200);
82             assertTrue(response.toString().isEmpty());
83          }
84          catch (IOException e)
85          {
86             fail(e.getMessage());
87          }
88       }
89       catch (MalformedURLException e1)
90       {
91          fail(e1.getMessage());
92       }
93       notifySrv.stopServer();
94       try
95       {
96          notifySrv.join(10 * 1000);
97       }
98       catch (InterruptedException e)
99       {
100         fail(e.getMessage());
101      }
102      assertFalse(notifySrv.isAlive());
103   }
104}
```

MySQLTest.java

```java
package ch.hevs.ISyPeM2.Tests;

import static org.junit.Assert.fail;

import org.junit.Before;
import org.junit.Test;

import ch.hevs.ISyPeM2.DB.MySQLCaseProvisioner;

/**
 * This class is a JUnit test that checks if connection to a remote SQL server
     is possible with the supplied credentials
 *
 * @author Imanol-Mikel Barba Sabariego
 *
 */
public class MySQLTest
{
   final String MySQLHostname = "mysql.virtualbox";
   final String MySQLUser = "root";
   final String MySQLPass = "mysqladmin";
   final String MySQLDB = "test1";

   @Before
   public void setUp() throws Exception
   {

   }

   @Test
   public void testSQL()
   {
      try
      {
         new MySQLCaseProvisioner(MySQLHostname,MySQLUser,MySQLPass,MySQLDB);
      }
      catch(Exception e)
      {
         fail(e.getMessage());
      }
   }
}
```

SQLReadTest.java

```
1  package ch.hevs.ISyPeM2.Tests;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertNotNull;
5  import static org.junit.Assert.assertTrue;
6  import static org.junit.Assert.fail;
7
8  import java.util.HashMap;
9  import java.util.HashSet;
10 import java.util.Iterator;
11
12 import org.junit.Before;
13 import org.junit.Test;
14
15 import ch.hevs.ISyPeM2.DB.MySQLCaseProvisioner;
16 import ch.hevs.ISyPeM2.Node.Case;
17 import ch.hevs.ISyPeM2.Node.CaseAggregation;
18 import ch.hevs.ISyPeM2.Node.CaseAttribute;
19 import ch.hevs.ISyPeM2.Node.CaseCollection;
20 import ch.hevs.ISyPeM2.Node.CategoricalAttribute;
21 import ch.hevs.ISyPeM2.Node.Node;
22 import ch.hevs.ISyPeM2.Node.NumericalAttribute;
23
24 /**
25  * This class is a JUnit test that reads a previously stored case aggregation
        into a MySQL database using
26  * the {@link MySQLCaseProvisioner}and checks if the operation is successful
        and if the retrieved data matches
27  * what we initially stored
28  *
29  * @author Imanol-Mikel Barba Sabariego
30  *
31  */
32 public class SQLReadTest
33 {
34    final String MySQLHostname = "mysql.virtualbox";
35    final String MySQLUser = "root";
36    final String MySQLPass = "mysqladmin";
37    final String MySQLDB = "test1";
38
39    MySQLCaseProvisioner provisioner;
40    Node origin;
41
42    CaseAggregation origAggregation;
43    CaseCollection origCollection;
44    HashMap<String,Case> origCases = new HashMap<String,Case>();
45
46    @Before
47    public void setUp() throws Exception
48    {
49       origin = new Node("localhost","127.0.0.1");
50       origin.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,MySQLUser,
            MySQLPass,MySQLDB));
```

```
51          //origin.setProvisioner(new SQLiteCaseProvisioner("/home/imanol/test.db
                "));
52
53          origAggregation = new CaseAggregation("Malaria", "People affected with
                malaria");
54          origCollection = new CaseCollection("Malaria_ethnia_white", "Malaria", "
                White people affected with malaria",origin);
55
56          CaseAttribute cAttr;
57          String g = "M";
58          for(int i = 1; i <= 15; i++)
59          {
60             Case c = new Case("Case " + Integer.toString(i),"Malaria", "Malaria
                   Case " + Integer.toString(i));
61             cAttr = new NumericalAttribute("age",i,i);
62             c.addAttribute(cAttr);
63             cAttr = new CategoricalAttribute("gender",g);
64             c.addAttribute(cAttr);
65             origCollection.addCase(c);
66             origCases.put(c.getID(),c);
67             //Alternate the gender (MFMFMFMFMFM...)
68             if(g.equals("M"))
69             {
70                g = "F";
71             }
72             else
73             {
74                g = "M";
75             }
76          }
77
78          origCollection.addKeyword("malaria");
79          origCollection.addKeyword("caucassian");
80
81          origAggregation.addCollection(origCollection);
82
83          assertTrue(origin.addAggregation(origAggregation));
84       }
85
86       @Test
87       public void testSQL()
88       {
89          try
90          {
91             CaseAggregation aggregation = origin.getAggregation(origAggregation.
                   getName());
92
93             assertEquals(aggregation.getName(),origAggregation.getName());
94             assertEquals(aggregation.getDescription(),origAggregation.
                   getDescription());
95             assertEquals(aggregation.getTimestamp(),origAggregation.getTimestamp
                   ());
96
97             int collectionCount = 0;
```

```
98          Iterator<CaseCollection> itCol = aggregation.getCollections();
99          while(itCol.hasNext())
100         {
101            collectionCount++;
102            CaseCollection collection = itCol.next();
103            assertEquals(collection.getID(),origCollection.getID());
104            assertEquals(collection.getType(),origCollection.getType());
105            assertEquals(collection.getDescription(),origCollection.
                  getDescription());
106            assertEquals(collection.getOriginNode().getName(),origCollection.
                  getOriginNode().getName());
107            assertEquals(collection.getOriginNode().getAddress(),
                  origCollection.getOriginNode().getAddress());
108
109            HashSet<String> origAttrs = new HashSet<String>();
110            origAttrs.add("G_age");
111            origAttrs.add("G_gender");
112            origAttrs.add("age");
113            origAttrs.add("gender");
114
115            Iterator<CaseAttribute> itAttr = collection.getAttributes();
116            while(itAttr.hasNext())
117            {
118               String attr = itAttr.next().getName();
119               assertTrue(origAttrs.contains(attr));
120               origAttrs.remove(attr);
121            }
122            assertTrue(origAttrs.isEmpty());
123
124            HashSet<String> origKws = new HashSet<String>();
125            origKws.add("malaria");
126            origKws.add("caucassian");
127
128            Iterator<String> itKw = collection.getKeywords();
129            while(itKw.hasNext())
130            {
131               String kw = itKw.next();
132               assertTrue(origKws.contains(kw));
133               origKws.remove(kw);
134            }
135            assertTrue(origKws.isEmpty());
136
137            Iterator<Case> itCase = collection.getCases();
138            while(itCase.hasNext())
139            {
140               Case c = itCase.next();
141               Case origCase = origCases.get(c.getID());
142               assertNotNull(origCase);
143               assertEquals(c.getPS(), origCase.getPS());
144               assertEquals(c.getType(), origCase.getType());
145               assertEquals(c.getHealthData(), origCase.getHealthData());
146               int numAttr = 0;
147               Iterator<CaseAttribute> itCAttr = origCase.getAttributes();
148               while(itCAttr.hasNext())
```

```
149                 {
150                    numAttr++;
151                    CaseAttribute origCAttr = itCAttr.next();
152                    CaseAttribute cAttr = c.getAttribute(origCAttr.getName());
153                    assertEquals(cAttr,origCAttr);
154                 }
155                 assertEquals(numAttr,4);
156              }
157           }
158           assertEquals(collectionCount,1);
159           assertTrue(origin.removeAggregation(origAggregation.getName()));
160        }
161        catch(Exception e)
162        {
163           fail(e.getMessage());
164        }
165     }
166  }
```

SQLWriteTest.java

```
1  package ch.hevs.ISyPeM2.Tests;
2
3  import static org.junit.Assert.assertTrue;
4  import static org.junit.Assert.fail;
5
6  import org.junit.Before;
7  import org.junit.Test;
8
9  import ch.hevs.ISyPeM2.DB.MySQLCaseProvisioner;
10 import ch.hevs.ISyPeM2.Node.Case;
11 import ch.hevs.ISyPeM2.Node.CaseAggregation;
12 import ch.hevs.ISyPeM2.Node.CaseAttribute;
13 import ch.hevs.ISyPeM2.Node.CaseCollection;
14 import ch.hevs.ISyPeM2.Node.CategoricalAttribute;
15 import ch.hevs.ISyPeM2.Node.Node;
16 import ch.hevs.ISyPeM2.Node.NumericalAttribute;
17
18 /**
19  * This class is a JUnit test that writes a case aggregation into a MySQL
        database using the {@link MySQLCaseProvisioner}
20  * and checks if the operation is successful.
21  *
22  * @author Imanol-Mikel Barba Sabariego
23  *
24  */
25 public class SQLWriteTest
26 {
27    final String MySQLHostname = "mysql.virtualbox";
28    final String MySQLUser = "root";
29    final String MySQLPass = "mysqladmin";
30    final String MySQLDB = "test1";
31
32    MySQLCaseProvisioner provisioner;
33    CaseAggregation aggregation;
34    Node origin;
35
36    @Before
37    public void setUp() throws Exception
38    {
39       origin = new Node("localhost","127.0.0.1");
40       origin.setProvisioner(new MySQLCaseProvisioner(MySQLHostname,MySQLUser,
            MySQLPass,MySQLDB));
41       //origin.setProvisioner(new SQLiteCaseProvisioner("/home/imanol/test.db
            "));
42
43       aggregation = new CaseAggregation("Malaria", "People affected with
            malaria");
44       CaseCollection collection = new CaseCollection("Malaria_ethnia_white", "
            Malaria", "White people affected with malaria",origin);
45       CaseAttribute cAttr;
46
47       String g = "M";
48       for(int i = 1; i <= 15; i++)
```

```
49          {
50             Case c = new Case("Case " + Integer.toString(i),"Malaria", "Malaria
                  Case " + Integer.toString(i));
51             cAttr = new NumericalAttribute("age",i,10+i);
52             c.addAttribute(cAttr);
53             cAttr = new CategoricalAttribute("gender",g);
54             c.addAttribute(cAttr);
55             collection.addCase(c);
56             //Alternate the gender (MFMFMFMFMFM...)
57             if(g.equals("M"))
58             {
59                g = "F";
60             }
61             else
62             {
63                g = "M";
64             }
65          }
66
67          collection.addKeyword("malaria");
68          collection.addKeyword("caucassian");
69
70          aggregation.addCollection(collection);
71       }
72
73       @Test
74       public void testSQL()
75       {
76          try
77          {
78             assertTrue(origin.addAggregation(aggregation));
79             assertTrue(origin.removeAggregation(aggregation.getName()));
80          }
81          catch(Exception e)
82          {
83             fail(e.getMessage());
84          }
85       }
86    }
```

TransferTest.java

```
1  package ch.hevs.ISyPeM2.Tests;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertFalse;
5  import static org.junit.Assert.fail;
6
7  import java.io.File;
8  import java.io.FileInputStream;
9  import java.io.InputStream;
10 import java.security.MessageDigest;
11
12 import org.junit.Before;
13 import org.junit.Test;
14
15 import ch.hevs.ISyPeM2.Node.Case;
16 import ch.hevs.ISyPeM2.Node.CaseAggregation;
17 import ch.hevs.ISyPeM2.Node.CaseAttribute;
18 import ch.hevs.ISyPeM2.Node.CaseCollection;
19 import ch.hevs.ISyPeM2.Node.CategoricalAttribute;
20 import ch.hevs.ISyPeM2.Node.Node;
21 import ch.hevs.ISyPeM2.Node.NumericalAttribute;
22 import ch.hevs.ISyPeM2.TransferServer.TransferClient;
23 import ch.hevs.ISyPeM2.TransferServer.TransferServer;
24
25 /**
26  * This class is a JUnit test that checks if the code is able to successfully
        start a Jetty server instance and receive,
27  * process and answer an empty request through the intended URL where the
        notifications will be sent to
28  *
29  * @author Imanol-Mikel Barba Sabariego
30  *
31  */
32 public class TransferTest
33 {
34    final String testFile = "/home/imanol/test.db";
35    final int port = 7999;
36
37    TransferServer transferSrv;
38    String reference;
39    String originalMD5;
40
41    public byte[] createChecksum(String filename) throws Exception
42    {
43       InputStream fis = new FileInputStream(filename);
44       byte[] buffer = new byte[1024];
45       MessageDigest complete = MessageDigest.getInstance("MD5");
46       int numRead;
47       do
48       {
49          numRead = fis.read(buffer);
50          if (numRead > 0)
51          {
```

```
52              complete.update(buffer, 0, numRead);
53           }
54        }while (numRead != -1);
55        fis.close();
56        return complete.digest();
57     }
58
59     public String generateMD5(String file) throws Exception
60     {
61         byte[] b = createChecksum(file);
62           String result = "";
63
64           for (int i=0; i < b.length; i++) {
65              result += Integer.toString( ( b[i] & 0xff ) + 0x100, 16).substring(
                    1 );
66           }
67           return result;
68     }
69
70     @Before
71     public void setUp() throws Exception
72     {
73        Node origin = new Node("localhost","127.0.0.1");
74
75        CaseAggregation aggregation = new CaseAggregation("Malaria", "People
             affected with malaria");
76        CaseCollection collection = new CaseCollection("Malaria_ethnia_white", "
             Malaria", "White people affected with malaria",origin);
77        CaseAttribute cAttr;
78
79        String g = "M";
80        for(int i = 1; i <= 15; i++)
81        {
82           Case c = new Case("Case " + Integer.toString(i),"Malaria", "Malaria
                Case " + Integer.toString(i));
83           cAttr = new NumericalAttribute("age",i,10+i);
84           c.addAttribute(cAttr);
85           cAttr = new CategoricalAttribute("gender",g);
86           c.addAttribute(cAttr);
87           collection.addCase(c);
88           //Alternate the gender (MFMFMFMFMFM...)
89           if(g.equals("M"))
90           {
91              g = "F";
92           }
93           else
94           {
95              g = "M";
96           }
97        }
98
99        collection.addKeyword("malaria");
100       collection.addKeyword("caucassian");
101
```

```
102         aggregation.addCollection(collection);
103
104         transferSrv = new TransferServer(port);
105         transferSrv.start();
106         while (!transferSrv.isStarted())
107         {
108            Thread.sleep(20);
109         }
110         reference = this.transferSrv.postAggregation(aggregation);
111         String file = "/tmp/" + Integer.toHexString(System.identityHashCode(
               aggregation)) + ".db";
112         TransferClient.decompress(file + ".gz");
113         this.originalMD5 = this.generateMD5(file);
114         new File(file).delete();
115      }
116
117      @Test
118      public void testServer()
119      {
120         try
121         {
122            String newFile = TransferClient.requestTransfer("http://host.
                  virtualbox:" + port + "/get/" + reference);
123            assertEquals(this.originalMD5,this.generateMD5(newFile));
124         }
125         catch (Exception e)
126         {
127            fail(e.getMessage());
128         }
129
130         transferSrv.stopServer();
131         try
132         {
133            transferSrv.join(10 * 1000);
134         }
135         catch (InterruptedException e)
136         {
137            fail(e.getMessage());
138         }
139         assertFalse(transferSrv.isAlive());
140      }
141   }
```

TuCSoNTest.java

```
1  package ch.hevs.ISyPeM2.Tests;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertNotNull;
5  import static org.junit.Assert.assertTrue;
6  import static org.junit.Assert.fail;
7
8  import org.junit.Before;
9  import org.junit.Test;
10
11 import alice.logictuple.LogicTuple;
12 import alice.logictuple.exceptions.InvalidLogicTupleException;
13 import alice.tucson.api.AbstractTucsonAgent;
14 import alice.tucson.api.ITucsonOperation;
15 import alice.tucson.api.SynchACC;
16 import alice.tucson.api.TucsonTupleCentreId;
17 import alice.tucson.api.exceptions.TucsonInvalidAgentIdException;
18 import alice.tucson.api.exceptions.TucsonInvalidTupleCentreIdException;
19 import alice.tucson.api.exceptions.TucsonOperationNotPossibleException;
20 import alice.tucson.api.exceptions.UnreachableNodeException;
21 import alice.tuplecentre.api.exceptions.OperationTimeOutException;
22 import alice.tuplecentre.core.AbstractTupleCentreOperation;
23 import ch.hevs.ISyPeM2.Node.Node;
24 import ch.hevs.ISyPeM2.TuCSoN.TuCSoNService;
25
26 /**
27  * This class is a JUnit test that checks if the code is able to connect to a
       certain TuCSoN node and then make {@code in}
28  * and {@code out} operations in the "default" Tuple Centre
29  *
30  * @author Imanol-Mikel Barba Sabariego
31  *
32  */
33 public class TuCSoNTest extends AbstractTucsonAgent
34 {
35    final String tucsonServiceHostname = "tucson1.virtualbox";
36    final int tucsonServicePort = TuCSoNService.DEFAULT_PORT;
37    Node n1 = new Node("Test Node 1","127.0.0.1");
38    TuCSoNService tucsonService;
39    static int agentID = 0;
40    boolean finished;
41    TuCSoNTest testAgent1;
42    TuCSoNTest testAgent2;
43
44    public TuCSoNTest() throws TucsonInvalidAgentIdException,
          TucsonInvalidTupleCentreIdException, TucsonOperationNotPossibleException
          , UnreachableNodeException, OperationTimeOutException,
          InvalidLogicTupleException
45    {
46       super(new String("testAgent" + Integer.toString(agentID++)));
47       finished = false;
48       tucsonService = new TuCSoNService(tucsonServiceHostname,
             tucsonServicePort,n1);
```

```
49      }
50
51      @Before
52      public void setUp() throws Exception
53      {
54         testAgent1 = new TuCSoNTest();
55         testAgent2 = new TuCSoNTest();
56      }
57
58      @Test
59      public void testTuCSoN()
60      {
61         try
62         {
63            testAgent1.go();
64            testAgent2.go();
65
66            int retries = 0;
67            while(!(testAgent1.hasFinished() && testAgent2.hasFinished()))
68            {
69               retries++;
70               Thread.sleep(100);
71               if(retries == 100)
72               {
73                  assertTrue(testAgent1.hasFinished());
74                  assertTrue(testAgent2.hasFinished());
75               }
76            }
77         }
78         catch (InterruptedException e)
79         {
80            fail(e.getMessage());
81         }
82      }
83
84      @Override
85      protected void main()
86      {
87         ITucsonOperation op;
88         SynchACC acc = this.getContext();
89         try
90         {
91            TucsonTupleCentreId defaultTC = this.tucsonService.getTupleCentre("
                  default");
92            if(this.getTucsonAgentId().toString().equals("testAgent1"))
93            {
94               op = acc.set(defaultTC, LogicTuple.parse("[hello(testAgent1)]"),
                     null);
95               assertNotNull(op);
96               assertTrue(op.isResultSuccess());
97               op = acc.in(defaultTC, LogicTuple.parse("hello(Agent)"), null);
98               assertNotNull(op);
99               assertTrue(op.isResultSuccess());
100              LogicTuple tuple = op.getLogicTupleResult();
```

```
101             assertEquals(tuple.getArg(0).toString(),"testAgent2");
102          }
103          else if(this.getTucsonAgentId().toString().equals("testAgent2"))
104          {
105             op = acc.in(defaultTC, LogicTuple.parse("hello(Agent)"), null);
106             assertNotNull(op);
107             assertTrue(op.isResultSuccess());
108             LogicTuple tuple = op.getLogicTupleResult();
109             assertEquals(tuple.getArg(0).toString(),"testAgent1");
110             op = acc.out(defaultTC, LogicTuple.parse("hello(testAgent2)"),
                   null);
111             assertNotNull(op);
112             assertTrue(op.isResultSuccess());
113          }
114          else
115          {
116             fail("Unknown agent ID : " + this.getTucsonAgentId().toString());
117          }
118          acc.exit();
119       }
120       catch(Exception e)
121       {
122          fail(e.getMessage());
123       }
124       finished = true;
125    }
126
127    public boolean hasFinished()
128    {
129       return finished;
130    }
131
132    @Override
133    public void operationCompleted(AbstractTupleCentreOperation arg0)
134    {
135
136    }
137
138    @Override
139    public void operationCompleted(ITucsonOperation arg0)
140    {
141
142    }
143 }
```

D.8 ch.hevs.ISyPeM2.TestSuites package

AllTestSuite.java

```
1  package ch.hevs.ISyPeM2.TestSuites;
2
3  import org.junit.runner.RunWith;
4  import org.junit.runners.Suite;
5
6  /**
7   * This class is a test suite for all of the suites available. It
        comprehensively checks all of the tests available
8   * @author Imanol-Mikel Barba Sabariego
9   *
10  */
11 @RunWith(Suite.class)
12 @Suite.SuiteClasses({ContextBrokerTestSuite.class,JettyTestSuite.class,
       TuCSoNTestSuite.class, SQLTestSuite.class})
13 public class AllTestSuite
14 {
15
16 }
```

ContextBrokerTestSuite.java

```
1  package ch.hevs.ISyPeM2.TestSuites;
2
3  import org.junit.runner.RunWith;
4  import org.junit.runners.Suite;
5
6  import ch.hevs.ISyPeM2.Tests.CBPublishTest;
7  import ch.hevs.ISyPeM2.Tests.CBSearchTest;
8  import ch.hevs.ISyPeM2.Tests.CBSubscribeTest;
9
10 /**
11  * This class is a test suite for all the tests of the Orion Context Broker
12  * @author Imanol-Mikel Barba Sabariego
13  *
14  */
15 @RunWith(Suite.class)
16 @Suite.SuiteClasses({CBPublishTest.class, CBSearchTest.class, CBSubscribeTest.
       class})
17 public class ContextBrokerTestSuite
18 {
19
20 }
```

JettyTestSuite.java

```java
package ch.hevs.ISyPeM2.TestSuites;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

import ch.hevs.ISyPeM2.Tests.JettyTest;
import ch.hevs.ISyPeM2.Tests.TransferTest;

/**
 * This class is a test suite for all the tests of the Jetty Embedded HTTP
    Server
 * @author Imanol-Mikel Barba Sabariego
 *
 */
@RunWith(Suite.class)
@Suite.SuiteClasses({JettyTest.class, TransferTest.class})
public class JettyTestSuite
{

}
```

SQLTestSuite.java

```java
package ch.hevs.ISyPeM2.TestSuites;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

import ch.hevs.ISyPeM2.Tests.MySQLTest;
import ch.hevs.ISyPeM2.Tests.SQLReadTest;
import ch.hevs.ISyPeM2.Tests.SQLWriteTest;

/**
 * This class is a test suite for all the tests related to SQL operations
 * @author Imanol-Mikel Barba Sabariego
 *
 */
@RunWith(Suite.class)
@Suite.SuiteClasses({MySQLTest.class, SQLWriteTest.class, SQLReadTest.class})
public class SQLTestSuite
{

}
```

TuCSoNTestSuite.java

```
1  package ch.hevs.ISyPeM2.TestSuites;
2
3  import org.junit.runner.RunWith;
4  import org.junit.runners.Suite;
5
6  import ch.hevs.ISyPeM2.Tests.TuCSoNTest;
7
8  /**
9   * This class is a test suite for all the tests of the TuCSoN-related code
10  * @author Imanol-Mikel Barba Sabariego
11  *
12  */
13 @RunWith(Suite.class)
14 @Suite.SuiteClasses({TuCSoNTest.class})
15 public class TuCSoNTestSuite
16 {
17
18 }
```

D.9 ch.hevs.ISyPeM2.TransferServer package

TransferClient.java

```
1  package ch.hevs.ISyPeM2.TransferServer;
2
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.FileOutputStream;
6  import java.io.IOException;
7  import java.net.HttpURLConnection;
8  import java.net.URL;
9  import java.nio.channels.Channels;
10 import java.nio.channels.ReadableByteChannel;
11 import java.util.zip.GZIPInputStream;
12
13 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
14
15 /**
16  * This class holds static methods to fetch a file from a remote {@link
       TransferServer}, decompress it and store it into
17  * disk
18  *
19  * @author Imanol-Mikel Barba Sabariego
20  */
21 public class TransferClient
22 {
23    /**
24     * Sends a request to a remote {@link TransferServer}, retrieves the file (
          if exists) and writes in into disk
25     *
26     * @param address Remote address where the resource is hosted
27     * @return Local path on disk where the resource has been stored, returns
          null if the resource is not available
28     *
29     * @throws IOException If the HTTP connection or the disk IO operations
          fail
30     */
31    public static String requestTransfer(String address) throws IOException
32    {
33       ErrorPrinter.printLog("TransferClient", "requestTransfer", ErrorPrinter.
             SEVERITY_INFO, "Requesting transfer from: " + address);
34       URL url = new URL(address);
35       HttpURLConnection con =(HttpURLConnection) url.openConnection();
36       con.setRequestMethod("HEAD");
37       if(con.getResponseCode() == HttpURLConnection.HTTP_OK)
38       {
39          url = new URL(address);
40          String filename = Integer.toHexString(System.identityHashCode(url));
41          ReadableByteChannel rbc = Channels.newChannel(url.openStream());
42          FileOutputStream fos = new FileOutputStream("/tmp/" + filename + ".gz
                ");
43          fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
44          fos.close();
```

```
45            decompress("/tmp/" + filename + ".gz");
46            new File("/tmp/" + filename + ".gz").delete();
47            ErrorPrinter.printLog("TransferClient", "requestTransfer",
                  ErrorPrinter.SEVERITY_INFO, "Transfer from: " + address + "
                  finished");
48            return "/tmp/" + filename;
49         }
50         return null;
51      }
52
53      /**
54       * Decompresses a downloaded resource, which are compressed using GZIP
55       *
56       * @param filename Path to the file to decompress
57       * @throws IOException If the disk IO operations fail
58       */
59      public static void decompress(String filename) throws IOException
60      {
61         byte[] buffer = new byte[1024];
62         GZIPInputStream gzis = new GZIPInputStream(new FileInputStream(filename)
               );
63         FileOutputStream out = new FileOutputStream(filename.replaceAll(".gz$",
               ""));
64         int len;
65         while ((len = gzis.read(buffer)) > 0)
66         {
67            out.write(buffer, 0, len);
68         }
69         gzis.close();
70         out.close();
71      }
72   }
```

TransferHandler.java

```
1  package ch.hevs.ISyPeM2.TransferServer;
2
3  import java.io.File;
4  import java.io.IOException;
5  import java.nio.channels.FileChannel;
6  import java.nio.file.StandardOpenOption;
7
8  import javax.servlet.ServletException;
9  import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 import org.eclipse.jetty.server.HttpOutput;
13 import org.eclipse.jetty.server.Request;
14 import org.eclipse.jetty.server.handler.AbstractHandler;
15
16 /**
17  * This class handles an incoming transfer request
18  *
19  * @author Imanol-Mikel Barba Sabariego
20  *
21  */
22 public class TransferHandler extends AbstractHandler
23 {
24    /**
25     * {@link TransferServer} instance from which we check if the requested
          resources exist and retrieve the resource's path
26     * on disk
27     */
28    private TransferServer transferServ;
29
30    /**
31     * Class constructor
32     *
33     * @param transferServ {@link TransferServer} instance storing the
          available resources
34     */
35    public TransferHandler(TransferServer transferServ)
36    {
37       this.transferServ = transferServ;
38    }
39
40    /**
41     * This method is called every time an incoming request arrives. It will
          check if the path and methods are correct and if the resource exists,
42     * and if it does, transfer it.
43     */
44    public void handle(String target,Request baseRequest,HttpServletRequest
          request,HttpServletResponse response) throws IOException,
          ServletException
45    {
46       baseRequest.setHandled(true);
47       if(request.getMethod().equals("GET"))
48       {
```

```
49          if(target.startsWith("/get/"))
50          {
51             String aggregationRequested = target.substring("/get/".length());
52             String path = this.transferServ.getAggregation(
                  aggregationRequested);
53             if(path == null)
54             {
55                response.setStatus(HttpServletResponse.SC_NOT_FOUND);
56             }
57             else
58             {
59                response.setStatus(HttpServletResponse.SC_OK);
60                response.setContentType("application/x-gzip");
61                ((HttpOutput) response.getOutputStream()).sendContent(
                     FileChannel.open(new File(path).toPath(),StandardOpenOption.
                     READ));
62                new File(path).delete();
63                this.transferServ.removeAggregation(aggregationRequested);
64             }
65             org.eclipse.jetty.util.log.Log.getLog().info("Received GET request
                   to " + target);
66          }
67          else
68          {
69             response.setStatus(HttpServletResponse.SC_FORBIDDEN);
70          }
71       }
72       else if(request.getMethod().equals("HEAD"))
73       {
74          if(target.startsWith("/get/"))
75          {
76             String aggregationRequested = target.substring("/get/".length());
77             String path = this.transferServ.getAggregation(
                  aggregationRequested);
78             if(path == null)
79             {
80                response.setStatus(HttpServletResponse.SC_NOT_FOUND);
81             }
82             else
83             {
84                response.setStatus(HttpServletResponse.SC_OK);
85             }
86             org.eclipse.jetty.util.log.Log.getLog().info("Received HEAD
                  request to " + target);
87          }
88          else
89          {
90             response.setStatus(HttpServletResponse.SC_FORBIDDEN);
91          }
92       }
93       else
94       {
95          response.setStatus(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
96       }
```

```
97          }
98      }
```

TransferServer.java

```
1  package ch.hevs.ISyPeM2.TransferServer;
2
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.FileNotFoundException;
6  import java.io.FileOutputStream;
7  import java.io.IOException;
8  import java.sql.SQLException;
9  import java.util.HashMap;
10 import java.util.zip.GZIPOutputStream;
11
12 import org.eclipse.jetty.server.Server;
13 import org.eclipse.jetty.server.ServerConnector;
14
15 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
16 import ch.hevs.ISyPeM2.DB.SQLiteCaseProvisioner;
17 import ch.hevs.ISyPeM2.Node.CaseAggregation;
18 import ch.hevs.ISyPeM2.Node.Node;
19
20 /**
21  * This class provides a minimal HTTP server to transfer files between nodes
        using Jetty.
22  *
23  * @author Imanol-Mikel Barba Sabariego
24  * @see <a href="http://eclipse.org/jetty/">Jetty project page</a>
25  */
26 public class TransferServer extends Thread
27 {
28    /**
29     * {@code Map} that holds all the resources made available to this server
           and their physical path on disk. The resource
30     * ID is used as key
31     */
32    private HashMap<String,String> availableAggregations;
33
34    /**
35     * Jetty {@code Server} object that manages the incoming HTTP requests
36     */
37    private Server server;
38
39    /**
40     * Default TCP port for the transfer server
41     */
42    public static final int DEFAULT_PORT = 7999;
43
44    /**
45     * Class constructor
46     *
47     * @param port TCP port where the server will be listening
48     */
49    public TransferServer(int port)
50    {
51          this.availableAggregations = new HashMap<String,String>();
```

```
52          this.server = new Server(port);
53      }
54
55      /**
56       * Checks if the Jetty server has started
57       * @return true if the server has started, false otherwise
58       */
59      public boolean isStarted()
60      {
61         return this.server.isStarted();
62      }
63
64      /**
65       * Thread start method. This will launch the Jetty embedded server and will
             end as the server does
66       */
67      public void run()
68      {
69         try
70         {
71            ServerConnector servConn = (ServerConnector)server.getConnectors()
                 [0];
72            servConn.setAcceptQueueSize(10);
73            servConn.setIdleTimeout(30000);
74            server.setHandler(new TransferHandler(this));
75            server.start();
76            server.join();
77         }
78         catch (Exception e)
79         {
80            ErrorPrinter.printLog("NotificationServer", "run", ErrorPrinter.
                 SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
81         }
82      }
83
84      /**
85       * Stops the currently running Jetty server. This will also make the {@code
             run()} method end and consequently
86       * end the thread the {@code NotificationServer} is running in
87       */
88      public void stopServer()
89      {
90         try
91         {
92            server.stop();
93         }
94         catch(Exception exc)
95         {
96            ErrorPrinter.printLog("NotificationServer", "stopServer",
                 ErrorPrinter.SEVERITY_ERROR, exc.getMessage());
97         }
98      }
99
100     /**
```

```
101      * Compresses a file into a GZIP file
102      *
103      * @param path Path to the original file in the disk
104      * @return Path with the new GZIP file
105      *
106      * @throws FileNotFoundException If the file doesn't exist
107      * @throws IOException If a disk IO operation fails
108      */
109     private String compress(String path) throws FileNotFoundException,
             IOException
110     {
111        byte[] buffer = new byte[1024];
112        File file = new File(path);
113        String gzipFile = "/tmp/" + file.getName() + ".gz";
114        GZIPOutputStream gzos = new GZIPOutputStream(new FileOutputStream(
               gzipFile));
115        FileInputStream in = new FileInputStream(path);
116        int len;
117        while ((len = in.read(buffer)) > 0)
118        {
119           gzos.write(buffer, 0, len);
120        }
121        in.close();
122        gzos.finish();
123        gzos.close();
124
125        return gzipFile;
126     }
127
128     /**
129      * Posts case aggregation in this transfer server to make it available for
               download
130      *
131      * @param aggregation Case aggregation to be added
132      * @return String with the resource ID generated
133      *
134      * @throws FileNotFoundException If the generated SQLite DB is not found
135      * @throws IOException If any disk IO operation fails
136      * @throws ClassNotFoundException If the SQLite JDBC driver is not
               available
137      * @throws SQLException If the SQLite DB could not be generated
138      */
139     public String postAggregation(CaseAggregation aggregation) throws
             ClassNotFoundException, SQLException, FileNotFoundException, IOException
140     {
141        Node dummy = new Node("localhost","127.0.0.1");
142        String filename = Integer.toHexString(System.identityHashCode(
               aggregation)) + ".db";
143        dummy.setProvisioner(new SQLiteCaseProvisioner("/tmp/" + filename));
144        dummy.addAggregation(aggregation);
145
146        String gzipFile = this.compress("/tmp/" + filename);
147        new File("/tmp/" + filename).delete();
```

```
148         String reference = Integer.toHexString(System.identityHashCode(gzipFile)
                );
149         this.availableAggregations.put(reference,gzipFile);
150         return reference;
151     }
152 
153     /**
154      * Returns the physical path on disk of a certain resource
155      * @param target ID of the requested resource
156      *
157      * @return Path of the resource being requested
158      */
159     public String getAggregation(String target)
160     {
161         String path = this.availableAggregations.get(target);
162         return path;
163     }
164 
165     /**
166      * Removes a certain resource from this server
167      *
168      * @param target ID of the resource to delete
169      */
170     public void removeAggregation(String target)
171     {
172         this.availableAggregations.remove(target);
173     }
174 }
```

D.10 ch.hevs.ISyPeM2.TuCSoN package

NegotiatingAgent.java

```
1  package ch.hevs.ISyPeM2.TuCSoN;
2
3  import java.io.BufferedInputStream;
4  import java.io.IOException;
5  import java.util.HashMap;
6  import java.util.HashSet;
7  import java.util.Iterator;
8  import java.util.List;
9  import java.util.Map.Entry;
10
11 import alice.logictuple.LogicTuple;
12 import alice.logictuple.exceptions.InvalidLogicTupleException;
13 import alice.tucson.api.AbstractTucsonAgent;
14 import alice.tucson.api.ITucsonOperation;
15 import alice.tucson.api.SynchACC;
16 import alice.tucson.api.TucsonTupleCentreId;
17 import alice.tucson.api.exceptions.TucsonInvalidAgentIdException;
18 import alice.tucson.api.exceptions.TucsonInvalidTupleCentreIdException;
19 import alice.tucson.api.exceptions.TucsonOperationNotPossibleException;
20 import alice.tucson.api.exceptions.UnreachableNodeException;
21 import alice.tuplecentre.api.exceptions.OperationTimeOutException;
22 import alice.tuplecentre.core.AbstractTupleCentreOperation;
23 import alice.tuprolog.Term;
24 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
25 import ch.hevs.ISyPeM2.DB.CaseProvisioner;
26 import ch.hevs.ISyPeM2.DB.SQLiteCaseProvisioner;
27 import ch.hevs.ISyPeM2.Node.CaseAggregation;
28 import ch.hevs.ISyPeM2.Node.CaseAttribute;
29 import ch.hevs.ISyPeM2.Node.CaseCollection;
30 import ch.hevs.ISyPeM2.Node.Node;
31 import ch.hevs.ISyPeM2.Node.RSDBCollection;
32 import ch.hevs.ISyPeM2.TransferServer.TransferClient;
33 import ch.hevs.ISyPeM2.TransferServer.TransferServer;
34
35 /**
36  * This class is a TuCSoN agent whose purpose is to negotiate data exchange
        with other {@code NegotiatingAgents}, either
37  * by being the instigator of the negotiating process (host) or by being
        spawned by a {@link SpawningAgent} to resolve an
38  * ongoing negotiation
39  *
40  * @author Imanol-Mikel Barba Sabariego
41  *
42  */
43 public class NegotiatingAgent extends AbstractTucsonAgent
44 {
45    /**
46     * Variable that on {@code true} expresses that the current {@code
           NegotiatingAgent instance} is the host of a
47     * negotiating process
```

```
48      */
49     private boolean host;
50
51     /**
52      * Instance of the TuCSoN node service the agent is operating in
53      */
54     private TuCSoNService tucsonService;
55
56     /**
57      * Name of the case aggregation the agent is interested in negotiating
58      */
59     private String caseName;
60
61     /**
62      * Tuple Centre ID where the negotiation process that this agent is taking
            part of is taking place
63      */
64     private String TCName;
65
66     /**
67      * Agent Coordination Context to operate in Tuple Centres
68      */
69     private SynchACC acc;
70
71     /**
72      * {@code Set} containing the peers that were initially invited (as a Host)
            or containing the Host (as a Peer)
73      */
74     private HashSet<Node> peers;
75
76     /**
77      * {@code Map} of {@link CaseAttribute} objects using their name as key
           which contain either the set of attributes
78      * of the medical cases the agent's interested in (as a host) or the set of
            attributes of the medical cases that
79      * the host {@code NegotiatingAgent} is interested in
80      */
81     private HashMap<String,CaseAttribute> attributes;
82
83     /**
84      * {@code Set} of keywords contained in the case collection we're
           interested in (as a host) or that we're being
85      * requested (as a peer)
86      */
87     private HashSet<String> keywords;
88
89     /**
90      * Number of cases we're interested in obtaining as a host
91      */
92     private int numCases = 0;
93
94     /**
95      * Transfer server owned by the current agent. It is instantiated when a
           case transfer is requested.
```

```
96      */
97     private TransferServer transServ = null;
98
99     /**
100     * RSDB built as the result of negotiation
101     */
102    private RSDBCollection newRSDB = null;
103
104    /**
105     * Generalization tree
106     */
107    private String tree = null;
108
109    /**
110     * K parameter for generalization
111     */
112    private int K = 0;
113
114    /**
115     * Command sent to obtain the number of cases a peer can provide, given
          some attributes
116     * and keywords
117     */
118    private static final int NUMCASES_COMMAND = 0;
119    /**
120     * Command sent to request the transfer of cases from a peer
121     */
122    private static final int TRANSFER_COMMAND = 1;
123
124    /**
125     * Class constructor
126     * @param tucsonService TuCSoN node service instance we are operating in
127     * @param caseName Case aggregation name that the negotiation is about
128     * @param host Whether this {@code NegotiatingAgent} is host of a
          negotiating process or not
129     * @throws TucsonInvalidAgentIdException If the generated agent ID is not a
          valid first-order ground logic term
130     */
131    public NegotiatingAgent(TuCSoNService tucsonService, String caseName,
          boolean host) throws TucsonInvalidAgentIdException
132    {
133       //We're using the node name as a base for Agent ID
134       super("'negotiatingAgent_" + tucsonService.getNode().getName() + "'");
135       this.tucsonService = tucsonService;
136       this.caseName = caseName;
137       this.host = host;
138       this.TCName = null;
139       this.peers = new HashSet<Node>();
140       this.attributes = new HashMap<String,CaseAttribute>();
141       this.keywords = new HashSet<String>();
142    }
143
144    /**
145     * Checks whether this agent is the host of a negotiating process
```

```
146      * @return {@code true} if the agent is the host of a negotiating process,
             {@code false} otherwise
147      */
148     public boolean isHost()
149     {
150        return host;
151     }
152
153     /**
154      * Sets the Tuple Centre ID where the negotiating process that the agent's
            been invited to is taking place
155      * @param TCName Tuple Centre ID
156      */
157     public void setTCName(String TCName)
158     {
159        this.TCName = TCName;
160     }
161
162     /**
163      * Sets the number of cases we're interested in
164      * @param numCases Number of cases we want
165      */
166     public void setNumCases(int numCases)
167     {
168        this.numCases = numCases;
169     }
170
171     /**
172      * Sets the transfer server instance this agent will use to transfer cases
            to other peers
173      *
174      * @param transServ {@link TransferServer} instance to be used by the agent
175      */
176     public void setTransferServer(TransferServer transServ)
177     {
178        this.transServ = transServ;
179     }
180
181     /**
182      * Adds an address to the set of addresses the node will be contacting with
            to negotiate. Addresses must be added
183      * before the host {@code NegotiatingAgent} is started
184      * @param peer IP address of a peer to invite to negotiate
185      */
186     public void addPeer(Node peer)
187     {
188        if(!this.peers.contains(peer))
189        {
190           this.peers.add(peer);
191        }
192     }
193
194     /**
195      * Adds a case attribute that the agent is either interested in as a host
```

```
196          or that the {@code NegotiatingAgent} host
197      * is requesting
198      * @param attr {@link CaseAttribute} object representing the attribute
199      */
200     public void addAttribute(CaseAttribute attr)
201     {
202        if(!this.attributes.containsKey(attr))
203        {
204           this.attributes.put(attr.getName(),attr);
205        }
206     }
207
208     /**
209      * Adds a keyword to the {@link #keywords} {@code Set}
210      * @param kw Keyword to be added
211      */
212     public void addKeyword(String kw)
213     {
214        if(!this.keywords.contains(kw))
215        {
216           this.keywords.add(kw);
217        }
218     }
219
220     /**
221      * Sets the generalization tree for this negotiation
222      * @param tree Generalization tree in JSON format
223      */
224     public void setTree(String tree)
225     {
226        this.tree = tree;
227     }
228
229     /**
230      * Sets the K parameter for generalization for this negotiation
231      * @param K K Parameter
232      */
233     public void setK(int K)
234     {
235        this.K = K;
236     }
237
238     /**
239      * Sends an invite to an agent who is not currently partaking in the
240          current negotiation
241      *
242      * @param peer Node to invite to the negotiation
243      *
244      * @throws TucsonInvalidTupleCentreIdException
245      * @throws InvalidLogicTupleException
246      * @throws TucsonOperationNotPossibleException
247      * @throws UnreachableNodeException
248      * @throws OperationTimeOutException
249      * @throws InterruptedException
```

```
248      */
249     public void addPeerToActiveNegotiation(Node peer) throws
            TucsonInvalidTupleCentreIdException, InvalidLogicTupleException,
            TucsonOperationNotPossibleException, UnreachableNodeException,
            OperationTimeOutException, InterruptedException
250     {
251        TucsonTupleCentreId TC = this.tucsonService.getTupleCentre(TCName);
252        String peerAgentID = "'negotiatingAgent_" + peer.getName() + "'";
253        acc.out(TC,LogicTuple.parse("resume(" + peerAgentID + ")"),null);
254        ITucsonOperation op;
255        do
256        {
257           op = acc.rdp(TC, LogicTuple.parse("invited(List)"), null);
258           if(op.isResultFailure())
259           {
260              Thread.sleep(1000);
261           }
262        }while(op.isResultFailure());
263        List<Term> agentList = op.getLogicTupleResult().getArg(0).toList();
264        Iterator<Term> itAgent = agentList.iterator();
265        while(itAgent.hasNext())
266        {
267           String agentID = itAgent.next().toString();
268           if(agentID.equals(peerAgentID))
269           {
270              return;
271           }
272        }
273        TuCSoNService peerService = new TuCSoNService(peer.getAddress(),
               TuCSoNService.DEFAULT_PORT,null);
274        TucsonTupleCentreId peerDefaultTC = peerService.getTupleCentre("default"
               );
275        while(true)
276        {
277           try
278           {
279              acc.out(peerDefaultTC, LogicTuple.parse("invite(negotiate,'" +
                     this.caseName + "','" + this.tucsonService.getNode().getAddress
                     () + "'," + this.TCName + ")"), (long)2000);
280              break;
281           }
282           catch(OperationTimeOutException e)
283           {
284              ErrorPrinter.printLog("NegotiatingAgent", "
                     addPeerToActiveNegotiation", ErrorPrinter.SEVERITY_ERROR, e.
                     getClass() + ": " + e.getMessage());
285           }
286        }
287        acc.out(TC, LogicTuple.parse("addInvite(" + peerAgentID + ")"), null);
288     }
289
290     /**
291      * Gets a Tuple Centre which name is formed by NODENAME_HASH where
            negotiations will take place from the TuCSoN Node Service. If the
```

```
292      * node has already been used for a negotiation, it will check if the
             previous negotiation failed, and if not, it will
293      * generate another hash until it finds an unused TC or one that can be
             reused
294      *
295      * @return {@code TucsonTupleCentreID} instance with a Tuple Centre
             available for negotiation
296      *
297      * @throws TucsonInvalidTupleCentreIdException
298      * @throws InvalidLogicTupleException
299      * @throws TucsonOperationNotPossibleException
300      * @throws UnreachableNodeException
301      * @throws OperationTimeOutException
302      * @throws IOException
303      */
304     private TucsonTupleCentreId acquireNegotiationTC() throws
             TucsonInvalidTupleCentreIdException, InvalidLogicTupleException,
             TucsonOperationNotPossibleException, UnreachableNodeException,
             OperationTimeOutException, IOException
305     {
306        TucsonTupleCentreId TC = null;
307        while(this.TCName == null)
308        {
309           String provisionalTCName = "'" + this.tucsonService.getNode().getName
                   ().concat("_" + Integer.toHexString(System.identityHashCode(new
                   Object()))) + "'";
310           TC = this.tucsonService.getTupleCentre(provisionalTCName);
311           ITucsonOperation op = acc.rdp(TC, LogicTuple.parse("negotiationStatus
                   (Status)"), null);
312           if(op.isResultSuccess())
313           {
314              if(op.getLogicTupleResult().getArg(0).toString().equals("failed"))
315              {
316                 this.TCName = provisionalTCName;
317              }
318           }
319           else
320           {
321              this.TCName = provisionalTCName;
322           }
323        }
324        BufferedInputStream br = new BufferedInputStream(ClassLoader.
               getSystemClassLoader()
325                  .getResourceAsStream("ch/hevs/ISyPeM2/TuCSoN/negotiation.rsp"));
326        byte[] res = new byte[br.available()];
327        br.read(res);
328        br.close();
329        acc.setS(TC,new String(res),null);
330        return TC;
331     }
332
333     /**
334      * Sends invites to all the peers in the peer list. Then, it writes the
             invited() tuple onto the Tuple Centre given
```

```
335      * as parameter, initiating the negotiation process
336      *
337      * @param TC Tuple Centre where the negotiation will take place
338      *
339      * @throws TucsonInvalidTupleCentreIdException
340      * @throws InvalidLogicTupleException
341      * @throws TucsonOperationNotPossibleException
342      * @throws UnreachableNodeException
343      * @throws OperationTimeOutException
344      */
345     private void sendInvites(TucsonTupleCentreId TC) throws
            TucsonInvalidTupleCentreIdException, InvalidLogicTupleException,
            TucsonOperationNotPossibleException, UnreachableNodeException,
            OperationTimeOutException
346     {
347        String inviteList = "[";
348        Iterator<Node> itAddr = this.peers.iterator();
349        while(itAddr.hasNext())
350        {
351           Node currentPeer = itAddr.next();
352           String peerAddress = currentPeer.getAddress();
353           TuCSoNService peerService = new TuCSoNService(peerAddress,
                  TuCSoNService.DEFAULT_PORT,null);
354           TucsonTupleCentreId peerDefaultTC = peerService.getTupleCentre("
                  default");
355           int retries = 3;
356           while(true)
357           {
358              try
359              {
360                 acc.out(peerDefaultTC, LogicTuple.parse("invite(negotiate,'" +
                        this.caseName + "','" + this.tucsonService.getNode().
                        getAddress() + "'," + this.TCName + ")"), (long)2000);
361                 inviteList += "'negotiatingAgent_" + currentPeer.getName() + "'
                        ";
362                 if(itAddr.hasNext())
363                 {
364                    inviteList += ",";
365                 }
366                 break;
367              }
368              catch(OperationTimeOutException e)
369              {
370                 ErrorPrinter.printLog("NegotiatingAgent", "sendInvites",
                        ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.
                        getMessage());
371                 if(--retries == 0)
372                 {
373                    ErrorPrinter.printLog("NegotiatingAgent", "sendInvites",
                           ErrorPrinter.SEVERITY_WARNING, "Invite aborted");
374                    break;
375                 }
376                 ErrorPrinter.printLog("NegotiatingAgent", "sendInvites",
                        ErrorPrinter.SEVERITY_WARNING, "Retrying invite...");
```

```
377                 }
378             }
379         }
380         inviteList += "]";
381         acc.out(TC, LogicTuple.parse("invited(" + inviteList + ")"), null);
382     }
383 
384     /**
385      * Waits until all invited peers have arrived to the negotiation Tuple
            Centre. The method contains a variable named
386      * {@code retries} that manages how many 10 seconds retries will be
            performed until the negotiation host starts kicking
387      * out the nodes that haven't arrived yet
388      *
389      * @param TC Tuple Centre where the negotiation is taking place
390      *
391      * @throws TucsonOperationNotPossibleException
392      * @throws UnreachableNodeException
393      * @throws OperationTimeOutException
394      * @throws InvalidLogicTupleException
395      */
396     private void waitForPeers(TucsonTupleCentreId TC) throws
            InvalidLogicTupleException, TucsonOperationNotPossibleException,
            UnreachableNodeException, OperationTimeOutException
397     {
398         boolean success = false;
399         int retries = 3;
400         do
401         {
402             /* Imanol Barba
403              *
404              * There is a good reason to do a non-blocking operation followed by
                    a Thread.sleep() call,
405              * instead of a blocking call with a timeout and that is the
                    following extract in the
406              * TuCSoN "documentation":
407              *
408              * "Notice that reaching the timeout just unblocks the agent, but the
                    request IS
409              * NOT REMOVED from TuCSoN node pending requests (will still be
                    served at sometime in the future)"
410              *
411              * Which practically means that the request will be buffered and
                    served later. Since there
412              * is *NO* documentation on this behavior, I'll consider it
                    unpredictable (I don't even think
413              * the author knows) and thus will not risk it by relying on TuCSoN,
                    but on Java's Thread class.
414              */
415             ITucsonOperation op = acc.rdp(TC,LogicTuple.parse("negotiationStatus(
                    active)"),null);
416             if(op.isResultFailure())
417             {
418                 if((retries--) == 0)
```

```
419             {
420                 HashSet<String> missingPeers = new HashSet<String>();
421                 do
422                 {
423                     op = acc.rdp(TC, LogicTuple.parse("inviteAccepted(List)"),
                            null);
424                     if(op.isResultFailure())
425                     {
426                         try
427                         {
428                             Thread.sleep(1000);
429                         }
430                         catch (InterruptedException e)
431                         {
432                             ErrorPrinter.printLog("NegotiatingAgent", "
                                    waitForPeers", ErrorPrinter.SEVERITY_ERROR, e.
                                    getClass() + ": " + e.getMessage());
433                         }
434                     }
435                 }while(op.isResultFailure());
436                 List<Term> acceptedList = op.getLogicTupleResult().getArg(0).
                        toList();
437                 do
438                 {
439                     op = acc.rdp(TC, LogicTuple.parse("invited(List)"), null);
440                     if(op.isResultFailure())
441                     {
442                         try
443                         {
444                             Thread.sleep(1000);
445                         }
446                         catch (InterruptedException e)
447                         {
448                             ErrorPrinter.printLog("NegotiatingAgent", "
                                    waitForPeers", ErrorPrinter.SEVERITY_ERROR, e.
                                    getClass() + ": " + e.getMessage());
449                         }
450                     }
451                 }while(op.isResultFailure());
452                 List<Term> invitedList = op.getLogicTupleResult().getArg(0).
                        toList();
453                 Iterator<Term> itInvited = invitedList.iterator();
454                 while(itInvited.hasNext())
455                 {
456                     boolean peerAccepted = false;
457                     String agentID = itInvited.next().toString();
458                     Iterator<Term> itAccepted = acceptedList.iterator();
459                     while(itAccepted.hasNext())
460                     {
461                         if(agentID.equals(itAccepted.next().toString()))
462                         {
463                             peerAccepted = true;
464                             break;
465                         }
```

```
466                     }
467                     if(!peerAccepted)
468                     {
469                        missingPeers.add(agentID);
470                     }
471                  }
472                  Iterator<String> itMissing = missingPeers.iterator();
473                  while(itMissing.hasNext())
474                  {
475                     acc.out(TC, LogicTuple.parse("removeInvite(" + itMissing.
                           next() + ")"), null);
476                  }
477                  retries = 2;
478               }
479               try
480               {
481                  Thread.sleep(10000);
482               }
483               catch (InterruptedException e)
484               {
485                  ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                     SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
486               }
487            }
488            else
489            {
490               success = true;
491            }
492         }while(!success);
493      }
494
495      /**
496       * Writes the negotiation parameters (keywords and case attributes) in the
             tuple centre
497       *
498       * @param TC Tuple Centre where the parameters will be written
499       *
500       * @throws InvalidLogicTupleException
501       * @throws TucsonOperationNotPossibleException
502       * @throws UnreachableNodeException
503       * @throws OperationTimeOutException
504       */
505      private void sendParameters(TucsonTupleCentreId TC) throws
            InvalidLogicTupleException, TucsonOperationNotPossibleException,
            UnreachableNodeException, OperationTimeOutException
506      {
507         String parametersTuple = "parameters(keywords([";
508         Iterator<String> itKw = keywords.iterator();
509         while(itKw.hasNext())
510         {
511            String kw = itKw.next();
512            parametersTuple += "keyword('" + kw + "')";
513            if(itKw.hasNext())
514            {
```

```
515             parametersTuple += ",";
516          }
517       }
518       parametersTuple += "]),attributes([";
519       Iterator<CaseAttribute> itAttr = attributes.values().iterator();
520       while(itAttr.hasNext())
521       {
522          CaseAttribute attr = itAttr.next();
523          parametersTuple += "attribute('" + attr.getName() + "','" + attr.
                getValue() + "')";
524          if(itAttr.hasNext())
525          {
526             parametersTuple += ",";
527          }
528       }
529       parametersTuple += "]),tree(" + this.tree + "),k(" + this.K + "))";
530       acc.inp(TC, LogicTuple.parse("parameters(Kws,Attrs,Tree,Kparam)"), null)
             ;
531       acc.out(TC, LogicTuple.parse(parametersTuple), null);
532    }
533
534    /**
535     * Reads the negotiation parameters (keywords and case attributes) from a
           Tuple Centre
536     *
537     * @param TC Tuple Centre where to read the parameters from
538     *
539     * @throws InvalidLogicTupleException
540     * @throws TucsonOperationNotPossibleException
541     * @throws UnreachableNodeException
542     * @throws OperationTimeOutException
543     * @throws InterruptedException
544     */
545    private void readParameters(TucsonTupleCentreId TC) throws
          InvalidLogicTupleException, TucsonOperationNotPossibleException,
          UnreachableNodeException, OperationTimeOutException,
          InterruptedException
546    {
547       this.attributes.clear();
548       this.keywords.clear();
549       ITucsonOperation op;
550       do
551       {
552          op = acc.rdp(TC, LogicTuple.parse("parameters(Kws,Attrs,Tree,Kparam)"
                ), null);
553          if(op.isResultFailure())
554          {
555             Thread.sleep(1000);
556          }
557       }while(op.isResultFailure());
558       LogicTuple keywordsTuple = LogicTuple.parse(op.getLogicTupleResult().
             getArg(0).toString());
559       List<Term> kwList = keywordsTuple.getArg(0).toList();
560       Iterator<Term> itKw = kwList.iterator();
```

```
561         while(itKw.hasNext())
562         {
563            LogicTuple tuple = LogicTuple.parse(itKw.next().toString());
564            String kw = tuple.getArg(0).toString();
565            this.addKeyword(kw);
566         }
567         LogicTuple attributesTuple = LogicTuple.parse(op.getLogicTupleResult().
               getArg(1).toString());
568         List<Term> attrList = attributesTuple.getArg(0).toList();
569         Iterator<Term> itAttr = attrList.iterator();
570         while(itAttr.hasNext())
571         {
572            LogicTuple tuple = LogicTuple.parse(itAttr.next().toString());
573            String argName = tuple.getArg(0).toString();
574            String argVal = tuple.getArg(1).toString();
575            this.addAttribute(CaseAttribute.parseCaseAttribute(argName,argVal));
576         }
577         LogicTuple treeTuple = LogicTuple.parse(op.getLogicTupleResult().getArg
               (2).toString());
578         this.setTree(treeTuple.getArg(0).toString());
579         LogicTuple KTuple = LogicTuple.parse(op.getLogicTupleResult().getArg(3).
               toString());
580         this.setK(Integer.parseInt(KTuple.getArg(0).toString()));
581      }
582
583      /**
584       * Generates a {@code query} tuple given the type of query to generate
             given the target, the type of query and the
585       * arguments (if any)
586       *
587       * @param TC Tuple Centre where to send the {@code request} tuple
588       * @param query Type of query, these are defined in this class as {@code
             static final int} variables
589       * @param agentID Agent to whom the query is addressed
590       * @param arguments Arguments needed to generate the query if any
591       *
592       * @return String containing the {@code request} tuple
593       *
594       * @throws InvalidLogicTupleException
595       * @throws TucsonOperationNotPossibleException
596       * @throws UnreachableNodeException
597       * @throws OperationTimeOutException
598       */
599      private String generateQuery(TucsonTupleCentreId TC, int query, String
             agentID, Object... arguments) throws InvalidLogicTupleException,
             TucsonOperationNotPossibleException, UnreachableNodeException,
             OperationTimeOutException
600      {
601         String request = "query(" + agentID + ",";
602         if(query == NUMCASES_COMMAND)
603         {
604            request += "numcases,null)";
605         }
606         else if(query == TRANSFER_COMMAND)
```

```
607         {
608             request += "transfer," + ((Integer)arguments[0]).toString() + ")";
609         }
610         return request;
611     }
612 
613     /**
614      * Processes a query from the negotiation host
615      *
616      * @param TC Tuple Centre where the negotiation is taking place
617      * @param query {@code LogicTuple} object containing the query() tuple sent
             by the negotiation host
618      *
619      * @throws InvalidLogicTupleException
620      * @throws TucsonOperationNotPossibleException
621      * @throws UnreachableNodeException
622      * @throws OperationTimeOutException
623      * @throws InterruptedException
624      */
625     private void processQuery(TucsonTupleCentreId TC, LogicTuple query) throws
         InvalidLogicTupleException, TucsonOperationNotPossibleException,
         UnreachableNodeException, OperationTimeOutException,
         InterruptedException
626     {
627         String command = query.getArg(1).toString();
628         String answerTuple = "";
629         if(command.equals("numcases"))
630         {
631             answerTuple = "answer(" + this.getTucsonAgentId() + ",";
632             this.readParameters(TC);
633             HashSet<CaseCollection> matchingCollections = this.tucsonService.
                 getNode().searchCollections(caseName, keywords, attributes);
634             /* Imanol Barba
635              * TODO: (Alevtina), now the agent has the paremeters asked and K and
                   the tree and the collections that match
636              * those parameters, reasoning should be here.
637              */
638             int casesAvailable = 0;
639             Iterator<CaseCollection> itCol = matchingCollections.iterator();
640             while(itCol.hasNext())
641             {
642                 casesAvailable += itCol.next().numCases();
643             }
644             answerTuple += casesAvailable + "," + query.toString() + ")";
645         }
646         else if(command.equals("transfer"))
647         {
648             answerTuple = "answer(" + this.getTucsonAgentId() + ",";
649             HashSet<CaseCollection> matchingCollections = this.tucsonService.
                 getNode().searchCollections(caseName, keywords, attributes);
650             int numCases = Integer.parseInt(query.getArg(2).toString());
651             int casesAvailable = 0;
652             Iterator<CaseCollection> itCol = matchingCollections.iterator();
653             while(itCol.hasNext())
```

```
654             {
655                casesAvailable += itCol.next().numCases();
656             }
657             if(numCases > casesAvailable)
658             {
659                answerTuple += "[no," + casesAvailable + "]";
660             }
661             else
662             {
663                CaseAggregation aggregation = new CaseAggregation(this.caseName,
                      this.tucsonService.getNode().getAggregation(this.caseName).
                      getDescription());
664                itCol = matchingCollections.iterator();
665                while(itCol.hasNext())
666                {
667                   aggregation.addCollection(itCol.next());
668                }
669                ErrorPrinter.printLog("NegotiatingAgent", "processQuery",
                      ErrorPrinter.SEVERITY_INFO, this.getTucsonAgentId() + ": Giving
                       " + aggregation.getNumCases() + " cases");
670                try
671                {
672                   String url = "http://" + this.tucsonService.getNode().
                         getAddress() + ":" + TransferServer.DEFAULT_PORT + "/get/" +
                          this.transServ.postAggregation(aggregation);
673                   answerTuple += "[yes,'" + url + "']";
674                }
675                catch (Exception e)
676                {
677                   answerTuple += "[error,null]";
678                }
679             }
680             answerTuple += "," + query.toString() + ")";
681          }
682          if(!answerTuple.equals(""))
683          {
684             ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                   SEVERITY_DEBUG, "Sending answer: " + answerTuple);
685             while(true)
686             {
687                try
688                {
689                   acc.out(TC, LogicTuple.parse(answerTuple), (long)2000);
690                   break;
691                }
692                catch(OperationTimeOutException e)
693                {
694                   ErrorPrinter.printLog("NegotiatingAgent", "processQuery",
                         ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.
                         getMessage());
695                }
696             }
697          }
698       }
```

```
699
700     /**
701      * Reads the answer of a query addressed to a certain agent and returns the
             result if applicable
702      *
703      * @param TC Tuple Centre where the negotiation is taking place
704      * @param request Request originally sent
705      * @return {@code Object} containing the result of the query (if any)
706      *
707      * @throws InvalidLogicTupleException
708      * @throws TucsonOperationNotPossibleException
709      * @throws UnreachableNodeException
710      * @throws OperationTimeOutException
711      * @throws InterruptedException
712      */
713     private Object processAnswer(TucsonTupleCentreId TC, String request) throws
              InvalidLogicTupleException, TucsonOperationNotPossibleException,
             UnreachableNodeException, OperationTimeOutException,
             InterruptedException
714     {
715        String command = LogicTuple.parse(request).getArg(1).toString();
716        Object result = null;
717        ITucsonOperation op;
718        do
719        {
720           op = acc.inp(TC, LogicTuple.parse("answer(AgentID,Answer," + request
                  + ")"), null);
721           if(op.isResultFailure())
722           {
723              Thread.sleep(1000);
724           }
725        }while(op.isResultFailure());
726        if(command.equals("numcases"))
727        {
728           LogicTuple resolution = op.getLogicTupleResult();
729           int casesAvailable = Integer.parseInt(resolution.getArg(1).toString()
                  );
730           result = new Integer(casesAvailable);
731        }
732        else if(command.equals("transfer"))
733        {
734           LogicTuple resolution = op.getLogicTupleResult();
735           List<Term> termList = resolution.getArg(1).toList();
736           String decision[] = {termList.get(0).toString(),termList.get(1).
                  toString() };
737           decision[1] = decision[1].substring(1, decision[1].length()-1);
738           result = decision;
739        }
740        return result;
741     }
742
743     /**
744      * Requests the number of cases from all the peers that are in a Tuple
          Centre
```

```
745      *
746      * @param TC Tuple Centre where to send the requests
747      * @return {@code Map} with all the number of cases each peer agent can
             provide, using
748      * their names as key
749      *
750      * @throws InvalidLogicTupleException
751      * @throws TucsonOperationNotPossibleException
752      * @throws UnreachableNodeException
753      * @throws OperationTimeOutException
754      * @throws InterruptedException
755      */
756     private HashMap<String,Integer> getPeerNumCases(TucsonTupleCentreId TC)
            throws InvalidLogicTupleException, TucsonOperationNotPossibleException,
            UnreachableNodeException, OperationTimeOutException,
            InterruptedException
757     {
758        ITucsonOperation op;
759        do
760        {
761           op = acc.rdp(TC, LogicTuple.parse("inviteAccepted(List)"), null);
762           if(op.isResultFailure())
763           {
764              Thread.sleep(1000);
765           }
766        }while(op.isResultFailure());
767        List<Term> agentList = op.getLogicTupleResult().getArg(0).toList();
768        do
769        {
770           op = acc.rdp(TC, LogicTuple.parse("finished(List)"), null);
771           if(op.isResultFailure())
772           {
773              Thread.sleep(1000);
774           }
775        }while(op.isResultFailure());
776        List<Term> finishedList = op.getLogicTupleResult().getArg(0).toList();
777        agentList.removeAll(finishedList);
778        Iterator<Term> itAgent = agentList.iterator();
779        HashMap<String,String> queries = new HashMap<String,String>();
780        while(itAgent.hasNext())
781        {
782           String agentID = itAgent.next().toString();
783           String request = this.generateQuery(TC, NUMCASES_COMMAND, agentID);
784           queries.put(agentID, request);
785           while(true)
786           {
787              try
788              {
789                 acc.out(TC, LogicTuple.parse(request), (long)2000);
790                 break;
791              }
792              catch(OperationTimeOutException e)
793              {
794                 ErrorPrinter.printLog("NegotiatingAgent", "getPeerNumCases",
```

```
                    ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.
                    getMessage());
795             }
796          }
797       }
798       HashMap<String,Integer> resolutions = new HashMap<String,Integer>();
799       Iterator<String> itQueries = queries.keySet().iterator();
800       while(itQueries.hasNext())
801       {
802          String agentID = itQueries.next();
803          String request = queries.get(agentID);
804          Integer casesAvailable = (Integer)this.processAnswer(TC, request);
805          if(casesAvailable.intValue() == 0)
806          {
807             acc.out(TC, LogicTuple.parse("removeInvite(" + agentID + ")"),
                    null);
808          }
809          else
810          {
811             resolutions.put(agentID, casesAvailable);
812          }
813       }
814       return resolutions;
815    }
816
817    /**
818     * Sends a case transfer request to a peer agent
819     *
820     * @param TC Tuple Centre where to send the request
821     * @param agentID ID of the agent that the request is addressed to
822     * @param numRecords Number of records being requested
823     * @return Number of cases obtained
824     *
825     * @throws OperationTimeOutException
826     * @throws UnreachableNodeException
827     * @throws TucsonOperationNotPossibleException
828     * @throws InvalidLogicTupleException
829     * @throws InterruptedException
830     */
831    private int requestRecords(TucsonTupleCentreId TC, String agentID, int
          numRecords) throws InvalidLogicTupleException,
          TucsonOperationNotPossibleException, UnreachableNodeException,
          OperationTimeOutException, InterruptedException
832    {
833       String request = this.generateQuery(TC, TRANSFER_COMMAND, agentID, new
             Integer(numRecords));
834       while(true)
835       {
836          try
837          {
838             acc.out(TC, LogicTuple.parse(request), (long)2000);
839             break;
840          }
841          catch(OperationTimeOutException e)
```

```
842             {
843                 ErrorPrinter.printLog("NegotiatingAgent", "requestRecords",
                        ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.getMessage
                        ());
844             }
845         }
846         String answer[] = (String[])this.processAnswer(TC, request);
847         if(answer[0].equals("yes"))
848         {
849             String url = answer[1];
850             String newFile = null;
851             try
852             {
853                 int retries = 3;
854                 while(newFile == null)
855                 {
856                     try
857                     {
858                         newFile = TransferClient.requestTransfer(url);
859                     }
860                     catch(IOException e)
861                     {
862                         ErrorPrinter.printLog("NegotiatingAgent", "requestRecords",
                                ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.
                                getMessage());
863                         if(--retries == 0)
864                         {
865                             ErrorPrinter.printLog("NegotiatingAgent", "requestRecords
                                    ", ErrorPrinter.SEVERITY_WARNING, "Download aborted");
866                             return 0;
867                         }
868                         ErrorPrinter.printLog("NegotiatingAgent", "requestRecords",
                                ErrorPrinter.SEVERITY_WARNING, "Retrying download...");
869                     }
870                 }
871                 CaseProvisioner provisioner = new SQLiteCaseProvisioner(newFile);
872                 CaseAggregation aggregation = provisioner.readCaseAggregation(this
                        .caseName);
873                 int casesStored = this.readTransferedCases(aggregation);
874                 if(aggregation.getNumCases() == numRecords)
875                 {
876                     acc.out(TC, LogicTuple.parse("finishedAgent(" + agentID + ")"),
                             null);
877                 }
878                 return casesStored;
879             }
880             catch (Exception e)
881             {
882                 ErrorPrinter.printLog("NegotiatingAgent", "requestRecords",
                        ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.getMessage
                        ());
883                 return 0;
884             }
885         }
```

```
886         else
887         {
888             acc.out(TC, LogicTuple.parse("removeInvite(" + agentID + ")"), null);
889             return 0;
890         }
891     }
892 
893     /**
894      * Reads the transfered cases and adds them to the RSDB
895      *
896      * @param aggregation {@link CaseAggregation} object that contains the
           cases transfered.
897      * @return The actual number of cases written into the RSDB (duplicates are
           discarded).
898      */
899     private int readTransferedCases(CaseAggregation aggregation)
900     {
901         /* Imanol Barba
902          *
903          * This is commented because ATM real RSDB's are still not implemented
               and thus this crashes
904          */
905 
906         /*int cases = 0;
907         RSDBCollection collection = null;
908         Iterator<CaseCollection> itCol = aggregation.getCollections();
909         while(itCol.hasNext())
910         {
911             RSDBCollection col = (RSDBCollection)itCol.next();
912             if(collection == null)
913             {
914                 collection = col;
915                 continue;
916             }
917             collection.unify(col);
918         }
919         if(this.newRSDB == null)
920         {
921             this.newRSDB = collection;
922             cases = this.newRSDB.numCases();
923         }
924         else
925         {
926             cases = this.newRSDB.unify(collection);
927         }
928         return cases;*/
929         return aggregation.getNumCases();
930     }
931 
932     /**
933      * TuCSoN agent code entry point
934      */
935     @Override
936     protected void main()
```

```
937     {
938        this.acc = this.getContext();
939        TucsonTupleCentreId TC = null;
940        ITucsonOperation op;
941        try
942        {
943           if(this.isHost())
944           {
945              if(this.peers.isEmpty())
946              {
947                 ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "No peer addresses set. Aborting...");
948              }
949              else if(this.attributes.isEmpty())
950              {
951                 ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "No attributes to negotiate have been set.
                       Aborting...");
952              }
953              else if(this.keywords.isEmpty())
954              {
955                 ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "No keywords have been set. Aborting...");
956              }
957              else if(this.numCases == 0)
958              {
959                 ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "No desired number of cases has been set.
                       Aborting...");
960              }
961              else if(this.tree == null)
962              {
963                 ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "No Generalization tree has been set.
                       Aborting...");
964              }
965              else if(this.K == 0)
966              {
967                 ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "No K Parameter has been set. Aborting...");
968              }
969              else
970              {
971                 TC = acquireNegotiationTC();
972                 sendInvites(TC);
973                 while(this.numCases > 0)
974                 {
975                    waitForPeers(TC);
976                    /* Imanol Barba
977                     *
978                     * Since we may potentially have removed more peers after
                          waitForPeers(), we should check
979                     * again if we still have peers to negotiate with
980                     */
```

```
981                 do
982                 {
983                     op = acc.rdp(TC, LogicTuple.parse("inviteAccepted(List)")
                        , null);
984                     if(op.isResultFailure())
985                     {
986                         Thread.sleep(1000);
987                     }
988                 }while(op.isResultFailure());
989                 List<Term> agentList = op.getLogicTupleResult().getArg(0).
                    toList();
990                 if(agentList.size() != 0)
991                 {
992                     sendParameters(TC);
993                     HashMap<String,Integer> candidates = this.getPeerNumCases
                        (TC);
994
995                     ErrorPrinter.printLog("NegotiatingAgent", "main",
                        ErrorPrinter.SEVERITY_DEBUG, candidates.toString());
996
997                     int availableCases = 0;
998                     Iterator<Integer> itNumCases = candidates.values().
                        iterator();
999                     while(itNumCases.hasNext())
1000                    {
1001                        availableCases += itNumCases.next().intValue();
1002                    }
1003                    if(availableCases >= this.numCases)
1004                    {
1005                        //Here we could do stuff if there are more cases that
                            requested
1006                    }
1007                    int caseCount = 0;
1008                    Iterator<Entry<String,Integer>> itCandidates = candidates
                        .entrySet().iterator();
1009                    while(itCandidates.hasNext())
1010                    {
1011                        Entry<String,Integer> entry = itCandidates.next();
1012                        caseCount += this.requestRecords(TC, entry.getKey(),
                            entry.getValue().intValue());
1013                    }
1014                    this.numCases -= caseCount;
1015                    ErrorPrinter.printLog("NegotiatingAgent", "main",
                        ErrorPrinter.SEVERITY_INFO, this.getTucsonAgentId() +
                        ": Got " + caseCount + " cases");
1016                }
1017                Thread.sleep(5000);
1018            }
1019            acc.out(TC, LogicTuple.parse("hostFinished"), null);
1020            /* Imanol Barba
1021             *
1022             * Now we check if after obtaining all the cases there is
                    anyone left (latecomers
1023             * from peer invitations and such)
```

```
1024                 */
1025                do
1026                {
1027                   op = acc.rdp(TC, LogicTuple.parse("inviteAccepted(List)"),
                          null);
1028                   if(op.isResultFailure())
1029                   {
1030                      Thread.sleep(1000);
1031                   }
1032                }while(op.isResultFailure());
1033                List<Term> agentList = op.getLogicTupleResult().getArg(0).
                       toList();
1034                if(agentList.size() != 0)
1035                {
1036                   Iterator<Term> itAg = agentList.iterator();
1037                   while(itAg.hasNext())
1038                   {
1039                      String agentID = itAg.next().toString();
1040                      acc.out(TC, LogicTuple.parse("finishedAgent(" + agentID +
                             ")"), null);
1041                   }
1042                }
1043                //TODO: Save this.newRSDB to the Database
1044                this.tucsonService.getNode().finishedNegotiation(this, this.
                       caseName);
1045                ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_INFO, "Negotiation finished");
1046             }
1047          }
1048          else
1049          {
1050             boolean negotiationFinished = false;
1051             if(this.TCName == null)
1052             {
1053                ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "TC name not set. Aborting...");
1054             }
1055             else if(this.peers.isEmpty())
1056             {
1057                ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
                       SEVERITY_ERROR, "Peer address not set. Aborting...");
1058             }
1059             else
1060             {
1061                TC = new TuCSoNService(this.peers.iterator().next().getAddress
                       (),TuCSoNService.DEFAULT_PORT,null).getTupleCentre(this.
                       TCName);
1062                do
1063                {
1064                   op = acc.rdp(TC, LogicTuple.parse("invited(AgentList)"),
                          null);
1065                   if(op.isResultFailure())
1066                   {
1067                      Thread.sleep(1000);
```

```
1068                    }
1069                }while(op.isResultFailure());
1070                acc.out(TC, LogicTuple.parse("hello(" + this.getTucsonAgentId()
                        + ")"), null);
1071                while(!negotiationFinished)
1072                {
1073                    do
1074                    {
1075                        op = acc.inp(TC, LogicTuple.parse("query(" + this.
                            getTucsonAgentId() + ",Command,Arguments)"),null);
1076                        if(op.isResultFailure())
1077                        {
1078                            ITucsonOperation opStatus = null;
1079                            do
1080                            {
1081                                opStatus = acc.rdp(TC, LogicTuple.parse("
                                    negotiationStatus(Status)"), null);
1082                                if(opStatus.isResultFailure())
1083                                {
1084                                    Thread.sleep(1000);
1085                                }
1086                            }while(opStatus.isResultFailure());
1087                            if(opStatus.getLogicTupleResult().getArg(0).toString()
                                .equals("success"))
1088                            {
1089                                negotiationFinished = true;
1090                                break;
1091                            }
1092                            Thread.sleep(1000);
1093                        }
1094                    }while(op.isResultFailure());
1095                    if(!negotiationFinished)
1096                    {
1097                        ErrorPrinter.printLog("NegotiatingAgent", "main",
                            ErrorPrinter.SEVERITY_DEBUG, "Received query: " + op.
                            getLogicTupleResult());
1098                        processQuery(TC,op.getLogicTupleResult());
1099                    }
1100                }
1101            }
1102        }
1103    }
1104    catch (InvalidLogicTupleException | TucsonOperationNotPossibleException
            | UnreachableNodeException | OperationTimeOutException |
            TucsonInvalidTupleCentreIdException | IOException |
            InterruptedException e)
1105    {
1106        ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
            SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
1107    }
1108    ErrorPrinter.printLog("NegotiatingAgent", "main", ErrorPrinter.
            SEVERITY_DEBUG, this.getTucsonAgentId() + " finished at: " + System.
            currentTimeMillis());
1109 }
```

```
1110
1111    @Override
1112    public void operationCompleted(AbstractTupleCentreOperation arg0)
1113    {
1114
1115    }
1116
1117    @Override
1118    public void operationCompleted(ITucsonOperation arg0)
1119    {
1120
1121    }
1122
1123 }
```

SpawningAgent.java

```
1  package ch.hevs.ISyPeM2.TuCSoN;
2
3  import alice.logictuple.LogicTuple;
4  import alice.logictuple.exceptions.InvalidLogicTupleException;
5  import alice.tucson.api.AbstractTucsonAgent;
6  import alice.tucson.api.ITucsonOperation;
7  import alice.tucson.api.SynchACC;
8  import alice.tucson.api.exceptions.TucsonInvalidAgentIdException;
9  import alice.tucson.api.exceptions.TucsonInvalidTupleCentreIdException;
10 import alice.tucson.api.exceptions.TucsonOperationNotPossibleException;
11 import alice.tucson.api.exceptions.UnreachableNodeException;
12 import alice.tuplecentre.api.TupleCentreId;
13 import alice.tuplecentre.api.exceptions.OperationTimeOutException;
14 import alice.tuplecentre.core.AbstractTupleCentreOperation;
15 import ch.hevs.ISyPeM2.Core.ErrorPrinter;
16 import ch.hevs.ISyPeM2.Node.Node;
17 import ch.hevs.ISyPeM2.TransferServer.TransferServer;
18
19 /**
20  * This class is a TuCSoN agent whose purpose is to receive incoming
        invitations to negotiate and spawn a
21  * {@link NegotiatingAgent} to handle it
22  *
23  * @author Imanol-Mikel Barba Sabariego
24  *
25  */
26 public class SpawningAgent extends AbstractTucsonAgent
27 {
28    /**
29     * Instance of the TuCSoN node service the agent is operating in
30     */
31    private TuCSoNService tucsonService;
32
33    /**
34     * Agent Coordination Context to operate in Tuple Centres
35     */
36    private SynchACC acc;
37
38    /**
39     * Control variable that signals agent termination
40     */
41    private boolean terminate;
42
43    /**
44     * Class constructor
45     * @param tucsonService TuCSoN node service instance we are operating in
46     * @throws TucsonInvalidAgentIdException If the generated agent ID is not a
           valid first-order ground logic term
47     */
48    public SpawningAgent(TuCSoNService tucsonService) throws
          TucsonInvalidAgentIdException
49    {
50       super("'spawningAgent_" + tucsonService.getNode().getName() + "'");
```

```
51          this.terminate = false;
52          this.tucsonService = tucsonService;
53      }
54
55      /**
56       * Toggles the {@link #terminate} variable and sends a fake invite so the
             thread running the agent code reads the tuple
57       * and then proceeds to check if termination has been signaled
58       */
59      public void terminateAgent()
60      {
61          this.terminate = true;
62          try
63          {
64              TupleCentreId defaultTC = tucsonService.getTupleCentre("default");
65              acc.out(defaultTC, LogicTuple.parse("invite(negotiate,Shutdown,'" +
                    this.tucsonService.getNode().getAddress() + "',default)"), null);
66          }
67          catch(Exception e)
68          {
69              ErrorPrinter.printLog("SpawningAgent", "terminateAgent", ErrorPrinter
                    .SEVERITY_ERROR, e.getClass() + " was thrown while trying to
                    terminate the spawning agent");
70          }
71      }
72
73      /**
74       * TuCSoN agent code entry point
75       */
76      @Override
77      protected void main()
78      {
79          this.acc = this.getContext();
80          TupleCentreId defaultTC = null;
81          try
82          {
83              defaultTC = tucsonService.getTupleCentre("default");
84          }
85          catch (TucsonInvalidTupleCentreIdException e1)
86          {
87              //This should not happen, but whatever
88              ErrorPrinter.printLog("SpawningAgent", "main", ErrorPrinter.
                    SEVERITY_ERROR, "Tuple centre \"default\" is invalid");
89              return;
90          }
91
92          TransferServer transServ = new TransferServer(TransferServer.
                DEFAULT_PORT);
93          transServ.start();
94          ITucsonOperation op;
95          while(true)
96          {
97              try
98              {
```

```
99              do
100             {
101                op = acc.inp(defaultTC, LogicTuple.parse("invite(negotiate,
                       CaseType,Address,TCID)"), null);
102                if(op.isResultFailure())
103                {
104                   try
105                   {
106                      Thread.sleep(5000);
107                   }
108                   catch (InterruptedException e)
109                   {
110                      ErrorPrinter.printLog("NegotiatingAgent", "main",
                           ErrorPrinter.SEVERITY_ERROR, e.getClass() + ": " + e.
                           getMessage());
111                   }
112                }
113             }while(op.isResultFailure());
114             if(terminate)
115             {
116                break;
117             }
118             else
119             {
120                LogicTuple resultTuple = op.getLogicTupleResult();
121                String negotiationTupleCentre = resultTuple.getArg(3).toString
                       ();
122                String caseName = resultTuple.getArg(1).toString().replace("'",
                        "");
123                String peerAddress = resultTuple.getArg(2).toString().replace("
                       '", "");
124                if(tucsonService.getNode().getAggregation(caseName) != null)
125                {
126                   try
127                   {
128                      NegotiatingAgent negotiator = new NegotiatingAgent(this.
                           tucsonService,caseName,false);
129                      negotiator.setTCName(negotiationTupleCentre);
130                      negotiator.addPeer(new Node("Host Node", peerAddress));
131                      negotiator.setTransferServer(transServ);
132                      negotiator.go();
133                   }
134                   catch (TucsonInvalidAgentIdException e)
135                   {
136                      ErrorPrinter.printLog("SpawningAgent", "main",
                           ErrorPrinter.SEVERITY_ERROR, "Generated name for the
                           NegotiatingAgent was invalid");
137                   }
138                }
139             }
140          }
141          catch (InvalidLogicTupleException e)
142          {
```

```
143             ErrorPrinter.printLog("SpawningAgent", "main", ErrorPrinter.
                    SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
144         }
145         catch (TucsonOperationNotPossibleException e)
146         {
147             ErrorPrinter.printLog("SpawningAgent", "main", ErrorPrinter.
                    SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
148         }
149         catch (UnreachableNodeException e)
150         {
151             ErrorPrinter.printLog("SpawningAgent", "main", ErrorPrinter.
                    SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
152         }
153         catch (OperationTimeOutException e)
154         {
155             ErrorPrinter.printLog("SpawningAgent", "main", ErrorPrinter.
                    SEVERITY_ERROR, e.getClass() + ": " + e.getMessage());
156         }
157       }
158       transServ.stopServer();
159   }
160
161   @Override
162   public void operationCompleted(AbstractTupleCentreOperation arg0)
163   {
164
165   }
166
167   @Override
168   public void operationCompleted(ITucsonOperation arg0)
169   {
170
171   }
172
173 }
```

TuCSoNService.java

```
1  package ch.hevs.ISyPeM2.TuCSoN;
2
3  import alice.tucson.api.TucsonTupleCentreId;
4  import alice.tucson.api.exceptions.TucsonInvalidTupleCentreIdException;
5  import ch.hevs.ISyPeM2.Node.Node;
6
7  /**
8   * This class represents a TuCSoN service node on the network
9   *
10  * @author Imanol-Mikel Barba Sabariego
11  *
12  */
13 public class TuCSoNService
14 {
15    /**
16     * Hostname of the node running the TuCSoN service
17     */
18    private String tucsonHostname;
19
20    /**
21     * Port where the TuCSoN service is reachable
22     */
23    private int tucsonPort;
24
25    /**
26     * {@link Node} owning this {@code TuCSoNService} instance. All TuCSoN
          operations requiring knowledge of the node and
27     * the cases, case collections and aggregations it holds will use this
          object to acquire such information
28     */
29    private Node ownNode;
30
31    /**
32     * Default port for the TuCSoN service
33     */
34    public static final int DEFAULT_PORT = 20504;
35
36    /**
37     * Class constructor
38     * @param hostname Hostname of the TuCSoN node
39     * @param port Port of the TuCSoN node
40     * @param ownNode {@link Node} instance that will use the TuCSoN service
41     */
42    public TuCSoNService(String hostname, int port, Node ownNode)
43    {
44       this.tucsonHostname = hostname;
45       this.tucsonPort = port;
46       this.ownNode = ownNode;
47    }
48
49    /**
50     * {@link #ownNode} getter
```

```
51      * @return The node performing operations in this {@code TuCSoNService}
            instance
52      */
53     public Node getNode()
54     {
55        return this.ownNode;
56     }
57
58     /**
59      * Retrieves, creating it if necessary, a tuple centre with a specific
            Tuple Centre ID
60      * @param centreID ID of the Tuple Centre to retrieve
61      * @return The requested Tuple Centre
62      * @throws TucsonInvalidTupleCentreIdException If the Tuple Centre ID is
            not a valid first-order ground logic term
63      */
64     public TucsonTupleCentreId getTupleCentre(String centreID) throws
            TucsonInvalidTupleCentreIdException
65     {
66        return new TucsonTupleCentreId(centreID,this.tucsonHostname,Integer.
              toString(this.tucsonPort));
67     }
68 }
```

negotiation.rsp

```
% Reaction for the start of a negotiation process
reaction(
   out(invited(AgentList)),
   (completion),
   (
      out(negotiationStatus(starting))
   )
).

% Reaction called before the previous one to check the presence of the
    inviteAccepted tuple and generate it if necessary
reaction(
   out(invited(AgentList)),
   (invocation),
   (
      nop(inviteAccepted(List)),
      out(inviteAccepted([])),
      out(finished([]))
   )
).

% Reaction for the arrival of a NegotiatingAgent
reaction(
   out(hello(AgentID)),
   (completion),
   (
      in(hello(AgentID)),
      rd(invited(AgentList)),
      rd(inviteAccepted(AcceptedList)),
      member(AgentID,AgentList),
      no_member(AgentID,AcceptedList),
      in(inviteAccepted(AcceptedList)),
      append([AgentID],AcceptedList,NewList),
      out(inviteAccepted(NewList))
   )
).

% Reaction to check if all Agents are present to start the negotiation process
reaction(
   out(inviteAccepted(AcceptedList)),
   (completion),
   (
      rd(invited(AgentList)),
      rd(inviteAccepted(AcceptedList)),
      length(AgentList,X),
      length(AcceptedList,Y),
      X = Y,
      X \= 0,
      in(negotiationStatus(starting)),
      out(negotiationStatus(active))
   )
).

```

```
53 % Reaction to add an Agent to an active negotiation process
54 reaction(
55    out(addInvite(AgentID)),
56    (completion),
57    (
58       in(addInvite(AgentID)),
59       in(negotiationStatus(_)),
60       in(invited(InvitedList)),
61       append(AgentID,InvitedList,NewList),
62       out(invited(NewList)),
63       in(inviteAccepted(AcceptedList)),
64       out(inviteAccepted(AcceptedList))
65    )
66 ).
67
68 % Reaction to remove an Agent from an active negotiation process
69 reaction(
70    out(removeInvite(AgentID)),
71    (completion),
72    (
73       in(removeInvite(AgentID)),
74       in(negotiationStatus(_)),
75       in(invited(InvitedList)),
76       rd(inviteAccepted(AcceptedList)),
77       delete(AgentID,InvitedList,NewInvitedList),
78       delete(AgentID,AcceptedList,NewAcceptedList),
79       out(invited(NewInvitedList)),
80       in(inviteAccepted(AcceptedList)),
81       out(inviteAccepted(NewAcceptedList))
82    )
83 ).
84
85 % Reaction to add an agent to the finished list when the agent finishes
86 reaction(
87    out(finishedAgent(AgentID)),
88    (completion),
89    (
90       in(finishedAgent(AgentID)),
91       rd(finished(FinishedList)),
92       rd(inviteAccepted(AcceptedList)),
93       member(AgentID,AcceptedList),
94       no_member(AgentID,FinishedList),
95       in(finished(FinishedList)),
96       append([AgentID],FinishedList,NewList),
97       out(finished(NewList))
98    )
99 ).
100
101 % Reaction to change negotiation status to success when all agents have
      finished (triggered by finished peer)
102 reaction(
103    out(finished(FinishedList)),
104    (completion),
105    (
```

```
106         rd(inviteAccepted(AcceptedList)),
107         length(FinishedList,X),
108         length(AcceptedList,Y),
109         X = Y,
110         X \= 0,
111         rd(hostFinished),
112         in(negotiationStatus(active)),
113         out(negotiationStatus(success))
114     )
115 ).
116 
117 % Reaction to change negotiation status to success when all agents have
        finished (triggered by finished host)
118 reaction(
119     out(hostFinished),
120     (completion),
121     (
122         rd(inviteAccepted(AcceptedList)),
123         length(FinishedList,X),
124         length(AcceptedList,Y),
125         X = Y,
126         X \= 0,
127         rd(hostFinished),
128         in(negotiationStatus(active)),
129         out(negotiationStatus(success))
130     )
131 ).
132 
133 % Reaction to remove an agent from the finished list
134 reaction(
135     out(resume(AgentID)),
136     (completion),
137     (
138         in(resume(AgentID)),
139         rd(finished(FinishedList)),
140         member(AgentID,FinishedList),
141         in(finished(FinishedList)),
142         delete(AgentID,FinishedList,NewFinishedList),
143         out(finished(NewFinishedList))
144     )
145 ).
146 
147 % no_member PROLOG function
148 
149 no_member(X,List) :- not(member(X,List)).
```