Grupo de Tecnología Informática
Inteligencia Artificial

GTI IA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Magentix 2

USER'S MANUAL

Versión 1.0 December 2010

AGREEMENT TECHNOLOGIES

GOBIERNO DE ESPAÑA
MINISTERIO DE CIENCIA E INNOVACIÓN

Authors:

Joan Bellver
Luis Búrdalo
Agustín Espinosa
Juan A. García-Pardo
Ricard López
Soledad Valero

Contents

List of Figures

List of Tables

Acknowledgments

Financial support from the Ministerio de Ciencia e Innovación of the Spanish Government under TIN2008-04446 project and under Consolider Ingenio CSD2007-00022 grant is kindly acknowledged.

CHAPTER

1

Introduction

1.1 Motivation

Magentix2 is an agent platform for open Multiagent Systems. Its main objective is to bring agent technology to real domains: business, industry, logistics, e-commerce, health-care, etc.

Magentix2 platform is proposed as a continuation of the Magentix platform[1]. The final goal is to extend the functionalities of Magentix, providing new services and tools to allow the secure and optimized management of open Multiagent Systems. Nowadays, Magentix2 provides support at three levels:

- Organization level, technologies and techniques related to agent societies.
- Interaction level, technologies and techniques related to communications between agents.
- Agent level, technologies and techniques related to individual agents (such as reasoning and learning).

Thus, Magentix2 platform uses technologies with the necessary capacity to cope with the dynamism of the system topology and with flexible interactions, which are both natural consequences of the distributed and autonomous nature of its components. In this sense, the platform

[1] `http://users.dsic.upv.es/grupos/ia/sma/tools/Magentix/index.php`

has been extended in order to support flexible interaction protocols and conversations, indirect communication and interactions among agent organizations. Moreover, other important aspects cover by the Magentix2 project are the security issues.

1.2 Manual Structure

In the following chapters, how Magentix2 platform must be installed, configured and used for programming agents is explained.

Specifically, chapter 2 clarifies how Magentix2 can be fully installed in only one host in a quickly and easy way. Furthermore, it is also explained how to develop and to execute simple Magentix2 agents.

Chapter 3 is about programming aspects in Magentix2. Thus, it is possible to consult in this chapter: the basic and more advanced classes of agents that the platform provides; the main issues related with agent communication; how to program agents in a secure environment; and finally, how agents can share information in an indirect way by means of the tracing service provide by Magentix2.

Chapter 4 explains the support for virtual organizations provided by the Magentix2 platform. In this way, this chapter gives details about how the THOMAS (Methods, Techniques and Tools for Open Multi-Agent Systems) framework has been integrated with Magentx2, and how Magentix2 agents can use it.

In order to customize the Magentix2 platform installation or distribute it in diverse hosts, the chapter 5 should be consulted. Concretely, this chapter is about administration and configuration aspects related with the different components of the platform: Apache Qpid, the implementation of AMQP (Advanced Message Queuing Protocol) used for agent communication; MySQL, the database server used to maintain persistent information about the virtual organizations manage by the platform; Apache Tomcat, which allows agents to access to and provide standard Java web services; Magentix2 platform services, such as the services which allows the communications with external agents or with the THOMAS framework; and the security module, which provides key features regarding security, privacy, openness and interoperability.

CHAPTER

2

Quick Start

2.1 Installing Magentix2 Desktop Edition

2.1.1 Requirements

- Oracle Java Development Kit (JDK) 6 update 01 or later[1].
- Ubuntu Linux version 9.10 or later.

2.1.2 Installation of Magentix2 Desktop

In order to install Magentix2 Desktop, the corresponded jar (for 32 bits architecture or 64 bits architecture) must be downloaded[2] firstly . Once Magentix2Desktop.jar is downloaded, the following command must be executed :

- If the architecture system is 32 bits:

```
$ java -jar Magentix2Desktop-i386.jar
```

- If the architecture system is 64 bits:

[1] http://www.oracle.com/technetwork/java/archive-139210.html

[2] The latest installable version is in: http://users.dsic.upv.es/grupos/ia/sma/tools/magentix2/downloads.php

```
$ java -jar Magentix2Desktop-x64.jar
```

Note that this commands uses superuser (root) privileges, therefore, prompt with **sudo** password will be required in the process.

Then, a graphical interface for the installation is provided. This interface guides the user through the installation process. It is possible to make a full installation or customize it. A **full installation** installs all of the components that Magentix2 needs to run appropriately. **Custom installation** is recommended only when any of these components have been installed [3] in the system. Please refer to section 5.1 to check which are these components.

When installation ends, an example is executed. This example checks that all is correctly configured and Magentix2 has been successfully installed and running.

The output of this example should be like the following one:

```
 Executing, I'm consumer
2010-12-13 13:07:48,364 INFO
[Thread-2] SingleAgent_Example.SenderAgent2 (?:?) -
        Executing, I'm the sender
Executing, I'm the sender
Mensaje received in consumer agent,
 by receiveACLMessage: Hello, I'm the sender
HearderValue
```

2.1.3 Magentix2 installation description

Once Magentix2 has been installed in the system in the full installation mode, the following folders are created:

- **bin/** includes the executable files and folders required to launch and start the platform and services. The main ones are the following, which allows users to start and stop the Magentix2 platform:
  - *Start-Magentix.sh*: it launches services (Tomcat, MySQL and Qpid) and platform agents (OMS, SF and bridge agents)[4]. This script is executed by the installation

[3] The installation of any of these components can cause conflicts in the system if already they exists. For example, if */etc/mysql* directory exists.

[4] Magentix2 is launched without security, to enabled security refer to section 5.6.3

application, so it is not necessary to execute it the first time the platform is used. If Magentix2 platform is stopped, in order to restart the services and platform agents, this script must be execute as follows:

```
$ cd  ~/Magentix2/bin
$ sh Start-Magentix.sh
```

– *Stop-Magentix.sh*: it stops services (Tomcat, MySQL and Qpid) and platform agents (OMS, SF and bridge agents). The commands needed to execute this script are:

```
$ cd  ~/Magentix2/bin
$ sh Stop-Magentix.sh
```

In addition, also the following sub-directories are included in **bin/**:

– **configuration/** sub-directory: includes the Settings.xml and loggin.xml configuration files, necessary to launch Magentix2 user agents.

– **security/** sub-directory: includes all required files to launch Magentix2 in secure mode.

- **docs/** includes javadoc and the Magentix2 documentation in Pdf format.
- **lib/** includes Magentix2 library an all additional libraries required by Magentix2. How to import this library in projects is explained in section 2.2.
- **examples/** includes some examples of Magentix2 agents implementation.
- **src/** includes Magentix2 sources.
- **thomas/** includes all services required by THOMAS and Magentix2 platform. It also includes a user web service example.

2.1.4 Unistalling Magentix2 Desktop Edition

Magentix2 is very simple to remove from one system. The steps to uninstall are:

1. Stopping Magentix2 platform.

```
$ cd bin
$ sh Stop-Magentix.sh
```

2. Uninstalling Apache Qpid.

```
$ sudo apt-get remove qpid
```

3. Deleting Magentix2 installation directory.

```
$ cd
$ sudo rm -r Magentix2
```

2.2 Developing and executing a first agent

This section explains step by step how to program a Magentix2 agent. The images shown here correspond to the Eclipse IDE, but everything should be similar in any other IDE. Magentix2 library works with jdk1.6 which is available at: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

First step is to start Eclipse and create a new project (MyFirstAgent). The java library magentix2.jar has to be included in the project as a referenced library. Magentix2 platform and the agents running on it need a configuration folder with two files: `Settings.xml` and `loggin.xml`. `Settings.xml` configures all the parameters related to the platform functionality, like mySQL parameters or how agents connect to QPid broker. `Loggin.xml` is the configuration file for the Magentix2 logger, in which is specified where log messages are displayed. Magentix2 uses log4j as debugger, for more information about this software, please, refer to: `http://logging.apache.org/log4j/1.2/manual.html`.

There is a valid configuration folder for any Magentix2 project in the bin sub-folder of Magentix2 installation folder. In this example project, this configuration folder will be used. Thus, it is only necessary to copy the folder `Magentix2/bin/configuration/` in the root folder of the project (in this case `/workspace/MyFirstAgent/}`), Finally, a new package named agent is added to the project. In figure 2.1 it is shown how Eclipse looks like after taking these actions.

The example shown here consists in two agents, agent Sender will send a message to agent Consumer who will write the content of the received message on the console. In order to set this example, it is required to create three java classes: Sender.java, Consumer.java and Main.java. Sender.java and Consumer.java will contain the code of the agents. Besides, Main.java will create the connexion to the broker for the agents and start them.

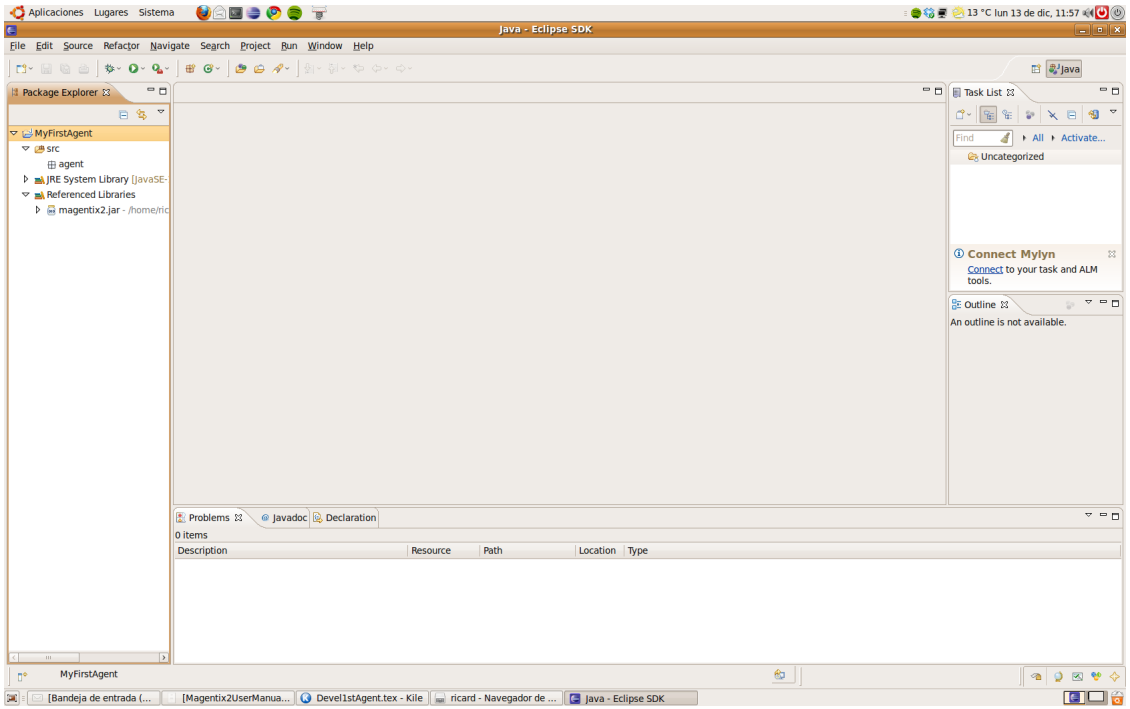

Figure 2.1: Project and package creation

The Sender.java class will be programmed the first. This class has to extend BaseAgent class 3.1.1. Therefore, it is necessary to import some classes from magentix2.jar. Eclipse will suggest to import some classes from magentix2.jar and will do the imports automatically. Figure 2.2 shows Sender.java at this moment. As it can be seen in the figure, the code of the agent has an error because it lacks a constructor. So, a basic constructor which calls the constructor of the base class is created.

A Magentix2 agent has three main methods `init`, `execute` and `finalize`. They are executed in the cited order. In the method `init`, the code that has to be executed at the beginning of the agent execution is added. The method `execute` is the main method of the agent and `finalize` is executed just before the agent ends its execution and is destroyed. In this specific example, it is only needed to implement code in the method `execute`. The code of the agent is shown below.

```
1 package agent;
2
3 import es.upv.dsic.gti_ia.core.ACLMessage;
4 import es.upv.dsic.gti_ia.core.AgentID;
5 import es.upv.dsic.gti_ia.core.BaseAgent;
```

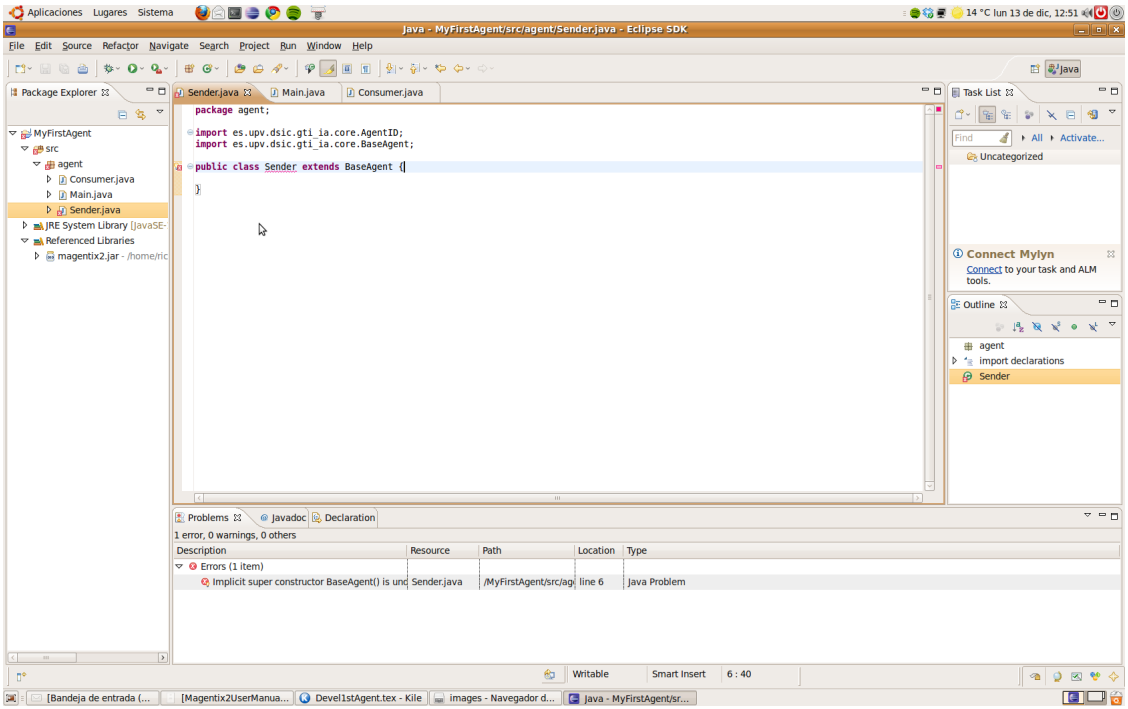

Figure 2.2: Programming a first agent with Eclipse

```
6
7  public class Sender extends BaseAgent {
8
9      public Sender(AgentID aid) throws Exception {
10         super(aid);
11     }
12
13     public void execute(){
14         System.out.println("Hi! I'm agent "+this.getName()+" and I
               start my execution");
15         ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
16         msg.setSender(this.getAid());
17         msg.addReceiver(new AgentID("Consumer"));
18         msg.setContent("Hi! I'm Sender agent and I'm running on
               Magentix2");
19         this.send(msg);
20     }
21
22 }
```

Following there is an explanation of all the code in the previously shown `execute` method line by line:

- The agent says hello and shows its name on the console (line 14).
- A new ACLMessage called *msg* is created (line 15). The performative of this message is Inform.
- This agent (Sender agent) is set as the sender of the message (line 16).
- The agent Consumer is added as a receiver of the agent (line 17).
- The content of the message *msg* is specified (line 18).
- Finally the agent sends the message, and with this ends its execution (line 19).

Now it is time to program the Consumer agent. This agent will wait until it receives the message from the Sender agent, then it will show the content of the message on the console and end its execution. The code of the agent is shown below.

```
1  package agent;
2
3  import es.upv.dsic.gti_ia.core.ACLMessage;
4  import es.upv.dsic.gti_ia.core.AgentID;
5  import es.upv.dsic.gti_ia.core.SingleAgent;
6
7  public class Consumer extends SingleAgent{
8
9     boolean gotMsg = false;
10
11    public Consumer(AgentID aid) throws Exception {
12       super(aid);
13    }
14
15    public void execute(){
16       System.out.println("Hi! I'm agent "+this.getName()+" and I
             start my execution");
17       ACLMessage msg = null;
18       try {
19          msg = this.receiveACLMessage();
20       } catch (InterruptedException e) {
```

```
21            e.printStackTrace();
22         }
23         System.out.println("Hi! I'm agent "+this.getName()+" and I've
               received the message: "+msg.getContent());
24      }
25   }
```

This agent does not extend from BaseAgent but from SingleAgent (section 3.1.2), this allows using the `receiveACLMessage` method. This method halts the agent execution until it receives a message. The method `execute` of the Consumer agent does nothing but wait until the agent receives a message. When the agent receives a message, it assigns the message to the variable `msg` and then it shows the message content on the console.

Once both agents are programmed, the Main.java class should be programmed. This class is in charge of connecting the agents to the broker and starting their execution. The code of this class is shown below.

```
1  package agent;
2
3  import org.apache.log4j.Logger;
4  import org.apache.log4j.xml.DOMConfigurator;
5  import es.upv.dsic.gti_ia.core.AgentID;
6  import es.upv.dsic.gti_ia.core.AgentsConnection;
7
8  public class Main {
9
10    public static void main(String[] args) {
11       /**
12       * Setting the Logger
13       */
14       DOMConfigurator.configure("configuration/loggin.xml");
15       Logger logger = Logger.getLogger(Main.class);
16
17       /**
18       * Connecting to Qpid Broker
19       */
20       AgentsConnection.connect("localhost", 5672,  "test", "guest",
             "guest", false);
21
22
```

```
23          try {
24              /**
25              * Instantiating a sender agent
26              */
27              Sender senderAgent = new Sender(new AgentID("Sender"));
28
29              /**
30              * Instantiating a consumer agent
31              */
32              Consumer consumerAgent = new Consumer(new AgentID("Consumer
                    "));
33
34              /**
35              * Execute the agents
36              */
37              consumerAgent.start();
38              senderAgent.start();
39
40          } catch (Exception e) {
41              logger.error("Error  " + e.getMessage());
42          }
43      }
44
45  }
```

In lines 14 and 15, the logger mechanism is set up. Its basic functionality is to show messages at some points of the code. This messages have a priority level associated, these levels go from info to error. It is needed to specify the configuration file for the debugger and the class to debug (Main class in this example). In line 20, the connection to the broker for all the agents launched in this class is set up. In this particuar case, it is specified that the QPid broker is running in the same host that the agents. The other parameters are the values for a default configuration of the broker. From lines 27 to 38, the agents are created, specifying an agent id for each one, and then they are started.

If Eclipse is used, the example can be run using the run button. The result of the execution will appear on the console, and it should be something similar to what is shown below.

```
Hi! I'm agent Consumer and I start my execution
Hi! I'm agent Sender and I start my execution
Hi! I'm agent Consumer and I've received the message: Hi! I'm Sender
agent and I'm running on Magentix2
```

CHAPTER

3

Programming agents

3.1 Basic classes for building agents: BaseAgent and SimpleAgent

3.1.1 BaseAgent

In order to create a basic Magentix2 agent, it is necessary to define a class which extends the class:
`es.upv.dsic.gti_ia.core.BaseAgent`. A unique identifier (with a new instance of AgentID class) must be associated to the agent and it is also necessary to implement the logic of the agent in the `execute()` method.

The following code shows how to implement a new BaseAgent class named SenderAgent. This agent only shows its name by the screen:

```
1 import es.upv.dsic.gti_ia.core.ACLMessage;
```

```
import es.upv.dsic.gti_ia.core.AgentID;
import es.upv.dsic.gti_ia.core.BaseAgent;

public class SenderAgent extends BaseAgent {

  public SenderAgent(AgentID aid) throws Exception {
        super(aid);
  }

  public void execute() {
        System.out.println("Executing, I'm " + getName());

  }

}
```

3.1.2 SimpleAgent

In order to create a simple Magentix2 agent, it is required to define a class which extends the class: `es.upv.dsic.gti_ia.core.SingleAgent`. It is a extended class from BaseAgent.

The SingleAgent defines a new message reception method (receiveACLMessage()) that performs blocking reception. It receives a new message in blocked mode. Then, when the agent retrieves the message, it is removed form the head of the agent's message queue.

The following code shows how to implement a new singleAgent with the receiveACLMessage() method:

```
public void execute() {
  /**
  * This agent has no definite work. Wait infinitely the arrival of
  * new messages.
  */
  try {
        /**
         * receiveACLMessage is a blocking function.
         * its waiting a new ACLMessage
         */
        ACLMessage msg = receiveACLMessage();
```

```
12
13          System.out.println("Mensaje received in " + this.getName()
14                  + " agent, by receiveACLMessage: " + msg.getContent
                        ());
15      } catch (Exception e) {
16          logger.error(e.getMessage());
17          return;
18      }
19  }
```

3.1.3 Initialization Tasks

Magentix2 platform uses log4j as a logging facility. It was developed by the Apache's Jakarta Project[1]. Its speed and flexibility allows log statements to remain in shipped code while giving the user the ability to enable logging at runtime without modifying any of the application binaries.

Log4j must be initialized inside the `main()` method of each Java application as follows:

```
1  DOMConfigurator.configure("configuration/loggin.xml");
2
3  Logger logger = Logger.getLogger(Run.class);
```

The file logging.xml [2] is used to specify what level of log messages are written to the log files for each component. Moreover, Log4j allows logging requests to print to multiple destinations called appenders.

Two appenders predefined for Magentix2 are showed in 3.1 and 3.2 figures. Specifically, the appender 1 (figure 3.1) indicates a file as the standard output. The appender 2 (figure 3.2) indicates the console as the standard output. Please, to learn more about appenders refer to: `http://logging.apache.org/log4j/1.2/index.html`

[1] http://jakarta.apache.org/

[2] This file is located in directory bin/configuration of the Magentix2 installation.

```
<appender name="File" class="org.apache.log4j.FileAppender">
        <param name="File" value="logs/Magentix2.log"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%t %-5p %c{2}
                   - %m%n"/>
        </layout>
</appender>
```

Figure 3.1: Appender 1

```
<appender name="Console" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d %-5p [%t] %C{2}
                (%F:%L) - %m%n"/>
        </layout>
</appender>
```

Figure 3.2: Appender 2

3.1.4 Connecting to the Qpid Broker

A connection to the Qpid broker must be established before launching any agent. This connection will be used by agents to communicate with each other. At this point, it is assumed that users have a Qpid broker running properly and the agents are launched without security.To employ connections with the security mode enabled, please refer to section 3.6.

The following parameters must be specified in any connection to the broker:

- <QpidHost> the host (or ip address) to connect to (defaults to 'localhost').
- <QpidPort> refers to the port to connect to (defaults to 5672).
- <QpidVhost> allows an Qpid 'virtual host' to be specified for the connection (defaults to 'test').
- <QpdidUser> user name to access Qpid.
- <QpidPassword> password to access Qpid.
- <QpidSSL> indicates if SSL is used during the connection (its value is always false when security is not enabled).

There are three different ways to establish a connection to the Qpid broker using the `connect()` method implemented in the `es.upv.dsic.gti_ia.core.AgentsConnection` class:

- Calling `connect()` without parameters. In this case the parameters are gathered from the Settings.xml [3] file. For example:

```
AgentsConnection.connect();
```

  Thus, it is possible to specify the connection parameters inside the Settings.xml file. Note that if all the parameters are not specified in the Settings.xml file, it is not feasible to use the `connect()` method without parameters.

  An example of the Settings.xml file could be:

```
<!-- Properties qpid broker -->
 <entry key="host">localhost</entry>
 <entry key="port">5672</entry>
 <entry key="vhost">test</entry>
 <entry key="user">guest</entry>
 <entry key="pass">guest</entry>
 <entry key="ssl">false</entry>
```

- Specifying all the parameters when calling `connect()`. Example:

```
AgentsConnection.connect("localhost",5672,"test","guest","guest
                         ",false);
```

- Specifying only the <qpidhost> parameter, leaving the rest as default parameters. In the current example case the default values will be (5672,"test","guest","guest",false) respectively. Example:

```
AgentsConnection.connect("host.domain");
```

[3] This file is located in the directory bin/configuration of the Magentix2 installation.

3.1.5 Running Agents

Once agents are implemented, they can be instantiated and launched. Please note that the platform do not allow different agents with the same name.

In order to instantiate an agent, an agent ID must be also created as follows:

- AgentID(String Identifier), where Identifier is the agent name.

Examples of creating a new instantiation:

```
SenderAgent agent1 = new SenderAgent(new AgentID("sender"));

ConsumerAgent agent2 = new ConsumerAgent(new AgentID("consumer"));
```

Once instantiated, agents can be launched by calling to their `start()` method.

Examples:

```
    agent1.start();
    agent2.start();
```

3.1.6 Optional Methods

The `init()` and `finalize()` methods are also defined in the `es.upv.dsic.gti_ia.core.BaseAgent` class. These methods are automatically executed before and after the `execute()` method. The programmer can overwrite these methods in order to include initialization or termination tasks.

3.1.7 Running Examples

In the examples folder of the Magentix2 package there are some basic examples of Magentix agents:

- BaseAgent_Example: this is an example of sender/consumer agents. The sender agent sends an ACLMessage to the consumer agent. When the ACLMessage arrives to the consumer agents, a message is shown.

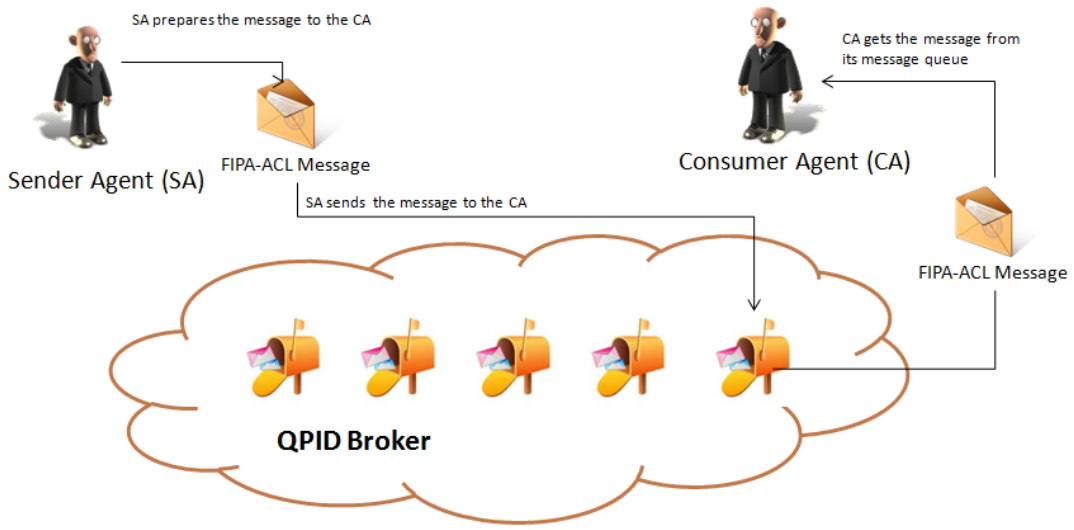

Figure 3.3: Messages exchange thought QPID Broker in Magentix2

- SingleAgent_Example: this is an example of sender/consumer agents. The sender agent sends an ACLMessage to the consumer agent. When the ACLMessage arrives to the consumer agents, a message is shown. The consumer is in blocked state waiting the message.

3.2 Agent Communication

In Magentix2, each agent has a message queue on the Qpid broker, where other agents can post messages addressed to her. The Figure 3.3 illustrates how a sender agent posts a message in a queue. Then, a consumer agent is able to read this message.

3.2.1 FIPA ACL Language

Messages exchanged by Magentix2 agents have the format specified by the ACL language defined by the FIPA[4] international standard for agent interoperability. This format comprises a number of fields, such as:

- Sender of the message.
- A list of receivers.

[4] `http://www.fipa.org`

- Performative: REQUEST, INFORM, QUERY_IF, CFP, PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL, etc.
- Content.
- Content Language.
- Content Ontology.
- Conversation-id, reply-with, in-reply-to, reply-by, etc.

A message in Magentix2 is implemented as an instance of the `es.upv.dsic.gti_ia.core.ACLMessage` class that provides get and set methods for handling all the fields of a message.

3.2.2 Sending Messages

To send a message to another agent the programmer must fill the fields of an ACLMessage object and then call the send() method of the `es.upv.dsic.gti_ia.core.BaseAgent` class.

The code below informs an agent whose identifier is receiver with the text: "Hello I'm sender".

```
1
2                   // Building a ACLMessage
3                   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
4                   msg.setReceiver(receiver);
5                   msg.setSender(this.getAid());
6                   msg.setLanguage("ACL");
7                   msg.setContent("Hello, I'm " + getName());
8
9
10                  // Sending a ACLMessage
11                  send(msg);
```

3.2.3 Receiving Messages

Whenever a message is posted in the message queue of an agent, this agent is notified by the *onMessage(ACLMessage msg)* method. This method allows agents to receive any message

automatically. Note that agents can also keep all received messages in an internal list (or queue) for reading them later. Agent programmers must overwrite the *onMessage* method when implementing a new agent, in order to process received messages. For instance:

```
1 public void onMessage(ACLMessage msg) {
2
3      // When a message arrives, it is shown in the screen
4
5    logger.info("Mensaje received in " + this.getName() + " agent,
      by onMessage: " + msg.getContent());
6 }
```

3.2.4 External Communication

An external agent is any agent not running over Magentix2 platform but communicating to any of the agents running on Magentix2. In this sense, Magentix2 implements the FIPA-HTTP message transport protocol by means of two special Magentix2 agents:

- *BridgeAgentInOut*: this agent is implemented in the `es.upv.dsic.gti_ia.core.BridgeAgentInOut` class. This agent is in charge of receiving all the messages sent by Magentix2 agents in which the recipient are agents running on another platform (that uses *http* as communication protocol). Then, the *BridgeAgentInOut* encapsulates the entire message and sends it via *http*.
- *BridgeAgentOutIn*: the implementation of this agent can be found in the `es.upv.dsic.gti_ia.core.BridgeAgentOutIn` class. The *BridgeAgentOutIn* routes messages from external agents (received via *http*) to Magentix2 agents. Therefore, BridgeAgentOutIn decodes the http message received and creates an ACLMessage message. After that, *BridgeAgentOutIn* sends the new created message to the recipient's mailbox.

It should be notice that the *BridgeAgentInOut* and the *BridgeAgentOutIn* agents must be launched and instantiate to allow external communication. This is made together with the rest of platform services and platforms agents by means of the *Start-Magentix.sh* command (explained in section 2.1.3. Thus, the *BridgeAgentInOut* and the *BridgeAgentOutIn* agents would be launched at localhost, and the agent *BridgeAgentOutIn* would be listening in the 8081 port. For other configurations, please refer to section 5.5.1

Inside Magentix2, external agents are identified (to send messages to them) by means of an *http* address that must be used when creating the corresponding AgentID. For instance, the following example shows the code which could be added into the *execute* method of a Magentix2 agent to send a Request message to another agent running into a JADE platform.

```
AgentID receiver = new AgentID();

//JADE default parameters.
receiver.name = "AgentName@hostname:1099/JADE";

//Host in which the JADE agent is running
receiver.host = "hostname.domain";

//JADE default port for ACC
receiver.port = "7778";

//Default protocol
receiver.protocol = "http";

/**
 * Building a ACLMessage         */

 //New Request message
 ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
 //The JADE agent is added as a receiver of the message
 msg.setReceiver(receiver);

 //The Magentix2 agent sends the message
 send(msg);
```

In a similar way, the following example shows how an external JADE agent sends a message to a MAgentix2 agent from the JADE platform (JADE source code):

```
AID receiver = new AID();

 //agentname@hotsname of the Magentix2 platform in which the agent
    is running
receiver.setName("consumer@hostname");

```

```
/Host in which the BridgeAgentOutIn agent is been executed and the
   port
//in which it is listened
receiver.addAddresses("http://host.domain:8081"); /

//Creation of a Request Message
ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);

 //Addition of the receiver of the message
msg.addReceiver(receiver);

//Sending the message
send(msg);
```

3.3 Basic conversational agents: QueueAgents

Magentix2 supports the set of basic interaction protocols defined by FIPA. Thereby, agents can communicate to each other by means of different protocols explained in this section.

Each implemented protocol provides the basic message exchange between two agents for a given type of conversation (a request, a query, etc.). In order to implement these protocols, a new agent template called *QueueAgent* has been defined. This template extends *BaseAgent* and can be found in the package `es.upv.dsic.gti_ia.architecture`. The figure 3.4 shows a UML class diagram for the QueueAgent class.

In order to use protocols, agents must extend the QueueAgent class and redefine its constructor. For instance:

```
        import es.upv.dsic.gti_ia.architecture.QueueAgent;
        import es.upv.dsic.gti_ia.core.AgentID;

        public class agent extends QueueAgent {

        public agent(AgentID aid) throws Exception {
                super(aid);
        }
```

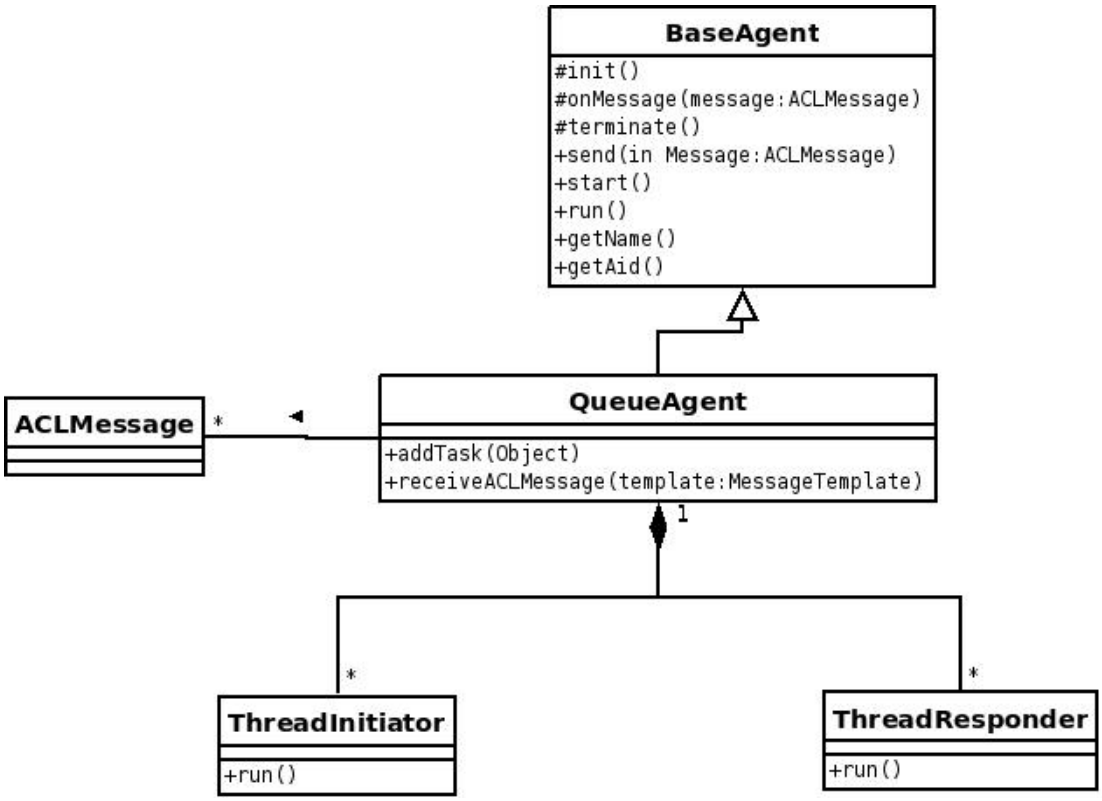

Figure 3.4: Diagram for the QueueAgent class

3.3.1 Communication Protocols on Magentix2 platform.

Three interaction protocols specified by FIPA (Request, Query and the Contract-Net) have been implemented in the Magentix2 basic conversational protocol. For this purpose, a set of classes have been implemented, and they can be found into the `es.upv.dsic.gti-ia .architecture package`. Within all of the protocols implemented, agents can play both initiator and responder role. These roles implement different behaviors. The initiators are executed once, while responders are executed cyclically, so they will return to its initial state after reaching the final one. The set of classes in the `es.upv.dsic.gti-ia.architecture` package have been designed so that programmers do not need to deal with neither message sending nor protocol status monitoring. Thus, programmers only have to define what should be done in each state of the protocol and prepare messages before sending. The actions performed in each state are defined by handlers for initiator roles and preparers for responder roles.

- Handlers: A handler is a method which is executed when a specific protocol state is reached for agents playing initiator roles. Each protocol has a handler per each state it can

reach. Although there are default handlers (which do nothing) defined for each protocol, agent programmers can overload each handler with the functionality they require in each protocol state.

```
1          protected void handleAgree(ACLMessage agree) {
2                  logger.info("Good news");
3                  }
```

- Preparers: Preparers are similar to handlers but are executed when the agent plays the responder role in the protocol. Messages must be filled carefully because leaving a field empty can interrupt the entire protocol. Therefore, we encourage the use of the method `createReply()` included in ACLMessage messages. This method produces a new answer to the original message with the required fields covered, so only required ones need to be modified.

```
1  protected ACLMessage prepareResultNotification(ACLMessage inmsg,
2                                                 ACLMessage outmsg)
                                                       {
3
4              ACLMessage msg = inmsg.createReply();
5              return (msg);
6          }
```

3.3.2 How to implement a FIPA-Protocol

Following, some explanations of how the FIPA-Protocols proportionated by Magentix2 have been implemented are proportioned, in order to illustrate how new protocols could be implemented.

3.3.2.1 FIPA-Request

This protocol allows agents to request other agents to perform an action and it is identified in the protocol parameter of the message with the FIPA-request value. The messages exchanged are:

1. **Request:** which contents the request.

2. **Agree or Refuse:** when the agent accepts the request or rejects it respectively.
3. **Failure:** when the previous message was an Agree and an error happened during the process.
4. **Inform-done:** when the previous message was an Agree and the process ends successfully.
5. **Inform-result:** when the previous message was an Agree, the process ends successfully and there is also a result.

   The protocol early terminates if:

   - The initiator send to the responder a message explicitly CANCEL instead of the next initiator.
   - The responder responds negatively to REFUSE, NOT_UNDERSTOOD or FAILURE performative.

The following code shows how to implement a responder rol:

```
1          public class Responder extends FIPARequestResponder {
2
3          public Responder(QueueAgent agent) {
4          super(agent,new MessageTemplate(InteractionProtocol.
               FIPA_REQUEST));
5
6          }
7
8          protected ACLMessage prepareResponse(ACLMessage msg) {
9          protected ACLMessage prepareResultNotification
10                                      (ACLMessage inmsg,
                                           ACLMessage outmsg)    {}
```

The following code shows how to implement an initiator rol:

```
1
2
3          class Initiator extends FIPARequestInitiator {
4
5
6          public Initiator(QueueAgent a, ACLMessage msg) {
```

```
7                   super(a, msg);
8           }
9
10          protected void handleAgree(ACLMessage agree) {
11          protected void handleRefuse(ACLMessage refuse) {
12          protected void handleNotUnderstood(ACLMessage notUnderstood)
                {
13          protected void handleInform(ACLMessage inform) {
14          protected void handleFailure(ACLMessage failure) {
```

3.3.2.2 FIPA-Query

This protocol allows agents to request other agents: to query whether a particular proposition is true or false (query-if) and to query for some identified objects (query-ref). Depending on the type of request, the messages can be:

1. **Query-If or Query-Ref:** it contents the request.
2. **Agree:** when the agent accepts the request.
3. **Refuse:** in the case the agent rejects the request.
4. **Failure:** in the case an error occurred during the process
5. **Inform-T/F:** when the previous message was an Agree and the first message was a Query-If.
6. **Inform-Result:** when the previous message was an Agree and the first message was a Query-Ref.

3.3.2.3 FIPA-Contract-Net

The classes ContractNet implements the behaviour of the protocol of the same name, whose operation is: the initiator sends a proposal to several responders, then evaluates their answers and finally chooses the preferred one (or no one). The messages exchanged are:

1. **CFP (Call For Proposal):** it specifyes the action to carry out and, when it is appropriate, the conditions on the performance.

2. **Refuse:** when responders reject their participation.
3. **Not-Understood:** when there were failings in the communication.
4. **Propose:** when a responder makes proposal to the initiator.
5. **Reject-Proposal:** in the case the initiator evaluates a proposal and reject it
6. **Accept-Proposal:** when the initiator evaluates a proposal and accepts it, sending this type of message to accept them.
7. **Failure:** responder send this type of message when their proposals were accepted and something wrong happened.
8. **Inform-Done:** this messages is sended by responders when their proposals were accepted and the action was performed successfully.
9. **Inform-Results:** this message is sended by responders when their proposals were accepted and they need to inform about the results of the operation performed.

The initiator (ContractNetInitiator) has two main methods: the handlePropose method, which is called each time a response is received and the handleAllResponses method, which is called when all responses are received or the timeout is exceeded. The responder agent has the handleAcceptProposal and handleRejectProposal methods, which are called depending on whether the proposal was accepted or not, and their main characteristic is that both of them receive as input parameters all the messages exchanged by both agents so far.

3.3.3 Running QueueAgents

Once defined the protocol, we have to create a new instance and add it to the agent tasks (method addTask). In the initiator role, it is necessay to create and fill in the appropriate message for the desired protocol.

For example, for the protocol request:

```
1          ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
2          msg.setReceiver(new AgentID("HospitalAgent"));
3          msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
4          msg.setContent("accident to " + "10" + " km");
5          msg.setSender(this.getAid());
6
```

```
7          this.addTask(new FIPARequestInitiator(this,msg));
```

For the responder, it is needed to create a template with the desired protocol:

```
1          MessageTemplate template = new MessageTemplate(
2                                    InteractionProtocol.FIPA_REQUEST);
3
4          this.addTask(new FIPARequestResponder(this,template));
```

When a task is added , a new thread is created for the agent. Therefore, it is necessary to be careful which the main thread of the agent is not finished. For instance, a monitor can be used to wait for the completion of the roles:

```
1          import es.upv.dsic.gti_ia.architecture.Monitor;
2
3          private Monitor monitor = new Monitor();
4
5          protected void execute() {
6                                    .
7                                    .
8                                    .
9                  this.addTask(new FIPARequestInitiator(this, msg));
10                 monitor.waiting();
11         }
```

3.3.4 Examples

In the examples folder of the Magentix2 packages there are some basic interaction protocols examples:

- **Request:** this example follows the FIPA Request protocol. In this example two agents are created. One agent plays the responding role by simulating a hospital which listens emergency calls. The other agent simulates an accident witness. When the witness see an accident sends a help message to the hospital. The hospital checks if the accident is placed in the action area and if it could attend it.
- **ContractNet:** this is an example where the FIPA ContractNet protocol is followed. In this example, two types of agents are created. One type of agents plays the responding

role (dealers). The other type, a single agent, plays the initiator role (buyer). The buyer sends a purchase request to each dealer agent. Each dealer agent answers the bid according her preferences and waits for the buyer decision. The buyer chooses one offer and informs the specific dealer and also sends a reject message to the rest of the dealers.

3.4 Advanced conversational agents: CAgents

CAgents facilitate the use and management of conversations. CAgents use CFactories and CProcessors, these two components control ongoing conversation and create new ones. On the one hand, CFactories act as Interaction Protocols (IPs) and are in charge of creating new CProcessors. On the other hand, CProcessors act as instances of CFactories, that is conversations. CFactories have a graph made up of states and arcs. A graph specifies the sequence of actions that a conversation which is following that protocol has to take. Each state represents a specific action, and each arc represents a possible transition between two states. A collection of states (actions) has been defined:

- *Begin*: This state represent that the agent starts the conversation.
- *Final*: This state represent that the agent ends the conversation.
- *Send*: In this state the agent sends a message.
- *Wait*: When a agent reaches this state, the conversation halts until a message is assigned to the conversation. Then, according to the type of the arrived message, an specific subsequent *Receive* state is executed. The type of the message is defined by its header.
- *Receive*: This state must be preceded by a *Wait* state. In this state the agent receives a message. Each *Receive* state manages messages with a specific set of headers.

CFactories can be initiator or participant, the use of each type depends on the role the agent will play in the conversations. On the one hand, initiator CFactories start conversations when directed by agent's logic, i.e. they do not depend on external stimuli in order to start a new conversation. On the other hand, participant CFactories start a CProcessor when they receive a message with the appropriate message parameters. These parameters are specified using a message filter associated to the participant CFactory.

The transition between two states occurs when the agent receives or sends a message related to that specific conversation. CProcessors are in charge of making these transitions as well as executing the actions of each state of the conversation.

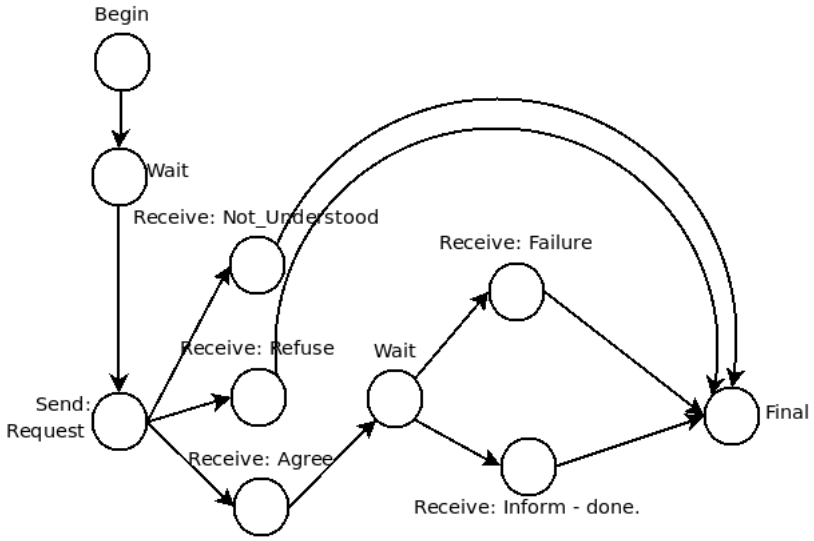

Figure 3.5: CFactory for FIPA Request Interaction Protocol for the initiator agent

When a CProcessor is created, it has a copy of the graph specified in the CFactory that created the CProcessor. During the conversation, the CProcessor will execute the actions of the state the conversation is currently at, and it will change the state of the conversation as new messages are sent and received. As each CProcessor has its own graph, an ongoing conversation can be dynamically modified without affecting the IP the conversation is following or other ongoing conversations which follows that IP.

In figure 3.5 an example of an IP transformed into a graph associated to a CFactory is shown. This IP corresponds to the *FIPA Request Interaction Protocol* [FIPA, 2002] for the initiator role.

In figure 3.6 a global view of a CAgent is shown. In this figure the agent shown has three CFactories, two of them are participant and the third one is initiator. At the same time the agent has two ongoing CProcessors that manage two conversations “Conv1” and “Conv2”. The first one has been created by the initiator CFactory 1, the other one by the participant CFactory 1. The initiator CFactory 1 created the CProcessor managing “Conv1” because the execution of the agent dictates that, instead the “Conv2” was created by the participant CFactory 1 because the agent received an *inform* message. From this moment on, every message with the message parameter *conversation_id* set to “Conv1” will be automatically assigned to the CProcessor managing that conversation. The same will occur with “Conv2” messages. If in the future the agent receives a *request* message, the participant CFactory 2 will create a new CProcessor which will manage the new conversation. Other possibility is that the agent receives an *in-*

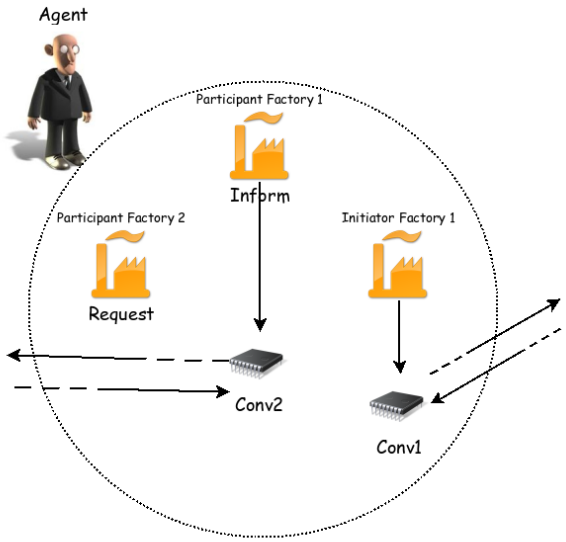

Figure 3.6: Global view of a CAgent

form message with an unknown *conversation_id*. In that case, the participant CFactory 1 will create a new CProcessor and two conversations which follows the same IP. This conversations will be managed simultaneously, the previous "Conv2" and the new conversation. For more information about CAgents please refer to [Fogués et al., 2010].

In the next sections some examples of CAgents are explained. All the code of these examples are available in the examples directory of Magentix2 platform.

3.4.1 "Hello World" CAgent

The code shown in this section corresponds to the code located in examples/src/myFirstCAgent/HelloWorldAgent.java. In the code below, the method `execution` is where the user has to implement the main code of the agent. In this "hello world" agent the main behaviour of the agent is just to say "hello world". The method `finalize` is executed when the agent is just about to finish its execution, thus, here is where the user has to implement ending actions of the agent.

```
1 class HelloWorldAgentClass extends CAgent {
2    public HelloWorldAgentClass(AgentID aid) throws Exception {
3       super(aid);
4    }
```

```
5
6       // The platform starts a conversation with each agent that has
            been just created
7       // by sending it a welcome message. This sending creates the
            first CProcessor
8       // of the agent. In order to manage this message the user must
            implement
9       // the execution method defined by the class CAgent, this method
            will
10      // be executed by the first CProcessor.
11      // It is also in this method where all other actions and
            behaviours of the
12      // agent has to be implemented
13
14      protected void execution(CProcessor myProcessor, ACLMessage
            welcomeMessage) {
15          System.out.println(myProcessor.getMyAgent().getName()
16              + ": the welcome message is " + welcomeMessage.getContent()
                    );
17          System.out.println(myProcessor.getMyAgent().getName()
18              + ":  inevitably I have to say hello world");
19          // ShutdownAgent method initialize the process which will
                finalize the
20          // active conversations of the agent. When this process ends,
                the platform
21          // sends a finalize message to the agent.
22          myProcessor.ShutdownAgent();
23      }
24
25      // In order to manage the finalization message, the user has to
26      // implement the finalize method defined by the CAgent class.
27      protected void finalize(CProcessor myProcessor, ACLMessage
            finalizeMessage) {
28          System.out.println(myProcessor.getMyAgent().getName()
29              + ": the finalize message is " + finalizeMessage.getContent
                    ());
30      }
31  }
```

3.4.2 Creating a CFactory and its CProcessor

The code shown in this section corresponds to the code located in examples/src/myFirstCProcessorFactories/Harry.java. The CFactory shown here corresponds to the one shown in figure 3.7.

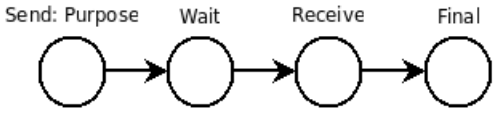

Figure 3.7: myFirstCProcessorFactories example

In order to create a CProcessor, first it is necessary to define the states and the transitions that compose the graph associated to the CProcessor. This can be done in the execution method of the agent. Every conversation starts with a *begin* state called "BEGIN". In the first line of the code below, a new CFactory is created. The parameters passed to the constructor are: the CFactory's name; a message filter that will determine which messages will make this CFactory to start a new CProcessor acting as a conversation; how many CProcessors this CFactory can manage simultaneously; and finally, a reference to the agent which owns the CFactory. Specifically, this CFactory (called "TALK") will create the CProcessor when the agent receives messages with *propose* performative and it can only manage one CProcessor at a time. When a CFactory is created, it has already a CProcessor template predefined by default. This template is from new instances of CProcessor will be created. This default CProcessor template has to be modified by the user in order to create its own IPs. The *begin* state of the CProcessor template is defined by default. It starts creating a new IP modifying this predefined *begin* state. In line 8, this *begin* state is got from the template. In the lines below, a new method for this state is created. This method will be executed when the conversation reaches this *begin* state. Therefore this method will be executed when the conversation starts. Every method in a conversation state has to return the name of the state the CProcessor will travel to. In this examples the begin state always travels to the state called "PROPOSE". In line 16, the method just defined is associated to the begin state.

Starting at line 18, a new send *state* is defined called "PROPOSE". Once the state is created, it is necessary to define a method for this state. A method for a *send* state has to return the name of the next state and also, has to assign a value to the variable `messageToSend` which is passed to the method as an argument. As can be seen, the message is going to be sent to agent Sally and the objective of the message is to propose Sally to go to the cinema.

Once the state "PROPOSE" has been created, registered and all the transitions to it have been

added. In line 31 a new *wait* state called "WAIT" is created. This type of states does not need a method to execute. During its creation, it is necessary to define the name of the state and the timeout, in this case 1000ms. It is possible to define a *wait* state that will wait an unlimited amount of time until a message is received if the timeout is set to 0. A transition from "PROPOSE" state to this *wait* state is added.

After a *wait* state it is mandatory to define one or more *receive* states. In this example it is only defined one *receive* state. A *receive* state needs a method to execute and a message filter. This filter specifies which messages can be managed by this *receive* state. In this example, the filter is set to null, therefore this *receive* state accepts any message. The code defining the *receive* state named "RECEIVE" starts at line 34.

Finally it is defined the *final* state of the protocol at line 46. Every protocol has to finish in a *final* state. *Final* states methods can use the *responseMessage* variable passed as argument as a mean to return a value of the conversation. Once the method that will be executed by this state is defined, the transitions from other states to this one are added and the state registered in the CProcessor.

There is only one thing left to finish the CAgent, it is necessary to add the CFactory to the set of CFactories of the agent. It is possible to add a CFactory as an initiator one or as a pariticipant one, depending on the role the agents will play on the conversations generated by the CFactory. In this case, the CFactory is added as an initiator one. Once the CFactory has been added, it can create new CProcessors. In line 63, the agent starts a synchronous conversation, this is, the execution of the agent will halt until the conversation ends. The method *startSyncConversation* is used in order to start a synchronous conversation. This method requires the name of the initiator CFactory which will create a new conversation as a parameter. The result of the conversation is stored in the variable response. In the last line of the code the result of the conversation is shown on the console.

```
1 filter = new MessageFilter("performative = PROPOSE");
2 CFactory talk = new CFactory("TALK", filter, 1,this);
3
4 // A CProcessor always starts in the predefined state BEGIN.
5 // We have to associate this state with a method that will be
6 // executed at the beginning of the conversation.
7
8 BeginState BEGIN = (BeginState) talk.cProcessorTemplate().getState("
     BEGIN");
9 class BEGIN_Method implements BeginStateMethod {
```

```
10      public String run(CProcessor myProcessor, ACLMessage msg) {
11          // In this example there is nothing more to do than continue
12          // to the next state which will send the message.
13          return "PROPOSE";
14      };
15  }
16  BEGIN.setMethod(new BEGIN_Method());
17
18  SendState PROPOSE = new SendState("PROPOSE");
19  class PROPOSE_Method implements SendStateMethod {
20      public String run(CProcessor myProcessor, ACLMessage
            messageToSend) {
21          messageToSend.setContent("Would you like to come with me to
                the cinema?");
22          messageToSend.setReceiver(new AgentID("Sally"));
23          messageToSend.setSender(myProcessor.getMyAgent().getAid());
24          return "WAIT";
25      }
26  }
27  PROPOSE.setMethod(new PROPOSE_Method());
28  talk.cProcessorTemplate().registerState(PROPOSE);
29  talk.cProcessorTemplate().addTransition(BEGIN, PROPOSE);
30
31  talk.cProcessorTemplate().registerState(new WaitState("WAIT", 1000))
        ;
32  talk.cProcessorTemplate().addTransition(PROPOSE, WAIT);
33
34  ReceiveState RECEIVE = new ReceiveState("RECEIVE");
35  class RECEIVE_Method implements ReceiveStateMethod {
36      public String run(CProcessor myProcessor, ACLMessage
            messageReceived) {
37          return "FINAL";
38      }
39  }
40
41  RECEIVE.setAcceptFilter(null); // null -> accept any message
42  RECEIVE.setMethod(new RECEIVE_Method());
43  talk.cProcessorTemplate().registerState(RECEIVE);
44  talk.cProcessorTemplate().addTransition(WAIT, RECEIVE);
45
```

```
46 FinalState FINAL = new FinalState("FINAL");
47 class FINAL_Method implements FinalStateMethod {
48    public void run(CProcessor myProcessor, ACLMessage
          responseMessage) {
49       messageToSend.copyFromAsTemplate(myProcessor.
             getLastReceivedMessage());
50       myProcessor.ShutdownAgent();
51    }
52 }
53 FINAL.setMethod(new FINAL_Method());
54
55 talk.cProcessorTemplate().registerState(FINAL);
56 talk.cProcessorTemplate().addTransition(RECEIVE, FINAL);
57 talk.cProcessorTemplate().addTransition(PROPOSE, FINAL);
58
59 // The template processor is ready. We add the factory, in this case
       as a participant one
60 this.addFactoryAsInitiator(talk);
61
62 // Finally Harry starts the conversation.
63 ACLMessage response = this.startSyncConversation("TALK");
64
65 System.out.println(this.getAid().name + " : Sally tell me "
66 + response.getPerformative() + " " + response.getContent());
```

3.4.3 Using a CFactory Template

Defining a CFactory and its CProcessor template can be a laborious task. In order to facilitate this, a set of CFactories templates are provided in Magentix2. At the moment the templates are:

- FIPA Request
- FIPA Contract-net
- FIPA Recruiting

All of this templates are available in both versions, initiator and participant.

The source code of the examples shown in this sections are accessible at the folder examples/src/requestFactory.

A CFactory template is a java class that has already defined the states and the transitions of the CProcessor template. A template can be modified in order to adapt it to any specification. Some templates have abstract methods that are necessary to implement by the users. Others methods offer a default behaviour that can be modified if needed. As an example, it is shown how to adapt the FIPA Request Initiator template provided in Magentix2 to a scenario where one agent (Harry) asks another agent (Sally) for her phone number.

In this case it is necessary to create a new class that extends the template FIPA_REQUEST_Initiator (lines 6-12). This class has an abstract method that is mandatory to implement, the `doInform` method (lines 7-11). This method is executed when the initiator receives the results of what it requested. All the other methods for the other states have a default behaviour that can be modified, in this example it is not necessary to do so.

The message that contains the request is created (lines 17-21). Afterwards a CFactory from the template is created (line 29). During the creation, it is required to specify the name of the new CFactory, the request message, the agent owner of the CFactory and the time in milliseconds that the agent will wait for the inform or failure message. Once the CFactory template is defined, it is possible to create a new instance from it.

Finally, the just created CFactory is added to the agent (line 30), and in the last line of the code, a synchronous conversation from this CFactory is started.

```
1  // In this example the agent is going to act as the initiator in the
2  // REQUEST protocol defined by FIPA.
3  // In order to do so, she has to extend the class
       FIPA_REQUEST_Initiator
4  // implementing the method that receives results of the request (
       doInform)
5
6  class myFIPA_REQUEST extends FIPA_REQUEST_Initiator {
7     protected void doInform(CProcessor myProcessor, ACLMessage msg) {
8        System.out.println(myProcessor.getMyAgent().getName() + ": "
9        + msg.getSender().name + " informs me "
10       + msg.getContent());
11    }
12 }
13
```

```
14 // We create the message that will be sent in the doRequest method
15 // of the conversation
16
17 msg = new ACLMessage(ACLMessage.REQUEST);
18 msg.setReceiver(new AgentID("Sally"));
19 msg.setContent("May you give me your phone number?");
20 msg.setProtocol("fipa-request");
21 msg.setSender(getAid());
22
23 // The agent creates the CFactory that creates processors that
       initiate
24 // REQUEST protocol conversations. In this
25 // example the CFactory gets the name "TALK", we don't add any
26 // additional message acceptance criterion other than the required
27 // by the REQUEST protocol
28
29 CFactory talk = new myFIPA_REQUEST().newFactory("TALK", msg,1, this,
       5000);
30 this.addFactoryAsInitiator(talk);
31
32 // finally the new conversation starts. Because it is synchronous,
33 // the current interaction halts until the new conversation ends.
34 this.startSyncConversation("TALK");
```

CFactory templates are useful for reusing code. It is possible to create templates of other IP or modify the existing ones in order to adapt them.

3.4.4 Creating a CFactory Template

This section explains how to implement a new CFactory template. In the following example we are going to implement a template for the CFactory shown in section 3.4.2.

First the states are defined, it is in this moment when it is possible to define default methods and choose which methods are abstract and therefore, mandatory to implement for the user. The definition of the states is shown below.

```
1 public abstract class MyTemplate {
2    //We can define a set of static values for referencig sates
3    public static String BEGIN = "BEGIN";
```

```
4     public static String PROPOSE = "PROPOSE";
5     public static String WAIT = "WAIT";
6     public static String RECEIVE = "RECEIVE";
7     public static String RECEIVE = "FINAL";
8
9     protected void doBegin(CProcessor myProcessor, ACLMessage msg) {
10       System.out.println("This is the begin state");
11    }
12
13    class BEGIN_Method implements BeginStateMethod {
14       public String run(CProcessor myProcessor, ACLMessage msg) {
15          doBegin(myProcessor, msg);
16          return PROPOSE;
17       };
18    }
19
20    // We want the user to implement his/her method here
21    protected abstract void doPropose(CProcessor myProcessor,
          ACLMessage messageToSend);
22
23    class PROPOSE_Method implements SendStateMethod {
24       public String run(CProcessor myProcessor, ACLMessage
             messageToSend) {
25          doPropose(myProcessor, messageToSend);
26          // This IP is so simple and hasn't any choices. If it had
27          // then we can set the return type of the doRequest method
28          // to String and use it as a return value for this method
29          return WAIT;
30       }
31    }
32    // We want the user to implement his/her method here
33    protected abstract void doReceive(CProcessor myProcessor,
          ACLMessage msg);
34
35    class RECEIVE_Method implements ReceiveStateMethod {
36       public String run(CProcessor myProcessor, ACLMessage
             messageReceived) {
37          doReceive(myProcessor, messageReceived);
38          return FINAL;
39       }
```

```
40      }
41
42      protected void doFinal(CProcessor myProcessor, ACLMessage
            messageToSend) {
43          messageToSend = myProcessor.getLastSentMessage();
44      }
45
46      class FINAL_Method implements FinalStateMethod {
47          public void run(CProcessor myProcessor, ACLMessage
                messageToSend) {
48              doFinal(myProcessor, messageToSend);
49          }
50      }
```

Once all the states are defined, the next step is to create a method that returns a new CFactory. In this method, new states are created and the methods defined before are assigned to them. The transitions between the states are also defined in this method. Depending on the IP, some parameters will be needed. In this case, it is only necessary to specify the name of the CFactory, the maximum number of simultaneous conversations, the agent who owns the CFactory and the timeout for the wait state. The code of this method is shown below.

```
1       public CFactory newFactory(String name, int
            availableConversations, CAgent myAgent, long timeout) {
2           CFactory theFactory = new CFactory(name, null,
                availableConversations, myAgent);
3           // Processor template setup
4           CProcessor processor = theFactory.cProcessorTemplate();
5
6           // BEGIN State
7           BeginState BEGIN = (BeginState) processor.getState("BEGIN");
8           BEGIN.setMethod(new BEGIN_Method());
9
10          // PROPOSE State
11          SendState PROPOSE = new SendState("PROPOSE");
12
13          PROPOSE.setMethod(new PROPOSE_Method());
14          processor.registerState(PROPOSE);
15          processor.addTransition(BEGIN, PROPOSE);
16
```

```
17          // WAIT State
18          WaitState WAIT = new WaitState("WAIT", timeout);
19          processor.registerState(WAIT);
20          processor.addTransition(PROPOSE, WAIT);
21
22          // RECEIVE State
23
24          ReceiveState RECEIVE = new ReceiveState("RECEIVE");
25          RECEIVE.setMethod(new RECEIVE_Method());
26          filter = new MessageFilter(""); //accept any message
27          RECEIVE.setAcceptFilter(filter);
28          processor.registerState(RECEIVE);
29          processor.addTransition(WAITE, RECEIVE);
30
31          FinalState FINAL = new FinalState("FINAL");
32          FINAL.setMethod(new FINAL_Method());
33          processor.registerState(FINAL);
34          processor.addTransition(RECEIVE, FINAL);
35
36          return theFactory;
37      }
38  };
```

How to use a CFactory template is shown in the previous section 3.4.3. For this specific template the instructions are the same.

3.5 BDI Agents: JasonAgents

Jason[Bordini et al., 2005] is an interpreter for an extended version of AgentSpeak(L)[Rao, 1996] and implements the operational semantics of that language. It has been developed by Jomi F. Hübner and Rafael H. Bordini. Jason has been integrated in Magentix2 platform, therefore we can program agents in AgentSpeak and run them on Magentix2 platform. For examples and demos of how to program in AgentSpeak(L), please refer to the webpage of the Jason project: `http://jason.sourceforge.net/Jason/Jason.html`.

Magentix2 integrates Jason providing two classes: JasonAgent and MagentixAgArch. MagentixAgArch manages the AgentSpeak(L) interpreter, the reasoning cycle of the agent, and how the agent acts and perceives to/from the environment. The JasonAgent class acts as a link

between the AgentSpeak(L) interpreter and the platform. Both classes can be modified and adapted to the desired needs, but usually, only MagentixAgArch would need to be modified in order to add external actions to the agent (external actions are actions which affect the agent environment).

The code of how to create and execute a basic JasonAgent is shown below.

```
1 MagentixAgArch arch = new MagentixAgArch();
2 JasonAgent agent = new JasonAgent(new AgentID("bob"), "./src/test/
      java/jasonTest_1/demo.asl", arch);
3 agent.start();
```

In the code shown above, first an instance of MagentixAgArch class called “arch” is created, then a JasonAgent called “agent” is created, in order to create a JasonAgent it is necessary to specify its AgentID, the file with the AgentSpeak(L) program that the interpreter will execute and the agent architecture the agent will use, in this case, a standard MagentixAgArch. Finally, start the execution of the agent starts.

It is possible to modify the agent architecture, in the following example the MagentixAgArch default architecture will not be used, instead a new one is created, which extends from MagentixAgArch.

```
1  public class SimpleArchitecture extends MagentixAgArch {
2
3  // this method just adds some perception to the agent
4  @Override
5     public List<Literal> perceive() {
6        List<Literal> l = new ArrayList<Literal>();
7        l.add(Literal.parseLiteral("x(10)"));
8        return l;
9     }
10 }
```

This new architecture, called SimpleArchitecture, just adds a perception to the agent overriding the method `perceive` of the MagentixAgArch class. As said before, usually the architecture is modified in order to add new actions to the agent, this could be done just overriding the method `act(ActionExec action, List<ActionExec> feedback)` of the architecture. This method receives an action as an argument and a list of action executions called feedback. In the code below, it is shown how to manage a new external action of the agent

called “doAction”.

```
@Override
public void act(ActionExec action, List<ActionExec> feedback)
{
getTS().getLogger().info("Agent " + getAgName() +" is doing: " +
    action.getActionTerm());
   if(action.getActionTerm().equals("doAction")){
      //perform the action
      //set the result, for example always true
      action.setResult(true);
      //add the executed action to the list of action executions
      feedback.add(action);
   }
}
```

The code in AgentSpeak(L) is written in a different file, the path to the file is passed as an argument to the constructor of the JasonAgent class. A sample code of an AgentSpeak(L) program is shown below.

```
vl(1).
vl(2).

+vl(X)[source(Ag)]
   :  Ag \== self
   <- .print("Received tell ",vl(X)," from ", Ag).

+!goto(X,Y)[source(Ag)] : true
   <- .println("Received achieve ",goto(X,Y)," from ", Ag).

+?t2(X) : vl(Y) <- X = 10 + Y.

+!kqml_received(Sender, askOne, fullname, ReplyWith) : true
   <- .send(Sender,tell,"Maria dos Santos", ReplyWith). // send the
      answer
```

3.6 Launching agents with security enabled

In order to ensure the integrity, confidentiality, privacity, no repudiation and mutual authentication of the agent comunications, a security module for Magentix2 platform was implemented. If the security module is enabled in the platform in which will connect, setting the public key infraestructure and configuring the security properties are necessary.

The basic elements of this public key infrastructure are:

- **Asymetrical key-pair:** This is used to sign and encrypt information messages.
- **X.509 Certificate:** This is a self-signed certificate with the information of the user, which guarantees the user identity.
- **keystore:** It is a file that contains the X.509 User Certificates, which is needed to connect with the platform. The keystore is protected by a password, hence this file must never be disclosed to others.
- **Trustore:** It is a file that contains a list of trusted parties. The trusted parties will be the Magentix2 platform to which the user connect.

3.6.1 Creating key-pair

If a key-pair and a certificate issued by a public entity (must be trusted for the platform) is available, the explanation continues in the section 3.6.3, if not, the next steps must be followed.

For generating [5] the key pair, this command must be executed in a linux terminal:

```
$ keytool -genkey -alias alias_name -keyalg RSA \
-dname "CN=alias_name,O=Magentix" \
-storepass mysecretpassword \
-keypass mysecretpassword \
-keystore keystore.jks
```

The result of this command is a new keystore with an asymmetrical key-pair user and X.509 Certificate.

[5] For this purpose the command keytool included in Java Sun JDK is used .

3.6.2 Exporting User Certificate

For one mutual authentication between the user and platform, a self-signed certificate that contains the user public key must be exported.

```
$ keytool -export -keystore keystore \
   -storepass mysecretpassword \
 -alias alias_name -file user.crt
```

The file user.crt will be transfered to the Magentix2 platform administrator by http, ftp, e-mail, etc..., in order that he adds in it his trusted store.

3.6.3 Importing MMS Certificate

This command append the Magentix Manager Service (MMS) trusted certificate in the keystore:

```
$ keytool -import -trustcacerts -noprompt -alias MMS \
-file mms.crt -storepass mysecretpassword  -keystore keystore
```

The file mms.crt contains the MMS public key, to obtain this file, please contact with the Magentix2 administrator.

3.6.4 Creating and importing truststore

If the MagentixCA self-signed certificate is not included in the truststore, the connections with the platform will be rejected.

The following command creates a truststore and appends the MagentixCA trusted certificate:

```
$ keytool -import -trustcacerts -noprompt -alias MagentixCA \
-file rootca.crt -storepass mysecretpassword \
-keystore truststore.jks
```

The root.crt file contains the MagentixCA public key, to obtain this file, please contact with the Magentix2 platform administrator.

3.6.5 Configuration

The security parameters must be indicated inside the securityUser.properties file and the configuration of the broker connection inside the Settings.xml. These files are inside the configuration directory.

- **securityUser.properties**

  The securityUser.Properties is divided into four sections. In the first section, the keystore and truststore is configured, hence the path and password must be specified, as it can be shown in the following example (Figure 3.8).

```
#<!-- User Security Properties -->
# set the base path for accessing keystore user
KeyStorePath=/full_path/keystore.jks

#set the password to keystore
KeyStorePassword=password

# set the base path for accessing truststore user
TrustStorePath=/full_path/truststore.jks

#set the password to truststore
TrustStorePassword=password
```

Figure 3.8: First section of the securityUser.properties file

In the second section it is indicated if the user certificate used has been issued by a public certificate autorithy (in this case the parametre type is *others*) or have been created by the user (the type is *own*).

If the type selected is *others*, then it is required to configuring the parameters that will be find below the *type* parameter. Again, an example is shown below (Figure 3.9).

In the section three (Figure 3.10), the MMS direction path is indicated. For this purpose it is necessary to consult with the Magentix2 administrator to provide the direction ([protocol] + [host] + [port] + [path]) where the MMS service is deployed.

The purpose of last section is to indicate the user alias certificate (Figure 3.11).

```
#set the type of certificates (own or others (FNMT, ACCV,
# ...))
type=own
#set the base path for access to correct keystore
othersPath=/full_path/keystore_fnmt.p12
#set the pin of the smartcard or password of your keystore
othersPin=password
#set the type of the keystore
othersType=PKCS12
```

Figure 3.9: Second section of the securityUser.properties file

```
# set the connection protocol to be used to access services
protocol=http
# set the name of the service host
host=localhost
# set the port for accessing the services
port=8080
# set the base path for accessing to services
path=/MMS/services/MMS
```

Figure 3.10: Third section of the securityUser.properties file

```
#set the alias to user certificate. This alias is
#a PrivateKeyEntry.
alias=[User_alias]
```

Figure 3.11: Fourth section of the securityUser.properties file

If the name of the alias certificate is not know, then it can be consulted with the following command:

```
$ keytool -list -keystore keystore.jks
```

In the case the certificate store is not a jks type:

```
$ keytool -list -storetype pkcs12 -keystore keystore_fnmt.p12
```

The alias will be displayed in the first place of line, for example in the following case the alias is *alice* (Figure 3.12).

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 5 entries

alice, 27-sep-2010, PrivateKeyEntry,
Hash of certificate (MD5):
 13:D2:8F:3A:F3:2B:C9:D9:CC:6B:A8:C0:DB:7D:DA:FA
```

Figure 3.12: Result of the *Keytool* command

```
    <!-- Properties qpid broker -->
    <entry key="host">localhost</entry>
    <entry key="port">5671</entry>
    <entry key="vhost">test</entry>
    <entry key="user">guest</entry>
    <entry key="pass">guest</entry>
    <entry key="ssl">true</entry>
    <!-- Secure qpid properties -->
    <entry key="secureMode">true</entry>
    <entry key="saslMechs">EXTERNAL</entry>
```

Figure 3.13: An example of the Settings.xml file

- **Settings.xml** The connection parameters must be specified inside the Settings.xml file. The Figure 3.13 shows an example of the content of this file. Note that the *port* parameter is where the broker Qpid is listening to the ssl connections and EXTERNAL is the method by default.

3.6.6 Running

In the secure mode, agent connection with the broker is formed automatically in the agent constructor. Therefore, in the main program, the method connect (explained in section 3.1.4) is not necessary. Thus, it is only required to complete carefully the previous configurations and import the required libraries [6] in the Java project build path.

(Please note that the platform do not allow agents with the same name and it is case sensitive.)

[6] The libraries are available in lib/security/ directory

3.6.7 Problems

The following exceptions can arise when security is used and correct configuration has not be done.

1. **(WSSecurityEngine: Callback supplied no password for: mms).** The MMS self-signed certificate has not been imported correctly in keystore.
2. **org.apache.axis2.AxisFault: Error in signature with X509Token.** The alias parameter of the securityUser.properties file has not correctly specified.
3. **General security error (No certificates were found for decryption (KeyId)).** The MMS self-signed certificate has not been imported correctly in the keystore.
4. **org.apache.axis2.AxisFault: The certificate used for the signature is not trusted.** The User certificate with the user public key has not been added in the truststore of magentix administrator, it is necessary to check that the user certificate have been exported and transfered correctly or contact with the magentix administrator. In administrator mode, section 5.6.6 explains how to import user certificate.

3.7 Tracing Service

This section describes the Tracing Service Support available in Magentix2. This Tracing Service Support allows agents in a Multiagent System (MAS) share information in an indirect way by means of trace events.

Before describing the API provided by Magentix2 to share tracing information, the following sections will present TRAMMAS, the trace model which was followed to incorporate event tracing facilities to Magentix2.

3.7.1 Trace Model and Features

The trace event model incorporated to Magentix is described in detail in [Búrdalo et al., 2010]. This section will comment briefly the main characteristics of the model and those of its features which have been incorporated to the platform.

The trace model described in [Búrdalo et al., 2010] offers any entity in the system (active or passive) the possibility to share information, in the form of trace events, with other entities in

the system. From the point of view of the trace model, entities in the system are seen as *tracing entities*; this is, entities which are able to generate and/or receive trace events. Trace events are offered by tracing entities as different *tracing services*, which can be requested by interested tracing entities when they want to receive these trace events.

3.7.1.1 Supported Features

The present version of Magentix2 does not support all of the tracing features considered in the model. Those which are not yet supported will be incorporated in future versions of the platform.

- Tracing Entities

  Although the model considers not only agents, but also non-agent entities or aggregations, event tracing facilities incorporated to the present version of Magentix2 platform only consider agents. Artifacts and aggregations will be included in future versions of the platform.

- Publication and Subscription

  Publication and unpublication of trace events is completely supported in the present version of the platform; so, tracing entities can dynamically (at run time) publish the event types which they can throw and unpublish them when they do not want to share that information anymore.

  In the same way, tracing entities can dynamically subscribe to trace events in which they are interested and unsubscribe from them whenever they do not want to receive these trace events anymore. However, although the model lets tracing entities filtering trace events according to many parameters, the present version of Magentix2 only allows agents to filter trace events according to the event type and the agent which threw the event.

  Tracing services composition has also been postponed for later platform releases and thus, it is not supported in the present version of Magentix2.

- Authorization

  Security issues addressed by the model have also been postponed for future versions of the platform and they are not considered in this version. Thus, any agent in the system can request trace events from any other agent without needing any direct or indirect authorization.

3.7.2 Trace Event

Trace events are represented in the platform as instances of the class `es.upv.dsic.gti_ia.core.TraceEvent`:

```
1 public class TraceEvent implements Serializable {
2    private String tService;
3    private long timestamp;
4    private TracingEntity originEntity;
5    private String content;
6 }
```

The different attributes of this class indentify the *tracing service* to which the trace event belongs, the *tracing entity* which originated the trace event and the time at which it was produced, expressed as the amount of milliseconds from 1 Jan 1970 at 00:00:00 (*Epoch*). This information can be complemented when needed with extra data stored in the `content` attribute.

Trace event instances are created by invoking the constructor of the class. Details of the parameters are explained in Table 3.1. As it can be seen, the `timestamp` attribute of the trace event is not stablished by any parameter of the constructor. This is because the `timestamp` of each trace event is internally set to the time at which the constructor of the class was invoked.

Table 3.1: `TraceEvent` class constructor parameters

`TraceEvent(String tService, AgentID originAid, String content)`	
`tService:`	String identifying the tracing service to which the trace event is associated.
`timestamp:`	Time at which the trace event was generated. The time is expresed in milliseconds from 1 Jan 1970 at 00:00:00 (*Epoch*).
`originAid:`	`AgentID` of the agent which originated the trace event. Internally, the constructor converts `originAid` to a `TracingEntity`.
`content:`	Any extra data which could be necessary to complement or understand the meaning of the trace event. This attribute can be empty.

Once created, trace events can be sent by means of the public method `sendTraceEvent(TraceEvent tEvent)`, included in the class `es.upv.dsic.gti_ia.core.BaseAgent`. This method does not create the instance of the trace event and thus, it is necessary to invoke the `TraceEvent` constructor before sending it. For instance, if an agent wants to create a trace event for a service called `SIMPLE_SERVICE`, it could be made in this way:

```
1 /* Creation of the trace event */
2 TraceEvent tEvent = new TraceEvent("SIMPLE_SERVICE", this.getAid(),
      "This is a simple trace event, provided by a simple tracing
      service");
3
4 /* Sending the event */
5 send(tEvent);
```

The class `es.upv.dsic.gti_ia.core.BaseAgent` includes a trace event handler method, similar to the ACL message handler: `onTraceEvent(TraceEvent tEvent)`. This method executes automatically each time a trace event is received by the agent; however, it is empty. This means that the developer has to write the source code to process trace events.

The source code in Section 3.7.6 (lines 59 to 79) shows an example of trace event handler. In this case, the received trace event is processed attending to its tracing service.

3.7.3 Tracing Services

Agents which are interested in sharing their trace events, offer them as *tracing services*. Agents publish their available tracing services and other agents can request those tracing services in which they are interested. As a consequence, only those trace events which have been requested by an agent in the system are traced and agents only receive those trace events which they have previously requested. In this way, trace event traffic is reduced and agents do not have to process trace events which they are not interested in.

In order to register available tracing entities and services, as well as to manage subscriptions to tracing services, the platform must have a *Trace Manager* running. In Magentix2, the Trace Manager is a single agent (the `TraceManager` class inherits from `BaseAgent`), which must be running in a host of the platform; however, the trace manager is intended to be a distributed entity in future versions of the platform.

The trace manager can be launched in any host where Magentix2 is running by invoking the corresponding constructor method `TraceManager(AgentID tmAid)`. An example of how to launch the Trace Manager can be found in the source code of the example in Section 3.7.8, in line number 29 of the main application source code:

```
1 TraceManager tm = new TraceManager(new AgentID("tm"));
```

Agents communicate with the Trace Manager using the methods available in the class `es.upv.dsic.gti_ia.trace.TraceInteract`. These methods internally send an ACL message to the Trace Manager. This message will be processed by the Trace Manager, which responds to these requests via ACL messages to requester agents. When the request sent to the Trace Manager, an error ACL message is sent to the requester tracing entity with an error code, in order to let the tracing entity know the reason why the request was rejected. These error codes are available in the class `es.upv.dsic.gti_ia.trace.TraceError` and their meaning is described in Table 3.2.

Table 3.2: Trace Manager error codes

TRACE_ERROR:	Undefined trace error.
ENTITY_NOT_FOUND:	Tracing entity not present in the system.
PROVIDER_NOT_FOUND:	Provider is not offering the tracing service.
SERVICE_NOT_FOUND:	Tracing service not offered by any tracing entity in the system.
SUBSCRIPTION_NOT_FOUND:	Subscription to the tracing service not found.
ENTITY_DUPLICATE:	Tracing entity already present in the system.
SERVICE_DUPLICATE:	Tracing service already offered by the tracing entity.
SUBSCRIPTION_DUPLICATE:	Subscription already exists.
BAD_ENTITY:	Tracing enity not correct.
BAD_SERVICE:	Tracing service not correct.
PUBLISH_ERROR:	Impossible to publish the tracing service.
UNPUBLISH_ERROR:	Impossible to unpublish the tracing service.
SUBSCRIPTION_ERROR:	Impossible to subscribe to the tracing service.
UNSUBSCRIPTION_ERROR:	Impossible to unsubscribe from the tracing service.
AUTHORIZATION_ERROR:	Unauthorized to do so.

Some of the methods available in `es.upv.dsic.gti_ia.trace.TraceInteract` assume that a default trace manager called `tm` is running, while others allow specifying the Trace Manager to which requests are to be directed.

Actions related to tracing services can be classified in three main groups, which are explained in more detail in the following sections: Publication/Unpublication of tracing services (Section 3.7.3.1), subscription/unsubscription to/from tracing services (Section 3.7.3.2) and listing of tracing entities or services (Section 3.7.3.3).

3.7.3.1 Tracing service publication

In order to publish and unpublish tracing services, agents have to use respectively the methods `publishTracingService` and `unpublishTracingService`. These methods are described in more detail in Table 3.3.

Table 3.3: Tracing service publication and unpublication methods

publishTracingService(`BaseAgent applicantAgent, String serviceName, String description`**)**
publishTracingService(`AgentID tms_aid, BaseAgent applicantAgent, String serviceName, String description`**)**
unpublishTracingService(`BaseAgent applicantAgent, String serviceName`**)**
unpublishTracingService(`AgentID tms_aid, BaseAgent applicantAgent, String serviceName`**)**

tms_aid:	`AgentID` of the Trace Manager which will process the request. If not specified, the request is directed to the default Trace Manager, `tm`.
applicantAgent:	Agent which is publishing the tracing service.
serviceName:	String identifying the tracing service which is being published or unpublished.
description:	Human readable description of the tracing service which is being published.

An example of tracing service publication can be:

```
publishTracingService(this, "TracingService_1", "An example of
    tracing service");
```

where the running agent (`this`) publishes a tracing service called *TracingService_1* which is described as *An example of tracing service*. Once the running agent is done sharing these trace events, it can unpublish the tracing service by invoking the corresponding method referring to the already published tracing service:

```
unpublishTracingService(this, "TracingService_1");
```

3.7.3.2 Tracing service subscription

Before receiving any trace event, agents have to request the corresponding tracing service by invoking `requestTracingService`. This subscription can be cancelled later by invoking

`cancelTracingServiceSubscription`.

It is also possible to subscribe to all available tracing services offered by any tracing entity in the system by invoking the method `requestAllTracingServices`. The Trace Manager only alows for subcribing to all available has to be launched in monitorization mode. Thus, the following code would not work and the requester agent would receive a `REFUSE` ACL message with an `AUTHORIZATION_ERROR` error code:

```
1 TraceManager tm = new TraceManager(new AgentID("tm"));
2
3 /* More code here... */
4
5 /* This will NOT work */
6 requestAllTracingServices(this);
```

Launching the Trace Manager with the monitorization flag set to true, like in the example, would do the job:

```
1 TraceManager tm = new TraceManager(new AgentID("tm"), true);
2
3 /* More code here... */
4
5 /* This will work */
6 requestAllTracingServices(this);
```

All these methods related to the subscription and unsubscription processes are described in more detail in Table 3.4.

An example of tracing service subscription can be found in Section 3.7.6, in line number 20, where the agent subscribes to the *NEW_AGENT* tracing service. Line number 68 also shows an example of subscription to a tracing service called *MESSAGE_SENT_DETAIL* offered by a specific origin entity. Later in that source code, in lines 74 and 75, it can be observed how the agent unsubscribes from these tracing services.

3.7.3.3 Listing

The Trace Manager allows for listing tracing entities and tracing services available in the system by invoking `listTracingEntities` and `listTracingServices` respectively. These

Table 3.4: Tracing service subscription and unsubscription methods

requestTracingService(BaseAgent requesterAgent, String serviceName, AgentID originEntity**)**
requestTracingService(AgentID tms_aid, BaseAgent requesterAgent, String serviceName, AgentID originEntity**)**
requestTracingService(BaseAgent requesterAgent, String serviceName**)**
requestTracingService(AgentID tms_aid, BaseAgent requesterAgent, String serviceName**)**
requestAllTracingServices(BaseAgent requesterAgent**)**
requestAllTracingServices(AgentID tms_aid, BaseAgent requesterAgent**)**
cancelTracingServiceSubscription(BaseAgent requesterAgent, String serviceName, AgentID originEntity**)**
cancelTracingServiceSubscription(AgentID tms_aid, BaseAgent requesterAgent, String serviceName, AgentID originEntity**)**
cancelTracingServiceSubscription(BaseAgent requesterAgent, String serviceName**)**
cancelTracingServiceSubscription(AgentID tms_aid, BaseAgent requesterAgent, String serviceName**)**

tms_aid:	AgentID of the Trace Manager which will process the request. If not specified, the request is directed to the default Trace Manager, tm.
requesterAgent:	Agent which is subscribing/unsubscribing to the tracing service.
serviceName:	String identifying the tracing service to which the request refers.
originEntity:	AgentID of the specific agent which offers the tracing service. If not specified, the subscription/unsubscription request is considered to refer to tracing services offered by any tracing entity in the system.

two methods are described in more detail in Table 3.5.

In response to any of these requests, a list of available tracing entities or available tracing services will be sent to the applicantAgent as an AGREE ACL message. The requested list will be included in the content field of the ACL message.

When a list of avalable tracing entities has been requested, the content of the reply message will be structured as follows:

- Message content:
  $list\#entities\# < number\ of\ t\ entities > \# < t\ entity\ description\ list >$
- <t entity description list>: List of concatenated tracing entity descriptions, each of which is structured as follows:

$\#<entity\ type>\#<entity\ identifier\ length>\#<entity\ identifier>$

- <entity type>: $\{0, 1, 2\}$ (meaning agent, artifact or aggregation)

When a list of tracing services has been requested, the `content` of the reply message will be structured as follows:

- Message content:
  $list\#services\#<number\ of\ t\ services>\#<t\ service\ description\ list>$
- <t service description list>: List of concatenated tracing service descriptions, each of which is structured as follows:
  $\#<service\ name\ length>\#<service\ name>\#<service\ description\ length>\#<service\ description>$

Table 3.5: Tracing services and tracing entities listing methods

listTracingEntities(`BaseAgent requesterAgent`**)**
listTracingEntities(`AgentID tms_aid, BaseAgent requesterAgent`**)**
listTracingServices(`BaseAgent requesterAgent`**)**
listTracingServices(`AgentID tms_aid, BaseAgent requesterAgent`**)**

tms_aid:	`AgentID` of the Trace Manager which will process the request. If not specified, the request is directed to the default Trace Manager, `tm`.
requesterAgent:	Agent which is requesting the list of available tracing entities and services.

3.7.4 Domain Independent Tracing Services

The platform offers a set of *domain independent* tracing services. Some of them can be requested by agents in order to receive the corresponding trace events, while others are not requestable and the corresponding trace events are received even without having requested them previously (this can be seen as a *default subscription* to the tracing service). The rest of the section will describe them as well as how to interpret their correspoding trace events, according to the `TraceEvent` class and its attributes, previously described in Section 3.7.2. Domain independent tracing services can be classified in four main groups: System, agent's lifecycle, messaging and tracing service publication. These tracing services are included in the class `es.upv.dsic.gti_ia.coreTracingService`

3.7.4.1 System domain independent tracing services

System domain independent tracing services provide information which may be necessary for any tracing entity in order to understand the sequence of trace events which it has received. These tracing services are not requestable and thus, trace events are received by tracing entities as they excute, without having previously subscribed to them. For instance, if a tracing service were not available anymore, all those agents which had previously requested it, would receive an `UNAVAILABLE_TS` trace event, so that these agents know that the service is not being offered anymore. Details on system domain independent tracing services and on the information which their events provide are detailed in Table 3.6.

3.7.4.2 Agent's lifecycle domain independent tracing services

These domain independent tracing services provide tracing information related to agents entering or leaving the system. These tracing services are requestable and thus, it is necessary to subscribe to them before receiving any trace event they may provide. An example of use of these tracing services can be observed in Section 3.7.6, in line number 20. The daddy agent requests the tracing service `NEW_AGENT` in order to receive a trace event each time a new agent enters the system. These events are later processed in the event handler, in line number 67.

More details on these tracing services and on how to understand the information provided by these trace events are available in Table 3.7.

3.7.4.3 Messaging related domain independent tracing services

These domain independent tracing services provide information related to message based agent communication. These tracing services are also requestable and thus, it is also necessary to subscribe to them before receiving any trace event they may provide. An example of use of these tracing services can be observed in Section 3.7.6, in line number 68, where the daddy agent requests a tracing service called `MESSAGE_SENT_DETAIL` in order to inspect every message that his boy agents, Bobby or Timmy, send.

More details on these tracing services and on how to understand the information provided by these trace events are available in Table 3.8.

Table 3.6: System related domain independent tracing services

TRACE_ERROR:	
Generic non determined error in the tracing process.	
tService:	TRACE_ERROR (0)
originEntity:	system@host
content:	Human-readable error description.

SUBSCRIBE:	
The tracing entity requested a tracing service with name tServiceName to an ES entity (ESid), which can be any to express interest in trace events of that tracing service coming from any of ES entities. From that time, trace events of that tracing service may be delivered to the tracing entity.	
tService:	SUBSCRIBE (1)
originEntity:	Trace Manager entity to which the tracing service request was made.
content:	tServiceName#tServiceDescription.length()# tServicedescription#ESid (ESid can be any).

UNSUBSCRIBE:	
The tracing entity cancelled the subscription to a tracing service (tServiceName) coming from the ES entity ESid, which can be any if the removed subscription referred to trace events coming from any ES entity which provided it.	
tService:	UNSUBSCRIBE (2)
originEntity:	Trace Manager entity to which the tracing service request was made.
content:	tServiceName#ESid (ESid can be any).

UNAVAILABLE_TS:	
The specified tracing service ServiceName is no longer available. This can be the consecuence of the origin entity which provided the tracing service terminating its own execution or unpublishing the tracing service. It also can be a consecuence of changes in the authorization graph of the tracing service. Receiving this trace event implies unsubscription from the tracing service, but no UNSUBSCRIBE trace event is expected.	
tService:	UNAVAILABLE_TS (3)
originEntity:	system@host
content:	tServiceName#ESid (ESid can be any).

3.7.4.4 Tracing service publication related domain independent tracing services

These domain independent tracing services provide information related to tracing services being published by tracing entities in the system. For instance, a generic agent may need to be noticed when a specific tracing service is being offered or when another agent is not sharing certain tracing information anymore. These domain independent tracing services and the

Table 3.7: Agent's lifecycle related domain independent tracing services

NEW_AGENT:	
A new agent (`AgentId`), executing in a host (`host`) was registered in the system.	
tService:	`NEW_AGENT` (4)
originEntity:	`system@host`
content:	`AgentId`
AGENT_DESTROYED:	
An agent (`AgentId`), executing in a host (`host`) was destroyed.	
tService:	`AGENT_DESTROYED` (5)
originEntity:	`system@host`
content:	`AgentId`

Table 3.8: Agent's messaging related domain independent tracing services

MESSAGE_SENT:	
A FIPA-ACL message was sent from `OriginAgentId` to `DestinationAgentId`.	
tService:	`MESSAGE_SENT` (6)
originEntity:	`OriginAgentId`
content:	`DestinationAgentId`
MESSAGE_SENT_DETAIL:	
A FIPA-ACL message was sent from `OriginAgentId`. The difference with `MESSAGE_SENT` is that `MESSAGE_SENT_DETAIL` trace events include the ACL message they refer to.	
tService:	`MESSAGE_SENT_DETAIL` (7)
originEntity:	`OriginAgentId`
content:	`SerializedMessage`
MESSAGE_RECEIVED:	
A FIPA-ACL message was received by `DestinationAgentId`.	
tService:	`MESSAGE_RECEIVED` (8)
originEntity:	`DestinationAgentId`
content:	`OriginAgentId`
MESSAGE_RECEIVED_DETAIL:	
A FIPA-ACL message was received by `DestinationAgentId`. The difference with `MESSAGE_RECEIVED` is that `MESSAGE_RECEIVED_DETAIL` trace events include the ACL message they refer to.	
tService:	`MESSAGE_RECEIVED_DETAIL` (9)
originEntity:	`DestinationAgentId`
content:	`SerializedMessage`

information they provide are detailed in Table 3.9

Table 3.9: Tracing service publication related domain independent tracing services

PUBLISHED_TRACING_SERVICE:	
A tracing service `ServiceName` was published by the tracing entity `ESId`	
tService:	`PUBLISHED_TRACING_SERVICE` (10)
originEntity:	`ESId`
content:	`ServiceName`
UNPUBLISHED_TRACING_SERVICE:	
A tracing service `ServiceName` was unpublished by the tracing entity `ESId`.	
tService:	`UNPUBLISHED_TRACING_SERVICE` (11)
originEntity:	`ESId`
content:	`ServiceName`

3.7.5 Example: TraceDaddy

Simple example of how to use domain independent tracing services to follow other agents' activities and to make decisions according to this activity.

In this case, a Daddy agent listens to his sons (Boy agents) while they are playing and when one of them starts crying, he proposes them to take them to the park. When both children agree, daddy and his sons leave the building and the application finishes.

- **Initialization:**
  - DADDY:
    Requests to the NEW_AGENT tracing service in order to know when children arrive.
    Prints on screen that he intends to read the newspaper.
  - BOYS (Bobby and Timmy):
    Print on screen their name and age.
- **Execution:**
  - DADDY:
    Each time a NEW_AGENT event is received, Daddy requests the tracing service MESSAGE_SENT_DETAIL in order to 'listen' to what that agent says.
    Each time a MESSAGE_SENT_DETAIL trace event is received, Daddy prints its content on screen and checks if the content of the message is equal to 'GUAAAAAA!'.
    If so, Daddy cancels the subscription to MESSAGE_SENT_DETAIL tracing services and sends ACL request messages to both children to propose the go to the

park.

When both childre have replied with an AGREE message, Daddy agent prints it on screen and ends its execution.

- BOYS (Bobby and Timmy):
  Bobby, which is only 5, sends each second an ACL request message to Timmy (which is 7) to request him his toy (Give me your toy). After 5 denials, Bobby starts requesting it by crying (sending an ACL message with a loud GUAAAAAA!).
  Both Boy agents reply NO! to any request which does not come from their father and only AGREE when their dad requests them to GO TO THE PARK.
  When dad requests them (via an ACL message) to go to the park, both sons agree and end their execution.

3.7.6 `Daddy` class

```
1  package TraceDaddy;
2
3  import java.text.DateFormat;
4  import java.text.SimpleDateFormat;
5  import java.util.Calendar;
6
7  import es.upv.dsic.gti_ia.core.ACLMessage;
8  import es.upv.dsic.gti_ia.core.AgentID;
9  import es.upv.dsic.gti_ia.core.BaseAgent;
10 import es.upv.dsic.gti_ia.core.TraceEvent;
11 import es.upv.dsic.gti_ia.trace.TraceInteract;
12
13 public class Daddy extends BaseAgent{
14    private boolean finish=false;
15    private boolean Bobby_agree=false;
16    private boolean Timmy_agree=false;
17
18    public Daddy(AgentID aid) throws Exception{
19       super(aid);
20       TraceInteract.requestTracingService(this, "NEW_AGENT");
21       System.out.println("[Daddy " + this.getName() + "]: I want to
             read the newspaper...");
22    }
23
```

```
24     public void execute(){
25        ACLMessage msg;
26        while(!finish){
27           try {
28              Thread.sleep(1000);
29           } catch (InterruptedException e) {
30              e.printStackTrace();
31           }
32
33           System.out.println("[Daddy " + this.getName() + "]: Ok! I
                 give up... Shall we go to the park?");
34
35           msg = new ACLMessage(ACLMessage.REQUEST);
36           msg.setSender(this.getAid());
37           msg.setContent("GO TO THE PARK");
38           msg.setReceiver(new AgentID("Timmy"));
39           send(msg);
40           msg.setReceiver(new AgentID("Bobby"));
41           send(msg);
42           while(!Bobby_agree || !Timmy_agree){
43              try {
44                 Thread.sleep(1000);
45              } catch (InterruptedException e) {
46                 e.printStackTrace();
47              }
48           }
49
50           try {
51              Thread.sleep(1000);
52           } catch (InterruptedException e) {
53              e.printStackTrace();
54           }
55
56           System.out.println("[Daddy " + this.getName() + "]: Ok! Let
                 's go, children!");
57        }
58
59        public void onTraceEvent(TraceEvent tEvent) {
60           DateFormat formatter = new SimpleDateFormat("HH:mm:ss.SSS")
                 ;
```

```
61
62          Calendar calendar = Calendar.getInstance();
63          calendar.setTimeInMillis(tEvent.getTimestamp());
64
65          ACLMessage msg;
66
67          if (tEvent.getTracingService().contentEquals("NEW_AGENT")){
68             TraceInteract.requestTracingService(this, "
                   MESSAGE_SENT_DETAIL", new AgentID(tEvent.getContent()
                   ));
69          }
70          else if (tEvent.getTracingService().contentEquals("
                MESSAGE_SENT_DETAIL")){
71             msg = ACLMessage.fromString(tEvent.getContent());
72             System.out.println("[" + this.getName() + " " +
                   formatter.format(calendar.getTime()) + "]: " + msg.
                   getSender().toString() + " said: " + msg.
                   getPerformative() + ": " + msg.getContent());
73             if (msg.getContent().contentEquals("GUAAAAAA..!")){
74                TraceInteract.cancelTracingServiceSubscription(this,
                      "MESSAGE_SENT_DETAIL",new AgentID("Timmy"));
75                TraceInteract.cancelTracingServiceSubscription(this,
                      "MESSAGE_SENT_DETAIL",new AgentID("Bobby"));
76                finish=true;
77             }
78          }
79       }
80
81       public void onMessage(ACLMessage msg){
82          if((msg.getPerformativeInt() == ACLMessage.AGREE) && (msg.
                getContent().contentEquals("GO TO THE PARK"))){
83             System.out.println("[Daddy " + this.getName() + "]: " +
                   msg.getSender().name + " says: " + msg.
                   getPerformative() + " " + msg.getContent());
84             if (msg.getSender().getLocalName().contentEquals("Bobby
                   ")){
85                Bobby_agree=true;
86             }
87
```

```
88              if (msg.getSender().getLocalName().contentEquals("Timmy
                    ")){
89                  Timmy_agree=true;
90              }
91          }
92      }
93  }
```

3.7.7 Boy class

```
1  package TraceDaddy;
2
3  import es.upv.dsic.gti_ia.core.ACLMessage;
4  import es.upv.dsic.gti_ia.core.AgentID;
5  import es.upv.dsic.gti_ia.core.BaseAgent;
6
7  public class Boy extends BaseAgent {
8     private int age;
9     private boolean finish=false;
10    AgentID dad;
11
12    public Boy (AgentID aid, int age, AgentID dad) throws Exception{
13       super(aid);
14       this.age=age;
15       this.dad=dad;
16       System.out.println("[" + this.getName() + "]: I'm " + this.
             getName() + " and I'm "+ this.age + " years old!");
17    }
18
19    public void execute(){
20       ACLMessage msg;
21       int counter=5;
22       while(!finish){
23          if (this.age <= 5) {
24             msg = new ACLMessage(ACLMessage.REQUEST);
25             msg.setSender(this.getAid());
26             if (counter > 0){
27                msg.setContent("Give me your toy...");
```

```
28              }
29              else{
30                 msg.setContent("GUAAAAAA..!");
31              }
32              counter--;
33
34              msg.setReceiver(new AgentID("qpid://Timmy@localhost
                    :8080"));
35              send(msg);
36           }
37           try {
38              Thread.sleep(1000);
39           } catch (InterruptedException e) {
40              e.printStackTrace();
41           }
42        }
43     }
44
45     public void onMessage(ACLMessage msg){
46        if (msg.getSender().getLocalName().contentEquals(dad.
              getLocalName())){
47           // Daddy!
48           if(msg.getPerformativeInt() == ACLMessage.REQUEST){
49              if (msg.getContent().contentEquals("GO TO THE PARK")){
50                 finish=true;
51                 ACLMessage response_msg = new ACLMessage(ACLMessage.
                       AGREE);
52                 response_msg.setSender(this.getAid());
53                 response_msg.setContent("GO TO THE PARK");
54                 response_msg.setReceiver(msg.getSender());
55                 send(response_msg);
56              }
57           }
58        }
59        else{
60           // You no daddy!
61           if(msg.getPerformativeInt() == ACLMessage.REQUEST){
62              ACLMessage response_msg = new ACLMessage(ACLMessage.
                    REFUSE);
63              response_msg.setSender(this.getAid());
```

```
64              response_msg.setContent("NO!");
65              response_msg.setReceiver(msg.getSender());
66              send(response_msg);
67           }
68        }
69     }
70  }
```

3.7.8 Main application source code

```
1  package TraceDaddy;
2
3  import org.apache.log4j.Logger;
4  import org.apache.log4j.xml.DOMConfigurator;
5
6  import es.upv.dsic.gti_ia.core.AgentID;
7  import es.upv.dsic.gti_ia.core.AgentsConnection;
8  import es.upv.dsic.gti_ia.trace.TraceManager;
9
10 public class Run {
11    public static void main(String[] args) {
12       Boy olderSon, youngerSon;
13       Daddy dad;
14       /**
15        * Setting the Logger
16        */
17       DOMConfigurator.configure("configuration/loggin.xml");
18       Logger logger = Logger.getLogger(Run.class);
19
20       /**
21        * Connecting to Qpid Broker
22        */
23       AgentsConnection.connect();
24
25       try {
26          /**
27           * Instantiating the Trace Manager
28           */
```

```
29            TraceManager tm = new TraceManager(new AgentID("tm"));
30
31            System.out.println("INITIALIZING...");
32
33            /**
34             * Instantiating Dad
35             */
36            dad = new Daddy(new AgentID("qpid://MrSmith@localhost
                  :8080"));
37
38            /**
39             * Instantiating sons
40             */
41            olderSon = new Boy(new AgentID("qpid://Timmy@localhost
                  :8080"), 7, dad.getAid());
42            youngerSon = new Boy(new AgentID("qpid://Bobby@localhost
                  :8080"), 5, dad.getAid());
43
44            /**
45             * Execute the agents
46             */
47            dad.start();
48            olderSon.start();
49            youngerSon.start();
50         } catch (Exception e) {
51            logger.error("Error  " + e.getMessage());
52         }
53      }
54   }
```

3.7.9 Results

```
INITIALIZING...
[Daddy MrSmith]: I want to read the newspaper...
[Timmy]: I'm Timmy and I'm 7 years old!
[Bobby]: I'm Bobby and I'm 5 years old!
[MrSmith 16:11:06.077]: qpid://Timmy@localhost:8080
        said: REFUSE: NO!
[MrSmith 16:11:07.072]: qpid://Bobby@localhost:8080
        said: REQUEST: Give me your toy...
[MrSmith 16:11:07.077]: qpid://Timmy@localhost:8080
        said: REFUSE: NO!
[MrSmith 16:11:08.075]: qpid://Bobby@localhost:8080
        said: REQUEST: Give me your toy...
[MrSmith 16:11:08.080]: qpid://Timmy@localhost:8080
        said: REFUSE: NO!
[MrSmith 16:11:09.077]: qpid://Bobby@localhost:8080
        said: REQUEST: Give me your toy...
[MrSmith 16:11:09.083]: qpid://Timmy@localhost:8080
        said: REFUSE: NO!
[MrSmith 16:11:10.080]: qpid://Bobby@localhost:8080
        said: REQUEST: Give me your toy...
[MrSmith 16:11:10.094]: qpid://Timmy@localhost:8080
        said: REFUSE: NO!
[MrSmith 16:11:11.082]: qpid://Bobby@localhost:8080
        said: REQUEST: GUAAAAAA..!
[MrSmith 16:11:11.087]: qpid://Timmy@localhost:8080
        said: REFUSE: NO!
[Daddy MrSmith]: Ok! I give up... Shall we go to the park?
[Daddy MrSmith]: Timmy says: AGREE GO TO THE PARK
[Daddy MrSmith]: Bobby says: AGREE GO TO THE PARK
[Daddy MrSmith]: Ok! Let's go, children!
```

CHAPTER

4

Virtual Organizations

As it has been pointed out in the introduction, Magentix2 platform not only has as aims to provide a guaranteed communication mechanism to the programmer, but also to provide a complete support for virtual organizations. THOMAS (Methods, Techniques and Tools for Open Multi-Agent Systems) framework has been integrated with Magentx2 with this purpose.

4.1 Overview of THOMAS architecture

The THOMAS framework tries to communicate agents and web services in a transparent, but independent way. Agents can offer and invoke services in a transparent way to other agents or entities, as well as external entities can interact with our agents through the use of the offered services. THOMAS architecture consists of a set of modular services. Agents have access to the infrastructure offered by THOMAS through a set of services including on two main components:

Service Facilitator (SF) This component offers simple and complex services to the active agents and organizations. Basically, its functionality is like a yellow page service and a service descriptor in charge of providing a green page service.

Organization Manager Service (OMS) It is mainly responsible of the management of the organizations and their entities. Thus, it allows the creation and the management of any organization, the roles the agents play and the norms that rule their behavior.

4.1.1 Service Facilitator

The SF is a mechanism and support used by organizations and agents to offer and discover services. The SF provides an environment in which the autonomous entities can register service descriptions as directory entries.

The SF acts as a gateway to access the THOMAS platform. The SF can find services searching for a given service profile . This is done using the matchmaking mechanisms that are provided by the SF.

The SF also acts as a yellow pages manager and in this way it can also find which entities provide a given service. Services may have some pre-conditions that have to be true before the service can be executed. They exchange one or more input and output messages. A successful service execution has some effects on its environment. This parameters are called IOPE (input/output/preconditions/effects). Moreover, there could be additional parameters, which are independent of the service functionality (non-functional parameters), such as quality of service, deadlines, and security protocols among others.

A service represents an interaction between two entities, which are modelled as communications among independent processes. In our case, the Multi-agent Technology provides us with FIPA communication protocols.

Taking into account that we are dealing with semantic services, another important data is the ontology used in the service. When the service description is accessed, any entity has all the needed information available to interact with the service.

The set of services provided by the SF to manage the services of the platform (meta-services) are classified in 3 categories:

- **Registration:** they allow adding, modifying and removing services from the SF directory (RegisterProfile, RegisterProcess, ModifyProfile, ModifyProcess).
- **Affordability:** for managing the association between providers and their services (AddProvider, RemoveProvider).

- **Discovery:** for searching and composing services as an answer to user requirements (SearchService, GetProfile, GetProcess).

4.1.2 Organization Manager Service

This component is in charge of organizations life cycle management, including specification and administration of their structural components (roles, organizative units and norms) and their execution components (participant agents and the roles they play; and active units in each moment). OMS offers all services needed for a suitable organization performance. These services are classified as:

- Structural services, that modify the structural and normative organization specification;
- Dynamical services, that allow agents to entry or leave the organization dynamically, as well as role adoption.

By means of the publication of the structural services, OMS allows modifying, in execution time, some aspects related to the organization structure, functionality or normativity. Some of these services should be restricted to internal roles of the system, which have enough permission to do this kind of operations (i.e. a supervisor role). Dynamic services manage the creation/registration of new agents in the organization, entry or exit of unit members and role adoption. These services are always published in the SF.

The OMS provides agents with a set of services for organization life-cycle management, classified in:

- **Structural services:** which modify the structural and normative organization specification, i.e. roles, norms and organizational units (RegisterRole, DeregisterRole, RegisterNorm, De-registerNorm,RegisterUnit,DeregisterUnit).
- **Informative services:** that give information of the current state of the organization (InformUnitRoles, InformAgentRoles, InformMembers, InformQuantity, InformUnit, InformRoleProfiles, InformRoleNorms).
- **Dynamic services:** that allow managing role enactment and dynamic entry/exit of agents (AcquireRole, LeaveRole,Expulse) [Val et al., 2009].

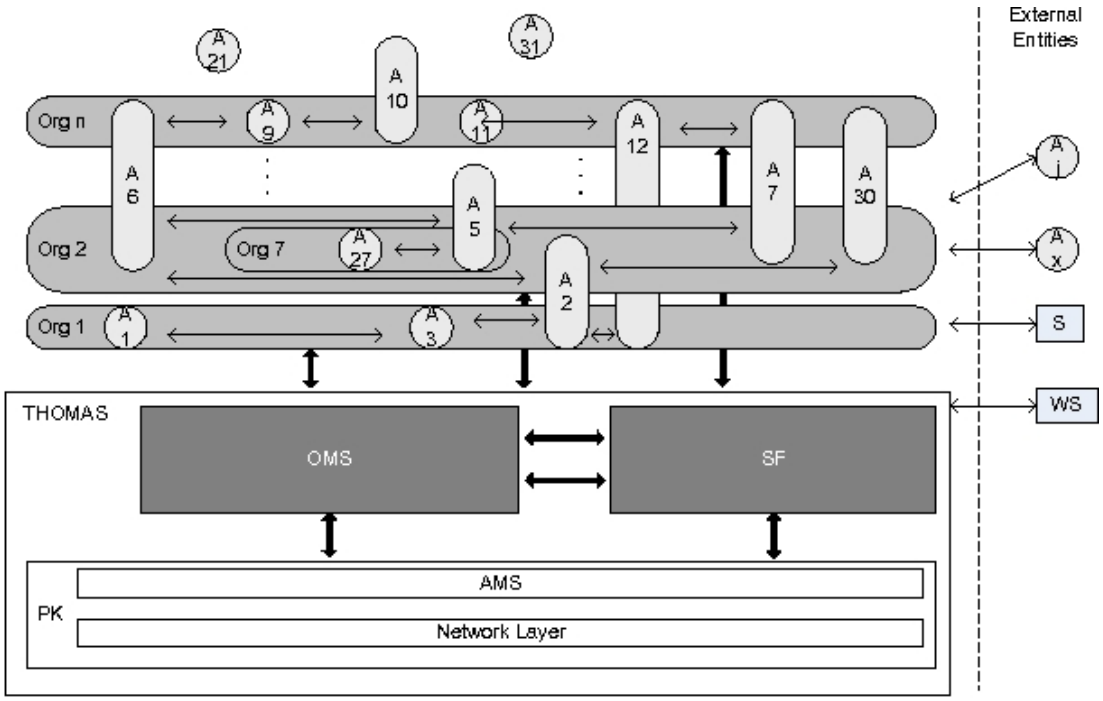

Figure 4.1: THOMAS architecture

4.2 Programming agents which use THOMAS

4.2.1 Magentix2 API for THOMAS

SF and OMS have been defined as two types of intermediary agents in order to address the translation between Magentix2 agents (or any external agent), that implement FIPA communication,and the services they provide. Services requests through FIPA-request protocol were received by this type of agents.

To ease the interaction among user agents and OMS and SF agents, two classes are provided (`OMSProxy` and `SFProxy`). They work as a proxy for OMS and SF respectively, encapsulating and hiding the details of the underlying communication protocol. By doing this, the developer can interact with OMS and SF using simple function calls.

4.2.1.1 OMSProxy

The OMSProxy can be found in the package `es.upv.dsic.gti_ia.organization`. To use its functionality, a new instance of the `OMSProxy` class must be created to access the methods

contained in the OMS. In the constructor, the agent who executes the service and the url where OMS services are deployed (.war) can be specified:

```
1          private OMSProxy omsProxy = new OMSProxy(this, "url");
```

The route can be empty if it is taken from the settings.xml configuration file, please see section 4.5.1 (recommended option).

```
1          private OMSProxy omsProxy = new OMSProxy(this);
```

OMSProxy provides a developer with a set of methods to manage available services, this services are divided into different types and sub-types. Tables 4.1, 4.2 and 4.3 show respectively these sub-types.

All these methods return a status indicating whether the operation was successful or not.

```
1
2          omsProxy.acquireRole("member", "virtual");
```

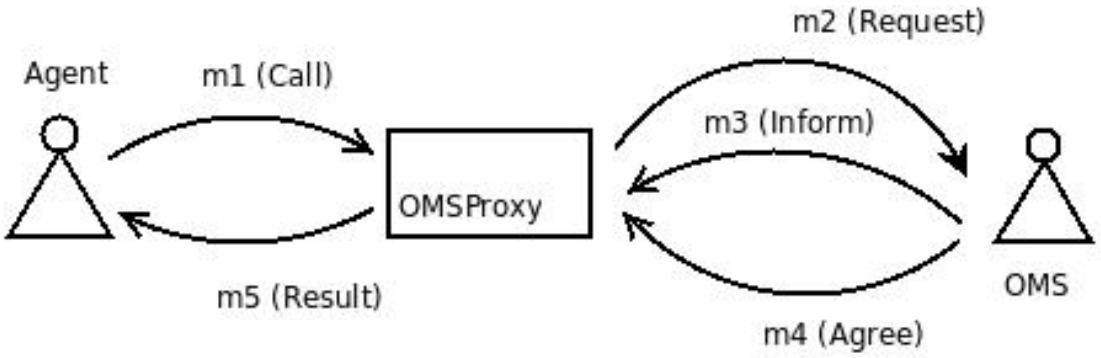

Figure 4.2: Interaction between user agent and OMS agent through the OMSProxy

4.2.1.2 SFProxy

The SFProxy we can find inside the package `es.upv.dsic.gti_ia.organization`. To use its functionality, a new instance of the `SFProxy` class must be created to access the methods contained in the SF. In the constructor, the agent who executes the service and the url where SF services are deployed (.war) can be specified:

OMS-Service	Description	Parameters
RegisterRole	Creates a new role within a unit	**RoleID:** Identifier of the new role. **UnitID:** Identifier of the organizational unit. **Accessibility:** Considers two types of roles: *internal* or *external*. Default is *external*. **Position:** Position inside the unit, such as *member*, *supervisor* or *subordinate*. Default is *member*. **Visibility:** Is defined (*public*) or from inside (*private*). Default is *public*. **Inheritance:** Identifier of the parent role in the role hierarchy. Default is *member*.
RegisterNorm	Includes a new norm within a unit	**NormID:** Identifier of the new norm. **NormContent:** Content of the rules.
RegisterUnit	Creates a new unit within a specific organization	**UnitID:** Identifier of the new unit. **Type:** Indicates the topology of the new unit: (i) *Hierarchy* (ii) *Team*,(iii) *Flat*. Default is *flat*.
DeregisterRole	Removes a specific role description from a unit scription from a unit	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
DeregisterNorm	Removes a specific norm description	**NormID:** Identifier of the norm.
DeregisterUnit	Removes a unit from an organization	**UnitID:** Identifier of the unit.

Table 4.1: OMS Proxy: Registration API

OMS-Service	Description	Parameters
InformAgentRole	Indicates roles adopted by an agent	**AgentID:** Is formed of *protocol://name@host:port*.
InformMembers	Indicates entities that are members of a specific unit	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
QuantityMembers	Provides the number of current members of a specific unit	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
InformUnit	Provides unit description	**UnitID:** Identifier of the unit.
InformUnitRoles	Indicates which roles are the ones defined within a specific Unit	**UnitID:** Identifier of the unit.
InformRoleProfiles	Indicates all profiles associated to a specific role	**UnitID:** Identifier of the unit.
InformRoleNorms	Provides all norms addressed to a specific role	**RoleID:** Identifier of the role.

Table 4.2: OMS Proxy: Information API

OMS-Service	Description	Parameters
AcquireRole	Requests the adoption of a specific role within a unit	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
LeaveRole	Requests to leave a role	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
Expulse	Forces an agent to leave a specific role	**ExpulseAgentID:** Is formed of *protocol://name@host:port*. **RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.

Table 4.3: OMS Proxy: Compound API

```
private SFProxy sfProxy = new SFProxy(this, "url");
```

The route can be empty if it is taken from the settings.xml configuration file, please see section 4.5.1 (recommended option).

```
private SFProxy sfProxy = new SFProxy(this);
```

SFProxy provides a developer with a set of methods to manage available services, as Table 4.4 shows:

All these methods return a status indicating whether the operation was successful or not.

```

        results=sfProxy.searchService("SearchCheapHotel");
```

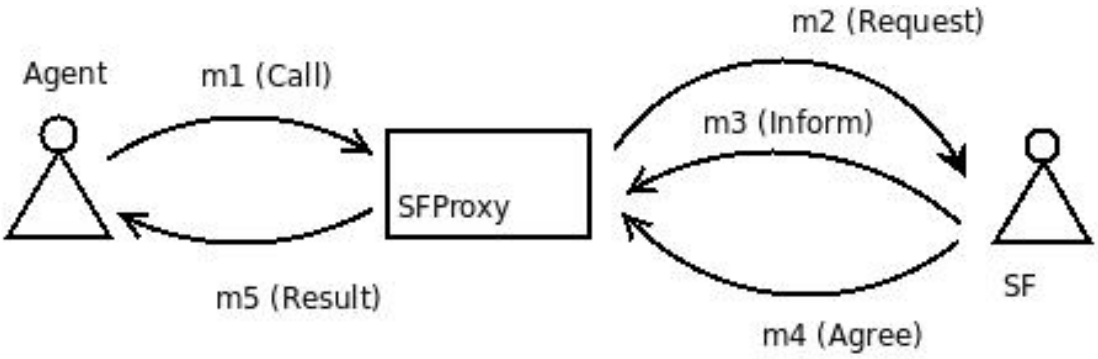

Figure 4.3: Interaction between user agent and SF agent through the SFProxy

4.2.1.3 Basic Service Management

Services are composed by a profile, which is a semantic description of the service, useful for customers to locate appropriate service, and a process, which details how to interact with the service.

Different agents can provide different processes (different implementations) to the same service profile (general description), so profiles and services are kept separately. Some templates which can adequately fill in all required fields in a profile (ProfileDescription) and process (ProcessDescription) have been created.

The **ProfileDescription** can be found inside the package `es.upv.dsic.gti_ia`

Type	SF-Service	Description	Parameters
Registration	RegisterProfile	Creates a new service description (profile)	**ProfileDescription:** Template with profile methods and properties.
	RegisterProcess	Creates a particular implementation (process) for a service	**ProcessDescription:** Template with process methods and properties.
	ModifyProfile	Modifies an existing service profile	**ProfileDescription:** Template with profile methods and properties.
	ModifyProcess	Modifies an existing service process	**ProcessDescription:** Template with process methods and properties.
	DeregisterProfile	Removes a service description	**ProfileDescription:** Template with profile methods and properties.
Affordb.	RemoveProvider	Removes a provider from a service	**ProcessDescription:** Template with process methods and properties.
Discovery	SearchService	Searches a service (or a composition of services) that satisfies the user requirements	**ServiceGoal:** String with name of service purpose.
	GetProfile	Gets the description (profile) of a specific a Service	**ServiceID:** String with the service profile ID.
	GetProcess	Gets the implementation (process) of a specific a service	**ServiceGoal**: String with name of service purpose.

Table 4.4: SF Proxy API

`.organization`. This template contains all methods and variables needed for the proper configuration of a profile. The most important slots are:

- `ServiceID`: It is generated automatically by the database. It is a numeric value.
- `URLProfile`: The url where the profile document (OWL-S) is located.
- `Serviceprofile`: It is a url which makes reference to the service profile description document.

- `Profile name`: Name of the service profile description document.

When an agent needs to register a new profile in the organization, the `RegisterProfile` method of the `SFProxy` class is invoked. The following code shows how to create a new profile and register it in the organization.

```
import es.upv.dsic.gti_ia.organization.SFProfileDescription;

ProfileDescription profile = new ProfileDescription(
                         "URLProfile","SearchCheapHotel");

sfProxy.registerProfile(profile);
```

The **ProcessDescription** can be found inside the package `es.upv.dsic.gti_ia .organization`. In contains all the methods and variables needed for the proper configuration of a process. The most important slots are:

- `ServiceID`: It is a number generated automatically by the SF database.
- `Implementation ID`: It is the id of the service implementation. It is composed of: serviceid@serviceprocessid-agentid.
- `URLProcess`: The URL where the process owl document is located.
- `Servicemodel`: It is a url which makes reference to the service process description document.
- `Process name`: Name of the service process description document.

An agent can register its own implementation of an existing profile or create a new service from the scratch. In the latter case, the profile and the process must be specified. The following code shows how register a new implementation (process) for an existing profile:

```
import es.upv.dsic.gti_ia.organization.SFProcessDescription;

ProcessDescription process = new ProcessDescription(
                     "URLProcess","SearchCheapHotel");
```

```
5
6  results = sfProxy.searchService("SearchCheapHotel");
7  process.setProfileID(results.get(0));
8  sfProxy.registerProcess(process);
```

4.2.1.4 Oracle: extracting information from OWL-S

The oracle is a class that allows developers to parse the profile of a semantic service, specified in OWL-S, in order to extract the required information(such as service inputs, outputs, list of roles or organizational units).

Oracle needs just the url of the OWL-S file containing the service profile to analyze it (as a parameter in the constructor of the class, see the example). This url can be obtained with the method `getProfile` of the `SFPRoxy` class.

Once the profile is obtained, the oracle can be asked about any field in the profile information. The available methods for the Oracle class can be found in the Javadoc documentation of the project.

```
1  import java.net.URL;
2  import es.upv.dsic.gti_ia.organization.Oracle;
3  private Oracle oracle;
4  //get the profile information
5  String URLProfile =  sfProxy.getProfile(results.get(0));
6  URL profile;
7  //load in the oracle the profile and parse it
8  try {
9          profile = new URL(URLProfile);
10         oracle = new Oracle(profile);
11 } catch (MalformedURLException e) {
12         logger.error("ERROR: Profile URL Malformed!");
13         e.printStackTrace();
14 }
15 //access to the profile information through the oracle
16 oracle.getProviderList();
```

4.3 Programming Agents that Offer Services

This section describes how an agent that offers services to the other agents inside the organization can register, announce and provide its services.

4.3.1 Register and Acquire Role

- **Registration of an agent on the platform.** The agents request to be registered as a member of a THOMAS platform using the `AcquireRole` service:

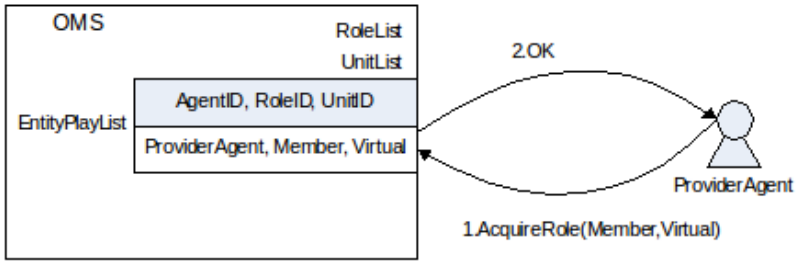

Figure 4.4: Agent interaction protocol to acquire role.

```
1
2 omsProxy.acquireRole("member", "virtual");
```

- **Registering a new provider.** Once the agent is registered, it can use public services offered by the OMS and SF. The provider agent requests to the SF the list of services that have been registered in the platform using a profile similar to the search cheap hotels service:

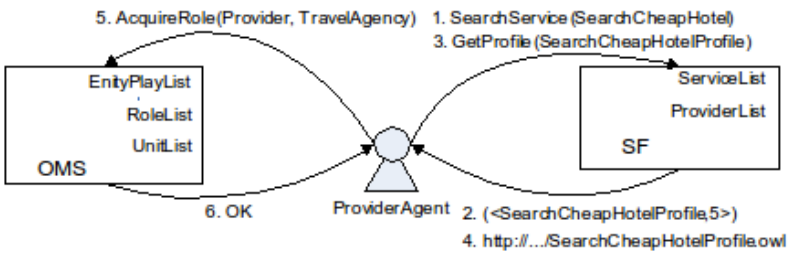

Figure 4.5: Agent interaction protocol to register process.

```
results=sfProxy.searchService("SearchCheapHotel");
```

If there is some service that meets the description, the provider agent uses the `getProfile` service to retrieve the service profile information:

```
String URLProfile = sfProxy.getProfile(results.get(0));
```

The SF responds with the URI where the service profile description is located. The `ProviderAgent` analyzes the profile of the service. It indicates that any provider of the service must adopt the `Provider` role inside the unit `TravelAgency`:

```
URL profile;
try {
        profile = new URL(URLProfile);
        oracle = new Oracle(profile);

} catch (MalformedURLException e) {
        logger.error("ERROR: Profile URL Malformed!");
        e.printStackTrace();
}
omsProxy.acquireRole(oracle.getProviderList().get(0),
                        oracle.getProviderUnitList().get(0));
```

4.3.1.1 Service Registration

- **Registering a new implementation of a service**. In this case, the provider agent want to register its own service implementation SearchCheapHotel. To do this, a new process for the SearchCheapHotel profile must be created and registered in the organization (using `RegisterProcess`). The `proccessDescription` must have the `ID` of profile which associate the process (returned by the method `searchService`).

```

processDescription.setProfileID(results.get(0));

sfProxy.registerProcess(processDescription);
```

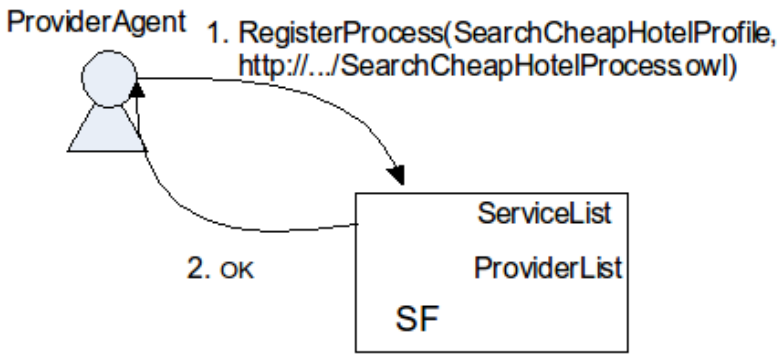

Figure 4.6: Agent interaction protocol to register process.

4.3.1.2 Execution of a Service (mindswap)

For the execution of the service,the Mindswap API[1] is used in Megentix2. This API offers an execution engine for OWL-S specifications. The `execute` method allows to execute directly a semantic service description in OWL-S. This method has two input parameters, a process description (`Process`) and a value map (`ValueMap`) with the values of all input parameters of any elemental service included in the service in execution.

The methods `getProcess` and `getServicerequestValues` allow the developer to get the process and the set of input values for any elemental service involved in the execution.

```
1  import org.mindswap.owls.OWLSFactory;
2  import org.mindswap.owls.process.Process;
3  import org.mindswap.owls.process.execution.ProcessExecutionEngine;
4  import org.mindswap.query.ValueMap;
5
6
7  // create an execution engine
8  ProcessExecutionEngine exec = OWLSFactory.createExecutionEngine();
9
10 try {
11         Process aProcess = processDescription.getProcess(inmsg);
12
13         // gwts the input values for the services
14         ValueMap values = new ValueMap();
```

[1] http://www.mindswap.org/

```
15          values = processDescription.getServiceRequestValues(inmsg);
16          //executes the service
17          values = exec.execute(aProcess, values);
18  } catch (Exception e) {
19          e.printStackTrace();
20  }
```

4.4 Programming Agents that Request Services

This section describes how an agent that requires services from other agents belonging the organization can search and request services.

4.4.1 Register and Acquire Role

- **Registration of an agent on the platform THOMAS.** The agents request to register as members of the THOMAS platform using the `AcquireRole` service:

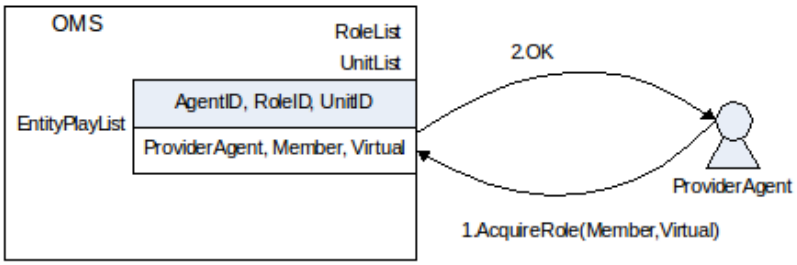

Figure 4.7: Agent interaction protocol to acquire role.

```
1
2   omsProxy.acquireRole("member", "virtual");
```

4.4.1.1 Service Search Process

- **Search of a service.** Once the agent is registered, it can use the `SearchService` service to find required services. The agent obtains a more detailed information of the service in which it is interested from the profile service, obtained with the `getProfile` service.

In that description, the agents realizes that it is necessary to play the `Customer` role, so the agents request it. Finally, when all this goals are accomplished, the agent can get the detailed information about the service and how to use it invoking the `GetProcess` service.

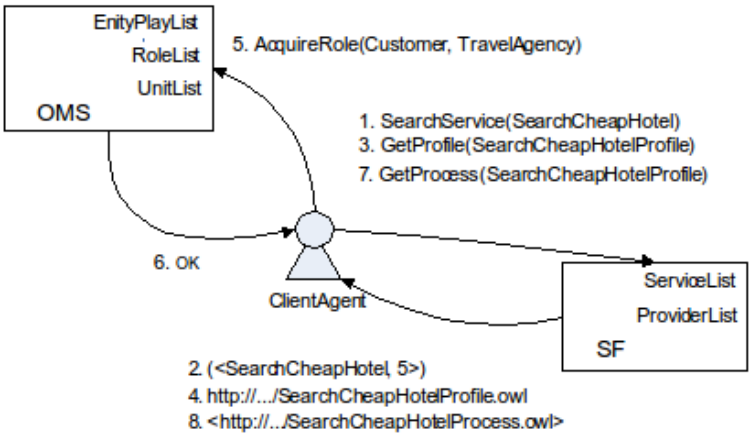

Figure 4.8: Agent interaction protocol to search service.

```
1
2  result = omsProxy.acquireRole("member", "virtual");
3
4  do {
5
6    results = sfProxy.searchService("SearchCheapHotel");
7
8  } while (results.size() == 0);
9
10   URLProfile = sfProxy.getProfile(results.get(0));
11   URL profile;
12
13   try {
14         profile = new URL(URLProfile);
15         oracle = new Oracle(profile);
16   } catch (MalformedURLException e) {
17         logger.error("ERROR: Profile URL Malformed!");
18         e.printStackTrace();
19   }
```

```
20
21 omsProxy.acquireRole(oracle.getClientList().get(0),
22             oracle.getClientUnitList().get(0));
23 agents = sfProxy.getProcess(results.get(0));
```

4.4.1.2 Service Request Process

- **Request a service.** After the agent has analysed the implementation details of the service (the process), it can contact with the provider agent and ask for the service (SearchCheapHotel) to the locate a hotel in the desired city. The `genericService` is used for this purpose.

```
1
2  ArrayList<String> arg = new ArrayList<String>();
3  int i = 0;
4
5  for (String input : oracle.getInputs()) {
6
7     switch (i) {
8         case 0:
9           System.out.println("Input: " + input);
10          arg.add("5");
11          break;
12        case 1:
13          System.out.println("Input: " + input);
14          arg.add("Spain");
15          break;
16        case 2:
17          System.out.println("Input: " + input);
18          arg.add("Valencia");
19          break;
20    }//switch
21    i++;
22   }//for
23
24 Enumeration<AgentID> agents1 = agents.keys();
25
26 AgentID agentToSend = agents1.nextElement();
27
```

```
28  URLProcess = agents.get(agentToSend);
29
30  // call the service SearchCheapHotel
31  list = sfProxy.genericService(agentToSend,URLProfile,
32                                              URLProcess, arg);
33  Enumeration<String> e = list.keys();
34
35  while (e.hasMoreElements()) {
36    String key = e.nextElement();
37    System.out.println(" " + key + " = " + list.get(key));
38  }//while
```

4.5 Running THOMAS Example

The examples folder of the Magentix2 packages contains a basic THOMAS example.

- **Thomas_Example**: This is an example of two agents (client and provider). These agents compose the travel agency organizational unit. The provider agent offers two services: searchs for touristic information and booking hotels. The client agent uses the SFProxy and the OMSProxy API in order to interact with the provider agent.

4.5.1 Initialization Tasks

Finally, how to specify the required parameters for THOMAS via the configuration file `Settings.xml` will be explained.

In section `Properties thomas`, the path where web services are deployed is required. Check if the direction where agents OMS and SF are running is different host from where the services (OMS and SF) are deployed, or if the port is different. If the OMS and SF are running in the same host that web services are deployed, the configuration by default is correctly.

```
<!-- Properties thomas -->
<entry key="OMSServiceDesciptionLocation">
    http://localhost:8080/omsservices/OMSservices/owl/owls/
</entry>
<entry key="SFServiceDesciptionLocation">
    http://localhost:8080/sfservices/SFservices/owl/owls/
</entry>
```

MySQL database information used by THOMAS:

```
<!-- Properties mysql -->
<entry key="serverName">localhost</entry>
<entry key="databaseName">thomas</entry>
<entry key="userName">thomas</entry>
<entry key="password">thomas</entry>
```

Jena database information used by THOMAS:

```
<!-- Properties jena -->
<entry key="dbURL">jdbc:mysql://localhost/thomas</entry>
<entry key="dbType">MySQL</entry>
<entry key="dbDriver">com.mysql.jdbc.Driver</entry>
```

Please, check that the data contained in the `Settings.xml` file fulfills with the actual settings for the existing databases.

CHAPTER

5

Platform administration

5.1 Custom Installation

The custom type installation allows selecting or un-selecting individually packages to be install in the system. If any of the following software have been installed in the system [1], it is necessary to unselect these packages for installation. Once the installation process finished, the instructions cited here per each particular software must be followed.

- **MySQL**

  In order to work correctly with THOMAS, it is necessary to add a new user called "thomas" and to create the THOMAS schema in the MySQL database. The following command must be executed:

  ```
  $ cd ~/Magentix2/bin/MySQL
  $ sh Import-ThomasDB.sh [MYSQL_ROOT_PASS]
  ```

  This command creates:

[1] The installation of any of these software components can cause conflicts in the system if already they exists.

  - User “thomas” with password “thomas”.
  - THOMAS schema. This contains all database tables and relationships needed for the THOMAS system.

- **Apache Tomcat** To configure Apache Tomcat to work properly with Magentix2, it is needed to move the content from the THOMAS directory to the current *webapps/* tomcat directory:

```
$ cd ~/Magentix2/thomas
$ mv * /tomcat_directory/webapps
```

- **Apache Qpid**

  The broker Qpid must be running in the system in order to connect agents:

```
$ cd $QPID_INSTALLATION
$ sbin/qpidd
```

On the other hand, if any of these components were installed during the Magentix2 Desktop version installation, they would be configured as follows:

- **MySQL:** it would be installed in the directory *mysql/*.

  By default, mysql root password is **mypassword**, this can be change in any moment.

- **Apache Tomcat:** it would be installed in the directory *apache-tomcat-7.0.4/* and **8080** is the port configured to receive services petitions.

  The Catalina.out is a log for tomcat, it can be found in log tomcat directory.

- **Apache Qpid:** it would be installed in the directory */opt/qpid/* and **5672** is the port configured to receive connections.

5.1.1 Magentix2 installation description

The following folders are created in the selected installation directory (by default *~/Magentix2*) both when full or custom installation mode is selected (depending on the selected packages):

- **bin/** includes the executable files and folders required to launch and start the platform and services. These are:

  - *Start-Magentix.sh*: it launches services and base agents of the platform [2]. Execute this script is not necessary the first time, because the installer program do it.
  - *Stop-Magentix.sh*: it stops services and agents of the platform.
  - *Launch-MagentixAgents.sh*: it launches the Magentix2 base agents (OMS, SF and bridge agents).
  - *Stop-MagentixAgents.sh*: it stops the Magentix2 base agents.

  This services can be managed separately, existing a subdirectory per each service.

  - **MySQL/** start, stop and configure MySQL service scripts.
  - **Tomcat/** start and stop Apache Tomcat service scripts.
  - **Qpid/** start and stop Apache Qpid service.

  Note: All the commands could been executed as follows:

```
$ cd  ~/Magentix2/bin or ~/Magentix2/bin/subdirectory
$ sh script_name.sh
```

  In addition, also the following sub-directories are included:

  - **configuration/** sub-directory: includes the Settings.xml and loggin.xml configuration files, necessary to launch Magentix2 user agents.
  - **security/** sub-directory: includes all required files to launch Magentix2 in secure mode.

- **docs/** includes javadoc and the Magentix2 documentation in Pdf format.
- **lib/** includes Magentix2 library an all additional libraries required by Magentix2. How to import this library in projects is explained in section 2.2.
- **examples/** includes some examples of Magentix2 agents implementation.
- **src/** includes Magentix2 sources.
- **thomas/** includes all services required by THOMAS and Magentix2 platform. It also includes an example of a user web service.

[2] Magentix2 is launched without security, to enabled security refer to section 5.6.3

5.1.2 Possible Errors

If MySQL is already installed in system, the system informs about it with this error message:

```
101124 16:20:15 mysqld_safe Logging to syslog.
101124 16:20:16 mysqld_safe A mysqld process already exists
bin/mysqladmin: connect to server at 'localhost' failed
```

Moreover, if the following log is showed and /ETC/MYSQL/ directory [3] not exists, then an unstable installation of MySQL is in system. In this case, a new installation of MySQL is recommended. In section 5.3 is explained how to install MySQL manually and section 5.1 explain how to import the THOMAS schema in the current MySQL database.

```
bin/mysqladmin: connect to server at 'localhost' failed
error: 'Can't connect to local MySQL server through
socket '/tmp/mysql.sock' (2)'
Check that mysqld is running and that the socket:
'/tmp/mysql.sock' exists!
bin/mysqladmin: connect to server at 'ubuntu' failed

error: 'Lost connection to MySQL server at 'reading initial
communication packet', system error: 111'
ERROR 2002 (HY000): Can't connect to local MySQL server
through socket '/tmp/mysql.sock' (2)
```

5.2 Apache Qpid

Qpid broker is a main component of Magentix2. In this section is described how it can be installed, in case it was not desired to install Qpid together with Magentix2. Apache Qpid can be downloaded from `http://qpid.apache.org/download.cgi`

The following libraries must be installed before building the source distribution of Qpid:

- libboostiostreams 1.35dev: `http://www.boost.org` (1.35)
- e2fsprogs: `http://e2fsprogs.sourceforge.net/` (1.39)

[3] If it exists, it is recommended to delete it.

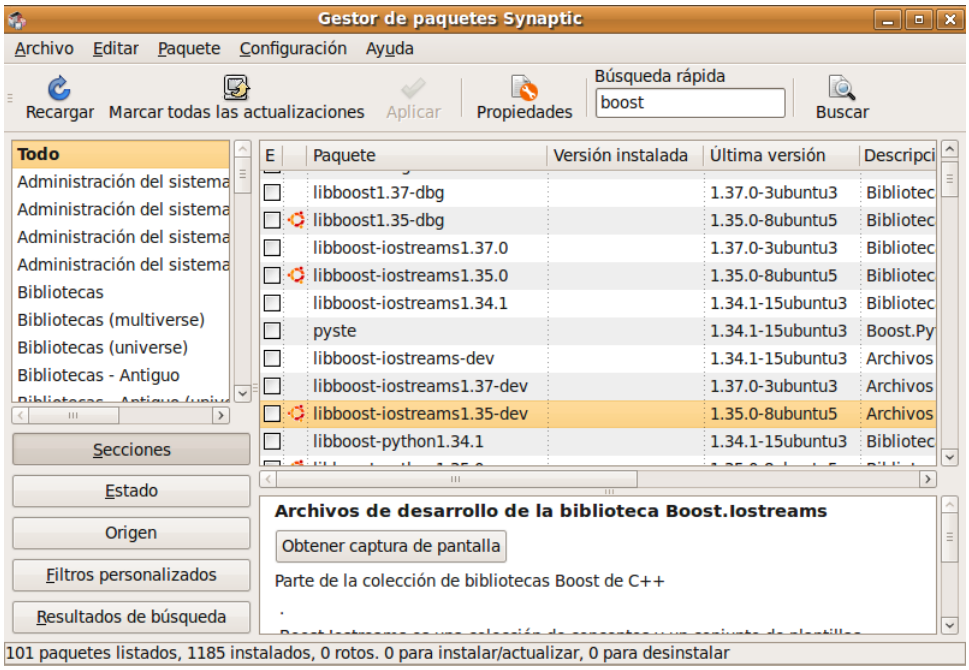

Figure 5.1: Installing libboostiostreams 1.35dev library with Synaptic tool

- pkgconfig: `http://pkgconfig.freedesktop.org/wiki/` (0.21)
- uuid 1.21.41.4
- ruby 4.2
- ruby 1.8

In Ubuntu operating systems these packages can be installed using the Synaptic package management tool, but any other package manager might be valid. As an example, figure 5.1 shows how to install the libboostiostreams 1.35dev library. Once all the required libraries have been installed [4], Qpid broker can be downloaded from: `http://qpid.apache.org/download.cgi`. To install Qpid the following steps must be performed:

- Uncompressing the donwloaded Qpid file.
- `$ ./configure --prefix= /home/hyperion/qpid` → Using the prefix option when configuring, the location where the Qpid binaries are installed can be specified (in this example case, /home/hyperion/qpid).

[4] To install a Qpid version with security support, please follow at this section 5.6.9

- `$ make install`

After a successful installation Qpid can be launched executing the following command inside the folder where Qpid was installed (in this example case, /home/hyperion/qpid): `$ ./qpidd`.

Some values Qpid broker set by default are used by some components of Magentix2, therefore if any of them is changed, it is possible that the Settings.xml file has to be also modified. This file can be found in the configuration directory of Magentix2 distribution folder. Specifically the parameters that affect Qpid configuration in the Settings.xml file are the following:

```
<entry key="host">localhost</entry>
<entry key="port">5672</entry>
<entry key="vhost">test</entry>
<entry key="user">guest</entry>
<entry key="pass">guest</entry>
<entry key="ssl">false</entry>
```

For example, if the port which Qpid broker is listening to is modified from the default 5672 to 5671, this change has to be also made on the Settings.xml file.

For those looking to adjust the Qpid broker operation there are plenty of advanced configuration options, for further information, please, refer to: `http://qpid.apache.org/books/0.7/AMQP-Messaging-Broker-CPP-Book/html/index.html`. Please note that if two ore more Qpid brokers have to work federated, a link between all the broker's amq.topic exchange has to be added.

5.3 MySQL

MySQL is a main component of the THOMAS framework. This section explains how to properly configure MySQL to work in conjunction with THOMAS. If MySQL was installed during the Magentix2 Desktop version installation, it will not be necessary to follow the steps shown here. However, if it was not installed together with Magentix2 or MySQL was already installed on the system, then this section helps to configure MySQL properly.

All the information about the organizations created with the THOMAS framework and running on the Magentix2 platform is permanently stored in a MySQL database. It is possible to create the database schema and the user employed by the THOMAS framework in MySQL by means of the the execution of the script *Import-ThomasDB.sh*. This script is located in the directory

`/bin/MySQL`, which can be found into the Magentix2 installation folder. The commands needed to execute this script are the following (from the Magentix2 root directory):

```
$ cd bin/MySQL
$ sh Import-ThomasDB.sh [MYSQL_ROOT_PASS]
```

However, it is also possible to create the THOMAS database infrastructure step by step, without using the cited script. In this case, it is necessary to load into MySQL the complete structure of the database from the *Thomas.sql* file. This file is also located into the directory `/bin/MySQL`. In order to load this file, the *MySQL Administrator* tool should be opened, and then the *Restore Backup* option must be selected, choosing the *Thomas.sql* backup file to be restored. An example of this procedure can be shown in the figure 5.2

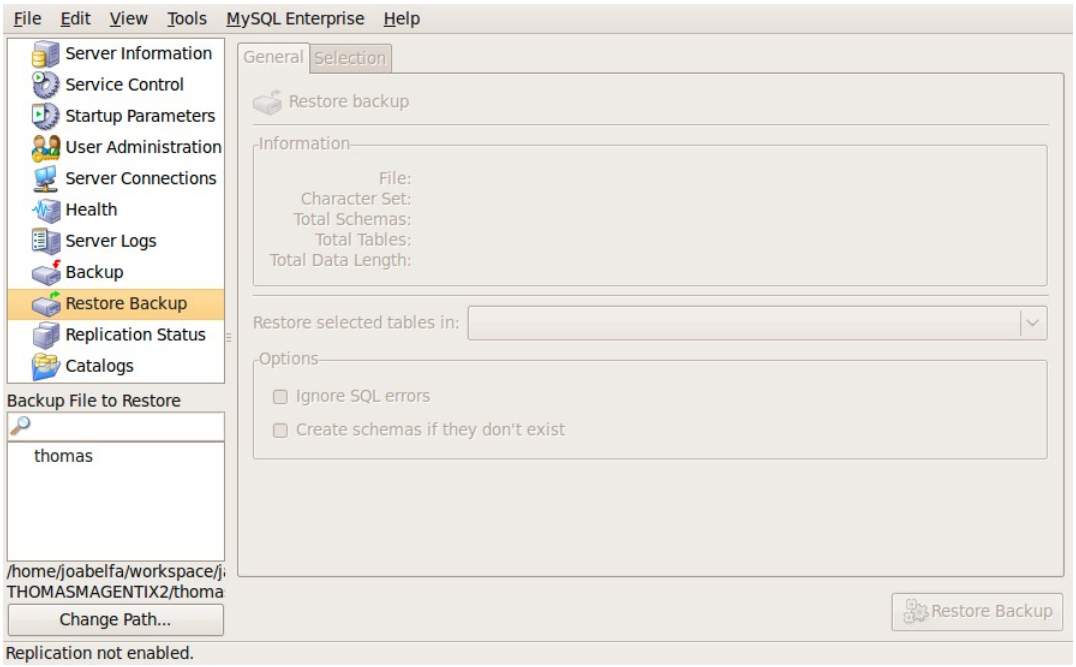

Figure 5.2: Restoring the *Thomas.sql* backup file in the *Restore Backup* option of the *MySQL Administrator*

The next required step is to add a new user to the THOMAS schema in the *User Administration* option of the *MySQL Administrator* tool (see Figure 5.3). The required fields must be fulfilled with the following information:

```
User=thomas
Password=thomas
```

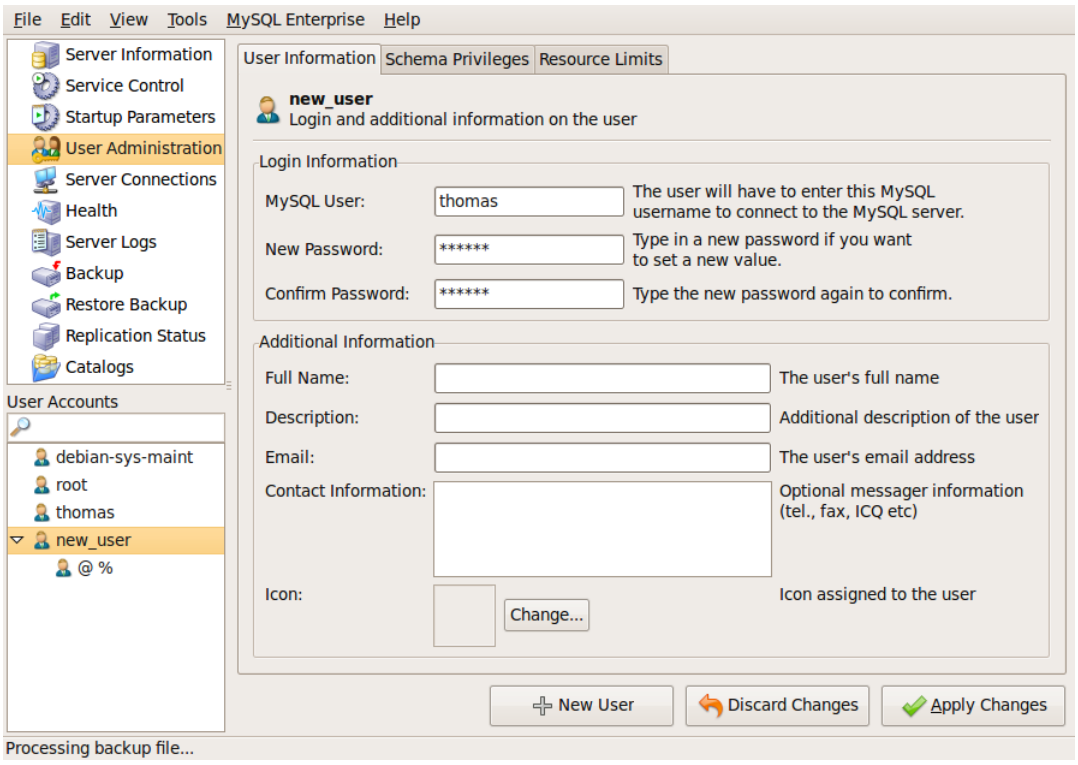

Figure 5.3: Adding the necessary user information into the THOMAS schema in the *User Administrator* option of the *MySQL Administrator* tool

The *ServerName*, *databaseName*, *userName* and *password* entries must be also configured in the settings.xml file located in the `Magentix2/Configuration` directory, as Figure 5.4 shows.

Finally, all available privileges for THOMAS tables must be assigned to the *thomas* user (Figure 5.5)

5.4 Apache Tomcat

Apache Tomcat is a main component of Magentix2, because it allows to access to standard Java web services. If Apache Tomcat was installed during the Magentix2 Desktop version installation, it will not be necessary to follow the steps shown here. On the contrary, in case Apache Tomcat was not installed together with Magentix2 or Apache Tomcat was already installed on the system, this section helps to configure it properly.

THOMAS platform is based on services (chapter 4), so SF and OMS service implementations

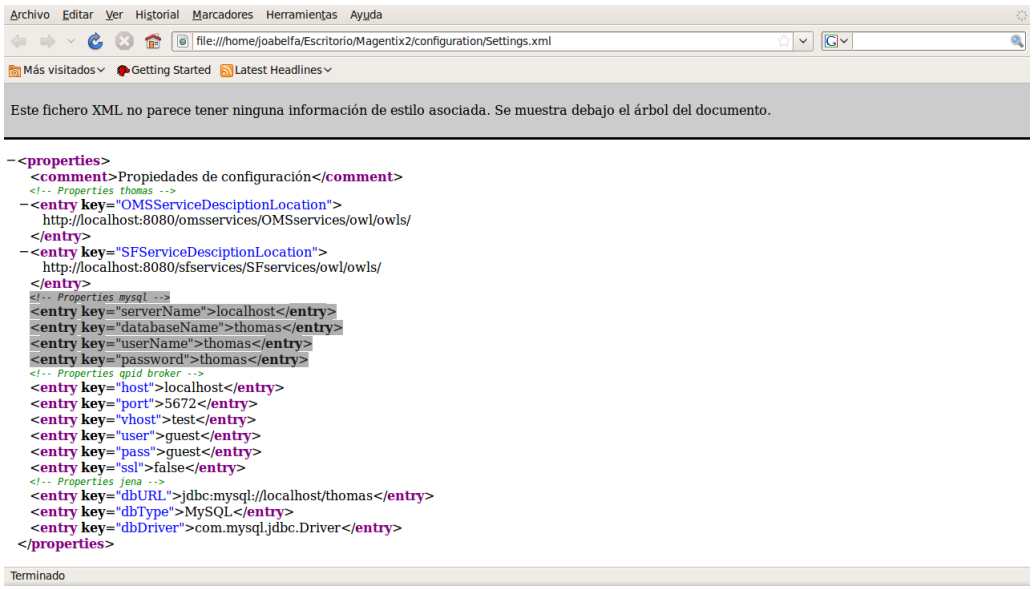

Figure 5.4: Entries of the settings.xml file than should be modified in order to work with THOMAS.

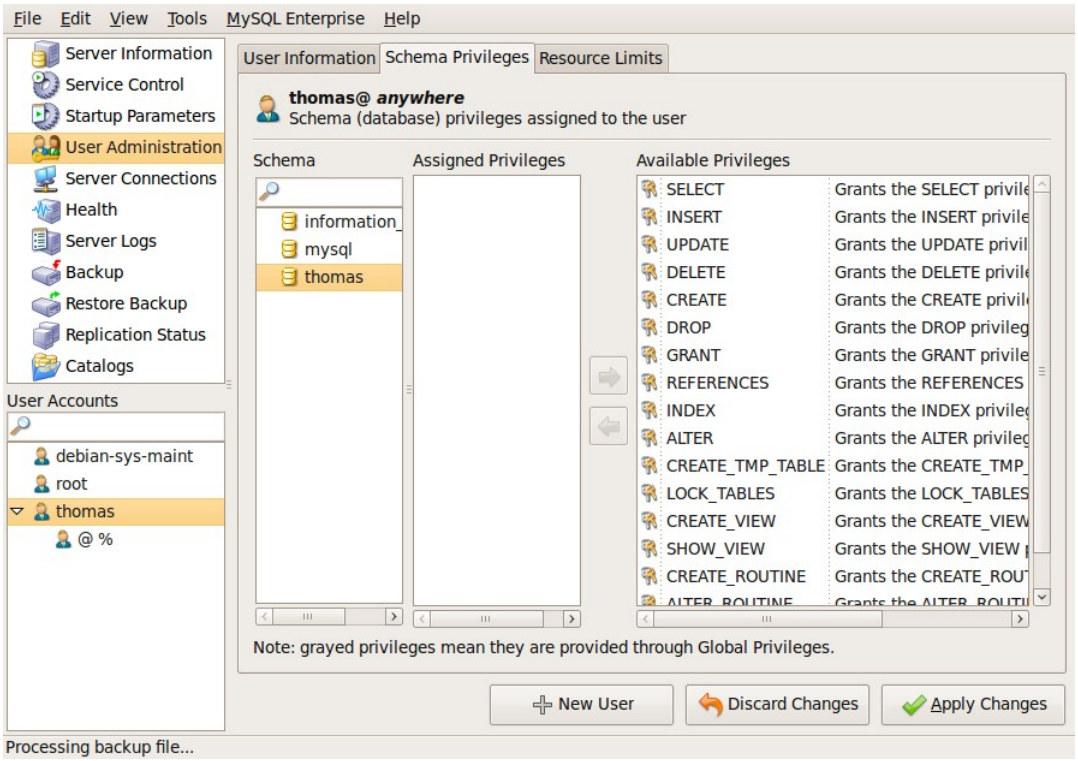

Figure 5.5: Assigning privileges to the *thomas* user in the *User Administration* option of the *MySQL Administration tool*

have to be available as standard web services. Moreover, Magentix2 offers another service named MMS (section 5.6), which is responsible of controlling the user access to the platform. This service must be also available as an standard web service. Any other user service (such as the application examples) need also to be available as standard web services. As mentioned above, Magentix2 uses Apache Tomcat to allow it.

Apache Tomcat can be downloaded from: `http://tomcat.apache.org/`. The installation instructions can be found at: `http://tomcat.apache.org/tomcat-7.0-doc/setup.html`.

Once Tomcat is installed, packaged libraries of THOMAS services (omsservices.war, sfservices.war) have to be copied from `Magentix2/thomas/` directory to the subdirectory `webapss/` of the Tomcat installation directory. Furthermore, the packaged library of the MMS service (MMS.war), which is also located at the `Magentix2/thomas/` directory, must be copied to the same `webapss/` subdirectory.

In the same way, in order to run any developed web service from Tomcat, it is necessary to copy the packaged library (serviceName.war) to the subdirectory `webapss/` of the Tomcat installation directory. In Magentix2 Desktop Version there is an example of a user web service, called *SearCheapHotel*. It is possible to run this example by means of coping the packaged library SearchCheapHotel.war (located at the `Magentix2/thomas/` directory) to the subdirectory `webapss/` of Tomcat

Once all the necessary services have been properly copied to the webapss directory, Tomcat can be started running the startup.sh file on the `/bin/` subdirectory of the Tomcat installation directory.

5.5 Platform services

This section explains how to launch platform agents without using the standard methods shown previously on this manual. This can be useful when some default parameters have been changed or if the platform runs in a distributed way.

5.5.1 Running Bridge Agents

Bridge agents are in charge of sending and receiving messages to or from foreign agents. For example, they allow Magentix2 agents to communicate with Jade agents. *BridgeAgentInOut*

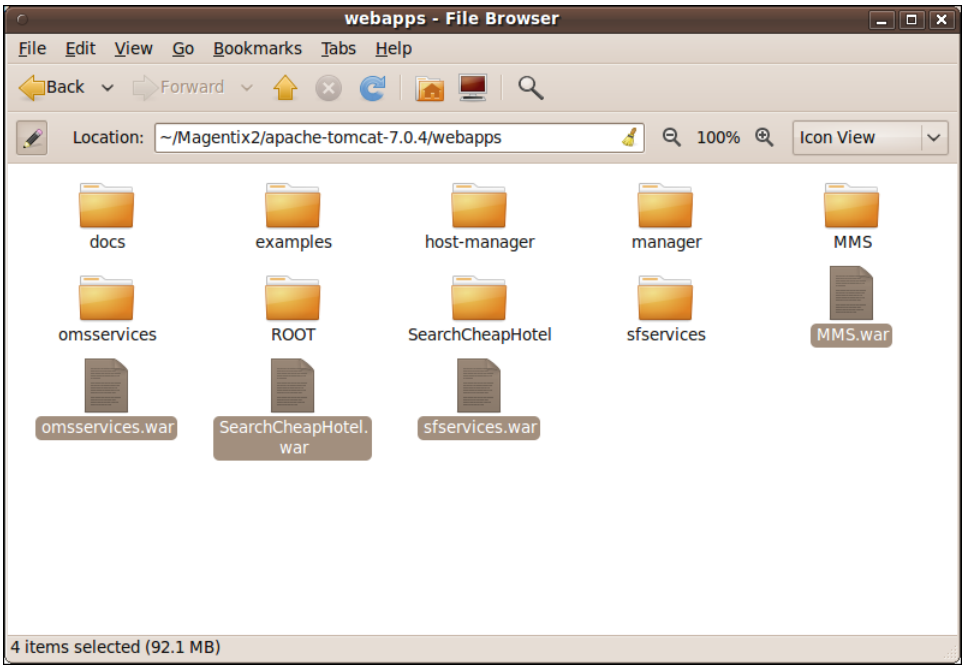

Figure 5.6: Location of web services files (*.war)

manages messages that go from inside the platform to outside, whereas *BridgeAgentOutIn* does the opposite. Bridge agents can be running on any host, they do not have to be in the same host where the QPid broker or other agents are running. A Java program has to be written and executed in order to launch bridge agents. The following code shows how to launch these agents:

```
1  import es.upv.dsic.gti_ia.core.AgentsConnection;
2  import es.upv.dsic.gti_ia.core.AgentID;
3  import es.upv.dsic.gti_ia.core.BridgeAgentInOut;
4  import es.upv.dsic.gti_ia.core.BridgeAgentOutIn;
5
6  public class Main {
7     public static void main(String[] args) throws Exception {
8        AgentsConnection.connect();
9        private BridgeAgentInOut inOutAgent;
10       private BridgeAgentOutIn outInAgent;
11       inOutAgent = new BridgeAgentInOut(new AgentID("
             BridgeAgentInOut"));
12       outInAgent = new BridgeAgentOutIn(new AgentID("
             BridgeAgentOutIn"));
13       inOutAgent.start();
```

```
14          outInAgent.start();
15      }
16  }
```

In the code shown above when the bridge agents are created (lines 11-12) they receive a new AgentID as argument. This new AgentID gets only one argument, the name of the agent. Platform agents, like bridge agents, must always have a well known name. For bridge agents the names must be: "BridgeAgentInOut" and "BridgeAgentOutIn" respectively.

In line 8 the connection of the agents to the platform is set using the method `AgentsConnection.connect()`. The parameters for this connection are specified in the configuration file `Settings.xml`. The method `AgentsConnection.connect()` should not be called if the platform security is enabled.

Once the agents have been created with the desired parameters, both are started (lines 13 & 14). This Java program has to be manually executed when starting Magentix2 platform.

5.5.2 Running OMS and SF Agents

OMS and SF agents provide all the services of Thomas framework. These agents can be running on any host, they don't have to be in the same host where the QPid broker or other agents are running. A Java program has to be written and executed in order to launch OMS and SF agents. The following code shows how to launch these agents:

```
1  import es.upv.dsic.gti_ia.architecture.Monitor;
2  import es.upv.dsic.gti_ia.core.AgentID;
3  import es.upv.dsic.gti_ia.core.AgentsConnection;
4  import es.upv.dsic.gti_ia.organization.OMS;
5  import es.upv.dsic.gti_ia.organization.SF;
6  import es.upv.dsic.gti_ia.core.AgentID;
7
8  public class Main {
9     public static void main(String[] args) throws Exception {
10       AgentsConnection.connect();
11       OMS agentOMS = OMS.getOMS();
12       SF agentSF = SF.getSF();
13       agentOMS.start();
14       agentSF.start();
15    }
```

```
16 }
```

In the code shown above the agents OMS and SF are created (lines 11-12). The creation of these agents does not require any parameter.

In line 10 the connection of the agents to the platform is set using the method `AgentsConnection.connect()`. The parameters for this connection are specified in the configuration file `Settings.xml`. The method `AgentsConnection.connect()` should not be called if the platform security is enabled.

Once the agents have been created with the desired parameters, both are started (lines 13 & 14). This Java program has to be manually executed when starting Magentix2 platform.

5.6 Configuring security

5.6.1 Introduction

Figure 5.7 shows an overview of the Magentix2 security infrastructure. Magentix2 Management Service (MMS) is a entity of Magentix2 that manages a part of the security of the platform (is located at the right of Figure 5.7). In fact, MMS is responsible of controlling the user accesses to the platform. Users will be present a certificate issued by a trusted third party certificate authority (the red one in Figure 5.7). Then, using the Magentix Certificate Authority (the blue one in Figure 5.7), the MMS issues certificates for the agents (blue certificates in Figure 5.7). Once an agent gets a valid certificate issued by the Magentix Authority Certificate, it can communicate to other agents in a secure way.

The Magentix2 communication is based on AMQP standard. For supporting this standard, the Apache Qpid open source implementation is used. So, the Qpid broker must be configured in order to support security. For this purpose, the broker will be use a certificate which issues for the Magentix2 Certificate Authority. Thus, agents in Magentix2 will communicate with each other in a secure way using their agent certificates, which allow to authenticate with the Qpid broker. Furthermore, this broker can also communicate with them using his certificate.

This section explains how to create the Magentix2 Certificate Authority and certificates for the Qpid broker and MMS. Moreover, how to configure MMS is also explained. Finally, it is explained how to install, configure and run the Qpid broker with security support.

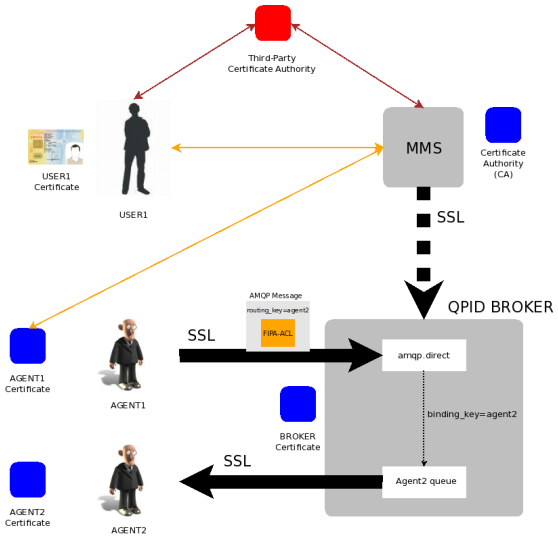

Figure 5.7: Magentix2 Security Infrastructure

5.6.2 Supported features

In order to ensure the security, the Magentix2 secure module supports the follow features:

1. **Authentication:** The security module allows agents running on the Magentix2 platform to show other agents that effectively they are the agents they claim. This characteristic is the basis for all other features.

2. **Integrity:** The messages that one agent sends to another can not be manipulated by unauthorized agents while they are transmitted by the network.

3. **Confidentiality:** The messages sent thought the network can not be accessible for unauthorized agents.

4. **Non-repudiation:** If an agent receives a message of the agent B, the agent B is responsible for its message and can not negate a previous commitment or action.

5. **Privacy:** The owner of one agent which is executed in Magentix2 is not known by the rest of the agents of the platform, excepting the MMS. Therefore, the agents interact among them without know the user whom they represent.

6. **Accountability:** Although it offers privacy about the identities of the users whose agents run in Magentix2, the MMS saves a log with the relationship with the agents and the user owners. Therefore, if an agent realizes not desirable actions, his owner can be known and sanctioned as appropriate.

7. **Broker Resources Control:** The MMS is the unique entity that is authorized for administer the broker resources. The agents only are authorized to access to their queues.

8. **Control of the Interactions between Agents:** Using the ACL file of the Qpid broker, the MMS can restrict the communications between agents.

5.6.3 Using security in Magentix2 Desktop version

If Magentix2 has been installed by means of the desktop version, the security is disabled. To active the security in the system, these simple steps must be followed:

- Executing MagentixPKI script. This script creates a public key infrastructure to support the Magentix Security Infrastructure. In order to customize the process, the steps of the section 5.6.4 must be followed. The commands needed to execute the script are:

```
$ cd Magentix2/bin/security
$ sh MagentixPKI.sh CA_PASSWORD BROKER_PASSWORD MMS_PASSWORD
```

- Changing[5] the passwords in the properties configurations (securityAdmin.properties and services.xml). How to configure these properties files is explained in section 5.6.8.

  Some aspects must be considered before configure the properties:

  - the `MMS_PASSWORD` is the same to the key store and trust store.
  - In the services.xml, only `MMS_PASSWORD` is required.
  - The `TOMCAT_HOME` is $HOME/Magentix2/apache-tomcat-7.0.4.

- Adding a new trusted third party certificate authority (explained in section 5.6.6).

- Restarting the Tomcat service executing the commands:

```
$ cd Magentix2/bin/Tomcat
$ sh Stop-Catalina.sh
$ sh Start-Catalina.sh
```

- Stopping Qpid without security and restart [6] with security, executing the commands:

[5] If Magentix2 desktop installation directory is: $HOME/Magentix2, and the passwords are the same of the script, only restart Qpid with security is necessary.

[6] Qpid was executing on background.

```
$ cd ../Qpid
$ sh Stop-Qpid.sh
$ sh Start-QpidWithSecurity.sh
```

The following log is showed if the process has been done correctly:

```
2010-12-17 13:18:24 notice Listening on TCP port 5672
2010-12-17 13:18:24 notice Listening for SSL connections
on TCP port 5671
2010-12-17 13:18:24 notice Read ACL file
 "../bin/security/broker.acl"
2010-12-17 13:18:24 notice Broker running
```

5.6.4 Creating certificates

In this section, how to create[7] a Magentix2 Public Key Infrastructure manually is explained. These steps can make a more custom configuration process (indicating the type of algorithm, duration of validity, etc.). The basic components that can be configured are:

- Magentix Certificate Authority (MagentixCA).
- Qpid broker.
- Magentix2 Management Service (MMS).

Firstly, it is recommended to create the PKI infrastructure in a new folder (but it is optional).

```
$ mkdir security
$ cd security
```

5.6.4.1 Magentix Certificate Authority (MagentixCA)

The following steps show how to create MagentixCA certificates.

- **Creating a new certificate database in the CA_db directory :**

[7] For this purpose, certutil (`http://www.mozilla.org/projects/security/pki/nss/tools/certutil.html`) and keytool (`http://download.oracle.com/javase/1.3/docs/tooldocs/win32/keytool.html`) commands are used.

```
$ mkdir CA_db
$ certutil -N -d CA_db
```

In this step, the password for CA is introduced. Afterwards, it will be referenced as PASSWORD_CA.

- **Creating a Self-signed Root CA certificate, specifying the subject name for the certificate (MagentixCA):**

```
$ certutil -S -d CA_db -n "MagentixTheCA" \
 -s "CN=MagentixCA,O=Magentix" -t "CT,," -x -2
```

The PASSWORD_CA is required. The system asks about same aspects, and the following answers must be proportioned:

  - Typing “y” for “Is this a CA certificate [y/N]?”
  - Pressing enter for “Enter the path length constraint, enter to skip [<0 for unlimited path]:“
  - Typing ”n” for ”Is this a critical extension [y/N]?”

- **Extracting to a file the CA certificate from the CAs certificate database:**

```
$ certutil -L -d CA_db -n "MagentixCA" \
 -a -o CA_db/rootca.crt
```

5.6.4.2 Qpid broker

A certificate identifies the broker, which will be signed by the Magentix CA. The following steps show how to create Qpid certificates.

- **Creating a certificate database for the Qpid Broker:**

```
$ mkdir broker_db
$ certutil -N -d broker_db
```

In this step the password for the Qpid broker is introduced. Afterwards, it will be referenced as PASSWORD_BROKER.

- **Importing the CA certificate into the brokers certificate database, and marking it trusted for issuing certificates for SSL client and server authentication:**

```
$ certutil -A -d broker_db -n "MagentixCA" -t "TC,," \
   -a -i CA_db/rootca.crt
```

The PASSWORD_BROKER is required.

- **Creating the server certificate request, specifying the subject name for the server certificate.**

```
$ certutil -R -d broker_db -s "CN=broker,O=Magentix" \
   -a -o broker_db/server.req
```

The PASSWORD_BROKER is required.

Note that this step generates the servers private key, so it must be done in the servers database directory.

- **This step simulates the CA signing and issuing a new server certificate based on the servers certificate request:**

```
$ certutil -C -d CA_db -c "MagentixCA" \
  -a -i broker_db/server.req \
  -o broker_db/server.crt -2 -6
```

The new certificate is signed with the CAs private key, so this operation uses the CAs databases (The PASSWORD_CA is required). For leaving the new certificates of the servers in a file the following actions must be performed:

  - Selecting "0 - Server Auth" at the prompt
  - Pressing 9 at the prompt
  - Typing "n" for "Is this a critical extension [y/N]?"
  - Typing "n" for "Is this a CA certificate [y/N]?"
  - Entering "-1" for "Enter the path length constraint, enter to skip [<0 for unlimited path]: >"
  - Typing "n" for "Is this a critical extension [y/N]?"

- **Importing (adding) the new server certificate to the brokers certificate database in the server_db directory with the broker nickname:**

```
$ certutil -A -d broker_db -n broker -a \
 -i broker_db/server.crt -t ",,"
```

- **Verifying if the certificate is valid:**

```
$ certutil -V -d broker_db -u V -n broker
```

  At the end, the following message must be showed: certificate is valid.

5.6.4.3 Magentix2 Management Service

This certificate identifies the MMS, which will be signed by the Magentix CA. In this section, how to create a new MMS keystore and trustore is explained.

- **Importing the CA certificate in to the trust store and import the CA certificate in to the key store (need for client authentication):**

```
$ keytool -import -trustcacerts -alias MagentixCA \
-file CA_db/rootca.crt -keystore MMSkeystore.jks
$ keytool -import -trustcacerts -alias MagentixCA \
-file CA_db/rootca.crt -keystore MMStruststore.jks
```

  In this step, the password for key store and trust store are introduced. Afterwards, will be referenced as `PASSWORD_MMSKEYSTORE` and `PASSWORD_MMSTRUSTSTORE` respectively.

- **Generating keys for the MMS certificate:**.

```
$ keytool -genkey -alias MMS -keyalg RSA\
        -dname "CN=MMS,O=Magentix" \
        -keystore MMSkeystore.jks
```

  Press enter when prompted for the password to select the same password as the key-store.

- **Creating a certificate request:**

```
$ keytool -certreq -alias MMS -keystore MMSkeystore.jks \
   -file mms.csr
```

The PASSWORD_MMSKEYSTORE is required.

- **Signing the certificate request using CA certificate:**

```
$ certutil -C -d CA_db/ -c "MagentixCA" -a -i mms.csr \
 -o mms.crt
```

The PASSWORD_CA is required.

- **Importing the certificate into the key store:**

```
$ keytool -import -trustcacerts -alias MMS -file mms.crt \
         -keystore MMSkeystore.jks
```

The PASSWORD_MMSKEYSTORE is required.

5.6.5 Exporting the MMS certificate with the public Key

This command is used to export the MMS public key into a file. It will be sent to users and they will must be imported into his trustores. Thereby, the security context (mutual authentication) with the MMS and users will be created.

```
$ keytool -export -alias MMS -file mms.crt \
 -keystore MMSkeystore.jks
```

The PASSWORD_MMSKEYSTORE is required.

5.6.6 Importing new trusted third party certificate authority

The following command is used to import a new trusted third party certificate authority. Thereby, all users with certificate issues by these trusted third party can be communicate in a security context (mutual authentication) with the MMS.

```
$ keytool -import -trustcacerts -alias FNMT \
  -file FNMTClase2CA.cer -keystore MMSkeystore.jks
```

<*alias*>: user name identifying

<*file*>: certificate with user public key[8].

The PASSWORD_MMSKEYSTORE is required.

5.6.7 Tracing Service

In the platform, only a user authorized can be launch a trace manager agent (TM) (explained in section 3.7), to avoid the supplanting of this user/agent. The functionality of MMS has been limited in order to not issue a certificates for a TM user. Therefore, the certificate for TM agent will be created manually by means of the following steps:

- **Importing the CA certificate in to the key store (needed for client authentication):**

  ```
  $ keytool -import -trustcacerts -alias MagentixCA  \
    -file CA_db/rootca.crt -keystore keystore.jks
  ```

  The PASSWORD_USERKEYSTORE is required.

  <*file*>: the rootca.crt. Section 5.6.4.1 shows how to obtain this file.

- **Generating keys for the TM certificate:**

  ```
  $ keytool -genkey -alias tm -keyalg RSA \
   -dname "CN=tm,O=Magentix" -keystore keystore.jks
  ```

  Pressing enter when prompted for the password to select the same password as the keystore.

  The PASSWORD_USERKEYSTORE is required.

- **Creating a certificate request:**

  ```
  $ keytool -certreq -alias tm -keystore keystore.jks
          -file tm.csr
  ```

  The PASSWORD_USERKEYSTORE is required.

[8]For example, to accept users issues for the Fabrica Nacional de Moneda y Timbre (FNMT) download and import this certificate (http://www.cert.fnmt.es/index.php?cha=cit&sec=4&page=139&lang=es).

- **Signing the certificate request using a CA certificate:**

```
$ certutil -C -d CA_db/ -c "MagentixCA" -a -i tm.csr \
         -o tm.crt
```

The PASSWORD_CA is required.

- **Importing the certificate into the key store:**

```
$ keytool -import -trustcacerts -alias tm -file tm.crt \
        -keystore keystore.jks
```

The PASSWORD_USERKEYSTORE is required.

5.6.8 Magentix2 Management System (MMS)

The MMS is distributed as a war project (Web Application aRchive). In webapps directory, where it is deployed, exists a MMS folder. In this folder, the configuration files are located.

Once the certificate infrastructure has been created, the properties files will have to be configured correctly. If the Magentix2 Desktop version has been installed, default options are correctly configured.

It should be remember that the root directory is TOMCAT_HOME/bin. for the relative path

- nss.cfg

  This file is in the folder: TOMCAT_HOME/webapps/MMS/WEB-INF/classes/.

  In this file, the path of Magentix2 Certificate Authority (CA_db) is represented. Only change the *nnsSecmondDirectory* if the CA_db is in other path than Magentix2/bin/security, as it can be shown in the following example (Figure 5.8).

```
name = NSSkeystore
nssLibraryDirectory = /usr/lib
nssSecmodDirectory = ../bin/security/CA_db
nssModule = keystore
```

Figure 5.8: An example of the nss.cfg file.

- services.xml

  This file is in: TOMCAT_HOME/webapps/MMS/WEB-INF/services/MMS/META-INF/.

  In this file, the rampart properties are configured. The Figure 5.9 shows an example of this file.

```
<ramp:signatureCrypto>
  <ramp:crypto provider="org.apache.ws.security.components
     .crypto.Merlin">
     <ramp:property name="org.apache.ws.security.crypto
     .merlin.keystore.type">
            JKS
     </ramp:property>
     <ramp:property name="org.apache.ws.security.crypto
     .merlin.file">
            ../bin/security/MMSkeystore.jks
     </ramp:property>
     <ramp:property name="org.apache.ws.security.crypto
     .merlin.keystore.password">
            MMS_PASSWORD
     </ramp:property>
 </ramp:crypto>
</ramp:signatureCrypto>
```

Figure 5.9: An example of the services.xml file.

- securityAdmin.properties

  This file is in the directory: TOMCAT_HOME/webapps/MMS/WEB-INF/classes/.

  - Setting the path and password to MMS key and trust store[9] (Figure 5.10).
  - Changing the algorithm type for encryption and validity time (in days) for the certificates (Figure 5.11).
  - Setting the base path for accessing to the ACL broker (Figure 5.12).
  - Setting the base path for logging the registers User/Agent (Figure 5.13). By default, the path is TOMCAT_HOME/logs.
  - Setting the properties for the Magentix Certificate Authority [10] (Figure 5.14).
  - Setting the base path for accessing to the configuration nss file (Figure 5.15).
  - Setting the properties of the Qpid broker (Figure 5.16).
  - Setting the values of the certificates (Figure 5.17).

[9] How to create the MMS keystore and trustore was explained in section 5.6.4.3

[10] The process to create Magentix CA was explained in section 5.6.4.1.

```
<!-- Only MMS Administrator -->


# set the base path for accessing keystore user
KeyStorePath=../bin/security/MMSkeystore.jks
#set the password to keystore
KeyStorePassword=MMS_PASSWORD

# set the base path for accessing truststore user
TrustStorePath=../bin/security/MMStruststore.jks
#set the password to truststore
TrustStorePassword=MMS_PASSWORD
```

Figure 5.10: First section of the securityAdmin.properties file.

```
#set the type of algorithm for the signature certificates
sigAlg=SHA1WithRSA
#set the validity time for the certificates (days)
Validity=90
```

Figure 5.11: Second section of the securityAdmin.properties file.

```
# set the base path for accessing to broker acl.
ACLPath=../bin/security/broker.acl
```

Figure 5.12: Third section of the securityAdmin.properties file.

```
#set the base path for log the registers User / Agent.
Userlog=logs/MMSregisters.log
```

Figure 5.13: Fourth section of the securityAdmin.properties file.

```
#set the alias to root certificate
aliasCA=MagentixCA
#set the password to root nss db
password=PASSWORD_CA
#set the type to root nss db
type=PKCS11
```

Figure 5.14: Fight section of the securityAdmin.properties file.

```
#set the base path for accessing to configuration file nss.
pathnsscfg=webapps/MMS/WEB-INF/classes/nss.cfg
```

Figure 5.15: Sixth section of the securityAdmin.properties file.

```
#set the properties of qpid broker
host=localhost
port=5671
vhost=test
user=guest
pass=guest
ssl=true
saslMechs=EXTERNAL
```

Figure 5.16: Seventh section of the securityAdmin.properties file.

```
#set the values of certificates
organizationalUnit=Magentix
organization=orgnization
city=city
state=state
country=ES
```

Figure 5.17: Eighth section of the securityAdmin.properties file.

– Setting the Rampart configuration properties (Figure 5.18).

```
#Rampart config: set the alias value of certificate for
sign the messages.
alias=mms
#set the password of mmskeystore
key=PASSWORD_MSSKEYSTORE
```

Figure 5.18: Ninth section of the securityAdmin.properties file.

Tomcat must be restarted in order to update changes. Therefore, the following commands must be executed. If Magentix2Desktop is installed:

```
$ cd Magentix2/bin/Tomcat
$ sh Stop-Catalina.sh
$ sh Start-Catalina.sh
```

In other cases:

```
$ cd TOMCAT_HOME/bin
$ sh ./shutdown.sh
$ sh ./startup.sh
```

5.6.9 Qpid broker with security support

5.6.9.1 Installing

In this section is explained how to install Qpid with security. If the Magentix2Desktop has been installed and Qpid was selected for install, it is not needed execute this step. In that case, it is only necessary to execute the script bin/Qpid/Start-QpidWithSecurity.sh in order to execute Qpid with security.

For installing the Qpid dependencies, please refer to the section 5.2. Once the basic dependencies are resolved, follow the next steps:

1. Installing the following libraries before building the source distribution of Qpid with security (SSL and SASL must be installed) :

   - libnss3-1d
   - libnspr-dev
   - libnss3-dev
   - libnss3-tools
   - libsasl2-dev
   - sasl2-bin

   On Ubuntu operating systems these packages can be installed using the Synaptic package management tool, but any other package manager might be valid.

2. Downloading [11] the 919487 Qpid revision, which already supports the security EXTERNAL mechanism.

   ```
   $ svn -r 922479 co http://svn.apache.org/repos/asf/qpid/trunk
   ```

[11] To install subversion, the following command should be used: sudo apt-get install subversion.

3. Accessing to trunk/qpid/cpp directory.

4. Executing the bootstrap command [12].

```
$ ./bootstrap
```

5. Configuring the package:

```
$ ./configure --prefix=QPID_INSTALL_DIR --with-ssl --with-sasl
```

<prefix>: Selecting the directory where Qpid will be installed.

6. Compiling and installing the package:

```
$ make install
```

5.6.9.2 Executing

Once the infrastructure of certificates is correctly configured and Qpid broker is installed with security support, only it remains to execute the broker Qpid with security.

In sbin directory of Qpid installation, this command must be executed:

```
$ ./qpidd --auth yes --ssl-cert-db security/broker\_db/ \
--ssl-cert-name broker --ssl-require-client-authentication \
--acl-file security/broker.acl
```

<ssl-cert-db>: directory where is the broker database (broker_db).

<ssl-cert-name>: alias: broker.

<acl-file>: directory where is the broker.acl file.

The `PASSWORD_BROKER` is required.

The following log is showed if the process finishes correctly:

[12] Autoconf and libtool are required.

```
2010-12-17 13:18:24 notice Listening on TCP port 5672
2010-12-17 13:18:24 notice Listening for SSL connections on
TCP port 5671
2010-12-17 13:18:24 notice Read ACL file "../bin/security/broker.acl"
2010-12-17 13:18:24 notice Broker running
```

Bibliography

Bordini, R., Hübner, J., and Vieira, R. (2005). Jason and the Golden Fleece of agent-oriented programming. *Multi-Agent Programming*, pages 3–37.

Búrdalo, L., Terrasa, A., Julián, V., and García-Fornes, A. (2010). TRAMMAS: A Tracing Model for Multiagent systems. In *First International Workshop on Infraestructures and Tools for Multiagent Systems*, pages 42–49.

FIPA (2002). *FIPA Request Interaction Protocol Specification*. FIPA.

Fogués, R. L., Alberola, J. M., Such, J. M., Espinosa, A., and Garca-Fornes, A. (2010). Towards Dynamic Agent Interaction Support in Open Multiagent Systems. In *Proceedings of the 13th International Conference of the Catalan Association for Artificial Intelligence*, volume 220, pages 89–98. IOS Press.

Rao, A. S. (1996). Agentspeak(l): Bdi agents speak out in a logical computable language. In *Modelling Autonomous Agents in a Multi-Agent World - MAAMAW*, pages 42–55. Springer-Verlag.

Val, E. D., Criado, N., Rebollo, M., Argente, E., and Julian, V. (2009). Service-Oriented Framework for Virtual Organizations. In *International Conference on Artificial Intelligence (ICAI)*, volume 1, pages 108–114. CSREA Press.