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 first Magentix platform. 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 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 8 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 10 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

2.1.1 Requirements

- Oracle Java Development Kit (JDK) 7 or later[1].
- Apache Tomcat 7 or later[2]
- MySQL server 5.0 or later[3]

2.1.2 Installation of Magentix2

In order to install Magentix2, the corresponded zipped package must be downloaded[4]. Once Magentix2 is downloaded, you need to unzip the file. Just double-click the file or run the following command:

```
$ unzip magentix2-2.1.zip
```

[1] http://www.oracle.com/technetwork/java/archive-139210.html
[2] http://tomcat.apache.org/download-70.cgi
[3] http://www.mysql.com/downloads/
[4] The latest installable version is in: http://gti-ia.upv.es/sma/tools/magentix2/downloads.php

In the unzipped directory you have now the Magentix2 agent platform™ready to be configured and started. Before running the Magentix2 platform you need to finish the installation by executing the setup script.

- In Linux and MacOS X run the following command:

```
$ ./magentix-setup.py
```

- In Windows XP, Windows 7 and Windows 8 double-click the `magentix-setup.exe` file.

Then, a text interface for the installation is provided. This setup process will ask you some required users and passwords to configure and deploy the components needed by Magentix2. It will also check taht all the required dependencies are installed and properly configured. The setup script will ask you for the following data:

- MySQL root password: used to create the magentix user and create the database schema for the platform.
- Tomcat user and password: used to deploy the Magentix webservices to the tomcat app server. Note that the tomcat user **MUST** have the `manager-script` role to be able to deploy webapps. To do this add the following lines to your `tomcat-users.xml` file (located where your tomcat installation is or at `/etc/tomcat`), where USERNAME and PASSWORD must be changed by the user and password you assign to tomcat:

```
<role rolename="manager-script"/>
<user username="USERNAME" password="PASSWORD"
                          roles="manager-script"/>
```

If every dependency is correctly installed and running the setup script will finish with the message:

```
"Magentix succesfully installed."
```

When installation ends, you can run the Start-Magentix script in order to start the services and platform agents. this script must be executed as follows:

- In Linux and MacOS X run the following command:

```
$ ./Start-Magentix.sh
```

- In Windows XP, Windows 7 and Windows 8:

```
> Start-Magentix.bat
```

To check that all is correctly configured and Magentix2 has been successfully installed and running, execute the example Start-BasicExample.sh. This script must be executed as follows:

```
$ cd  examples
$ ./Start-BasicExample.sh
```

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, the following folders are created:

- **/** magentix root directory: 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* and *Start-Magentix.bat*: it launches the Qpid server and the platform agents (OMS, SF, TM and bridge agents).
  - *Stop-Magentix.sh* and *Stop-Magentix.bat*: it stops Qpid and the platform agents (OMS, SF, TM and bridge agents). The commands needed to execute this script are:

```
$ ./Stop-Magentix.sh
```

  - *LICENSE.txt*: includes the license under Magentix2 is distributed. Magentix2 is licensed under the GNU LESSER GENERAL PUBLIC LICENSE[5].
  - *RELEASE_NOTES*: includes the changelog of the last releases of the platform.
  - *magentix-setup.py* and *magentix-setup.exe*: configure magentix to be launched at first time.

- **configuration/** sub-directory: includes the Settings.xml and loggin.xml configuration files, necessary to launch Magentix2 user agents.
- **docs/** includes the API in html format and the Magentix2 User manual in pdf format.
- **lib/** includes Magentix2 library an all additional libraries required by Magentix2. How to import this library in projects is showed in section 2.2.
- **examples/** includes some examples of Magentix2 agents implementation.
- **src/** includes Magentix2 sources.
- **webapps/** includes all services required by THOMAS and Magentix2. It also includes a user web service example.
- **bin/** includes some executables needed by Magentix2 to run.

2.1.4 Unistalling Magentix2

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

1. Stopping Magentix2 platform.

   ```
   $ ./Stop-Magentix.sh
   ```

2. Deleting Magentix2 installation directory.

[5] http://www.gnu.org/copyleft/lesser.html

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 jdk1.7 which is available at: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

The first step is to start Eclipse and create a new project (MyFirstAgent). The java library `magentix2-2.1-jar-with-dependencies.zip` 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 the QPid broker. `Loggin.xml` is the configuration file for the Magentix2 logger, where is specified how 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 Magentix2 installation folder. In this example project, this configuration folder will be used. Thus, it is only necessary to copy the folder `configuration/` in the root folder of the project (in this case `/workspace/MyFirstAgent/}`). Then, it is necessary to create a new package named *agent* in 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: *Sender* and *Consumer*. The agent *Sender* sends a message to agent *Consumer*, who writes 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.

Now, how to program the Sender.java class is shown. This class has to extend BaseAgent class (section 3.1.1). Therefore, it is necessary to import some classes from magentix2-2.1-jar-with-dependencies.zip. Once the library is included, Eclipse suggests you to import the necessary classes that library. Figure 2.2 shows Sender.java at that 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

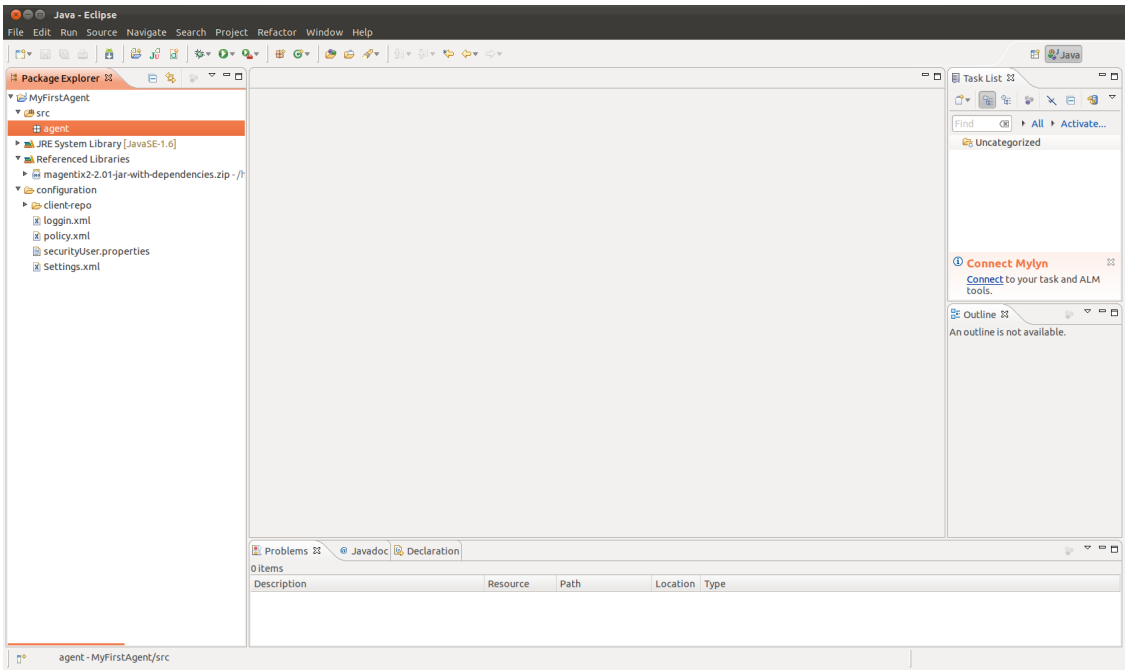

Figure 2.1: Project and package creation

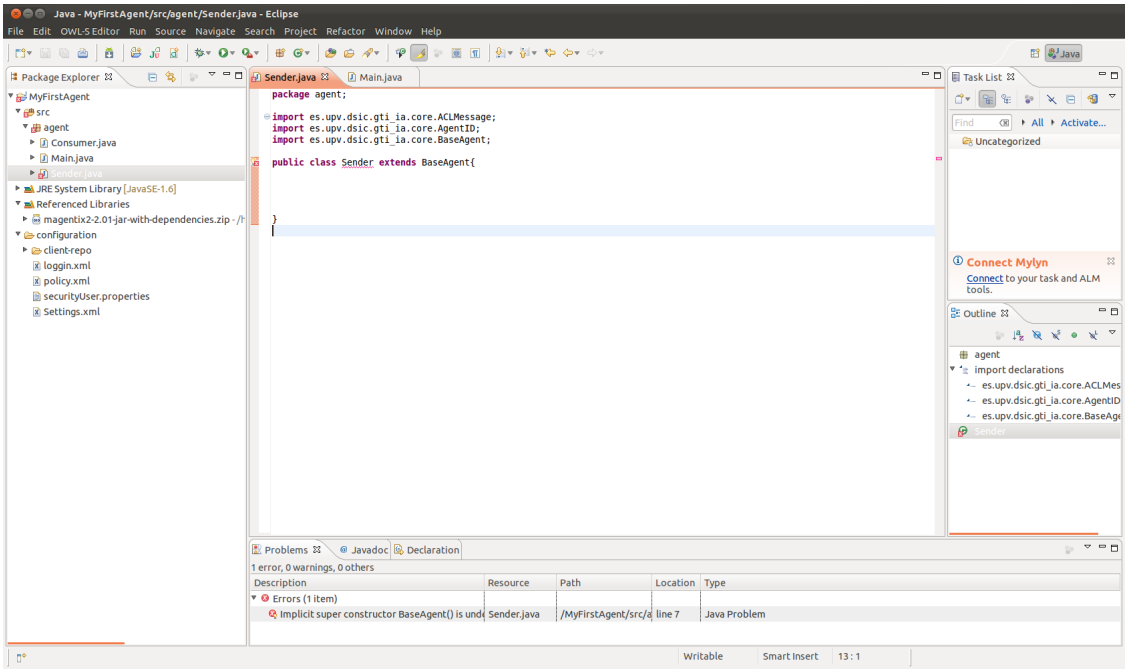

Figure 2.2: Programming a first agent with Eclipse

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 it 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;
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 *Consumer* agent 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 it ends its execution. The code of the *Consumer* 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;
2 import es.upv.dsic.gti_ia.core.AgentID;
3 import es.upv.dsic.gti_ia.core.BaseAgent;
4
5 public class SenderAgent extends BaseAgent {
6
```

```
7     public SenderAgent(AgentID aid) throws Exception {
8           super(aid);
9     }
10
11    public void execute() {
12          System.out.println("Executing, I'm " + getName());
13
14    }
15
16  }
```

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:

```
1   public void execute() {
2     /**
3     * This agent has no definite work. Wait infinitely the arrival of
4     * new messages.
5     */
6     try {
7           /**
8            * receiveACLMessage is a blocking function.
9            * its waiting a new ACLMessage
10           */
11          ACLMessage msg = receiveACLMessage();
12
13          System.out.println("Mensaje received in " + this.getName()
14                  + " agent, by receiveACLMessage: " + msg.getContent
                        ());
15    } catch (Exception e) {
```

```
<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

```
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 loggin.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 the `configuration/` directory of the Magentix2 installation.

```
<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 **??**.

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:

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

```
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");
```

3.1.5 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 Agent life cycle

The agent life cycle for a BaseAgent is composed of the following steps: `init()-> execute()-> finalize()-> terminate()`

These methods are defined in the `es.upv.dsic.gti_ia.core.BaseAgent` class. The `init()` and `finalize()` methods are automatically executed before and after the `execute()` method. The programmer can override them in order to include initialization or termination tasks. The `terminate()` method **MUST NOT** be overriden! since it terminates the qpid conections with the broker. Otherwise the agent will not finalize correctly. All the user code designated to stop the agent must be placed in the `finalize()` method.

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.
- 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.

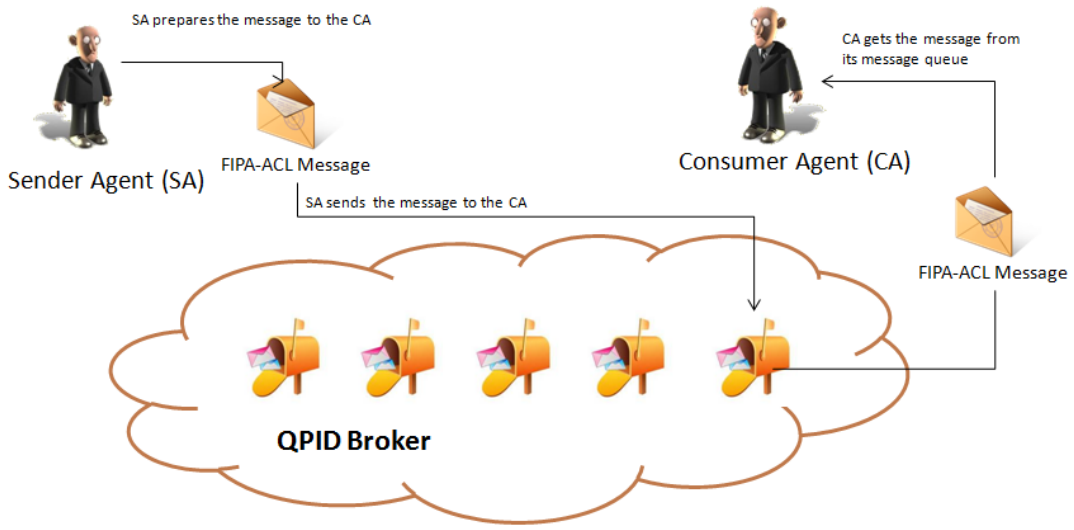

Figure 3.3: Messages exchange thought QPID Broker in Magentix2

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.
- Performative: REQUEST, INFORM, QUERY_IF, CFP, PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL, etc.
- Content.
- Content Language.

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

- 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               l o g g e r .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.

Notice that the *BridgeAgentInOut* and the *BridgeAgentOutIn* agents must be launched and instantiated 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* will be listening in the 8081 port. For other configurations, please refer to section 10.4.1

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"); /

```

```
10 //Creation of a Request Message
11 ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
12
13  //Addition of the receiver of the message
14 msg.addReceiver(receiver);
15
16 //Sending the message
17 send(msg);
```

CHAPTER 4

Advanced conversational agents: CAgents

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.).

4.1 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          }
```

4.2 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.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.3 Programming 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.

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

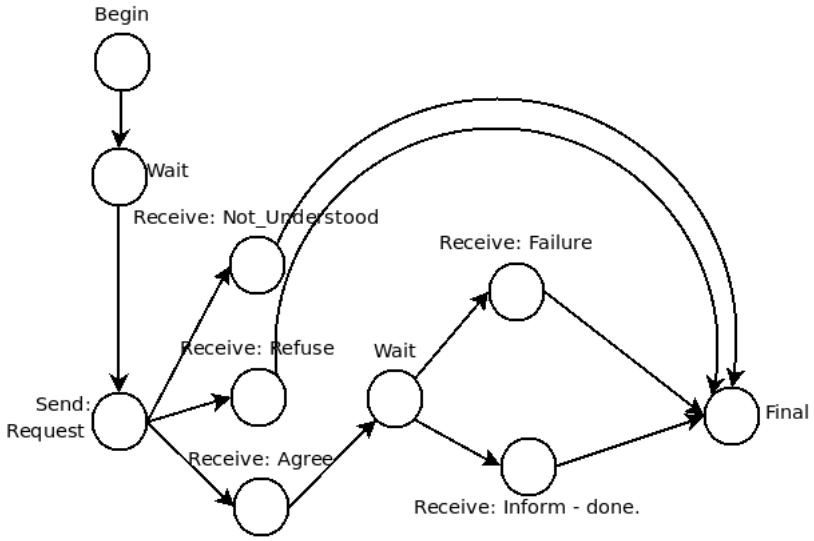

Figure 4.1: CFactory for FIPA Request Interaction Protocol for the initiator agent

dynamically modified without affecting the IP the conversation is following or other ongoing conversations which follows that IP.

In figure 4.1 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 4.2 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 *inform* 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].

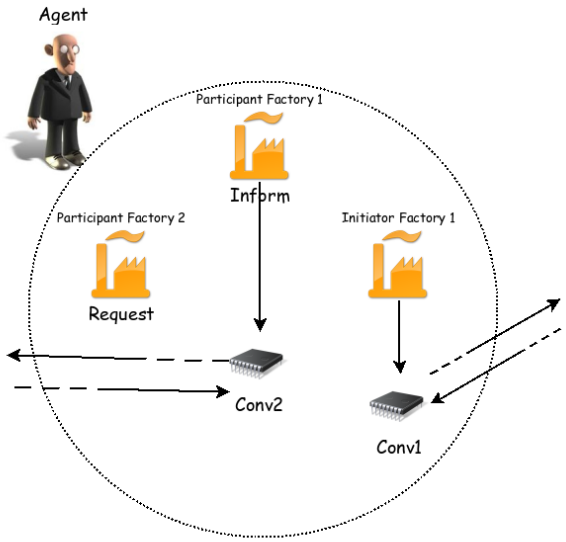

Figure 4.2: Global view of a CAgent

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.

4.4 “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 only way to terminate a cAgent is to call the `Shutdown()` method from the agent instance or the `ShutdownAgent()` method from a CProcessor instance of the CAgent. The method `finalize` is executed when the agent is just about to finish its execution, thus, this is where the user can introduce ending actions for 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  }
```

4.5 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 4.3.

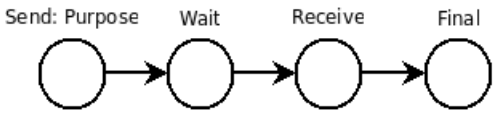

Figure 4.3: myFirstCProcessorFactories example

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());
```

4.6 Sending Errors

It is possible that a CAgent is unable to send a message due an error in the transport layer. In this case the conversation will automatically jump to the SENDING_ERRORS state, which is predefined to shutdown the agent. However, the agent programmer can override this state for their conversations to perform any other action and manage transport layer errors.

```
1 class SENDING_ERRORS_Method implements SendingErrorsStateMethod {
2
3         @Override
4         public String run(CProcessor myProcessor, ACLMessage
              errorMessage) {
5
```

```
6                   return "SHUTDOWN";
7           }
```

4.7 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.

4.8 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 4.5.

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 4.7. For this specific template the instructions are the same.

CHAPTER

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`.

5.1 Programming BDI Agents

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);
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.

```
public class SimpleArchitecture extends MagentixAgArch {

// this method just adds some perception to the agent
@Override
   public List<Literal> perceive() {
      List<Literal> l = new ArrayList<Literal>();
      l.add(Literal.parseLiteral("x(10)"));
      return l;
   }
}
```

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
```

```
7         //set the result, for example always true
8         action.setResult(true);
9         //add the executed action to the list of action executions
10        feedback.add(action);
11    }
12 }
```

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.

```
1  vl(1).
2  vl(2).
3
4  +vl(X)[source(Ag)]
5     :  Ag \== self
6     <- .print("Received tell ",vl(X)," from ", Ag).
7
8  +!goto(X,Y)[source(Ag)] : true
9     <- .println("Received achieve ",goto(X,Y)," from ", Ag).
10
11 +?t2(X) : vl(Y) <- X = 10 + Y.
12
13 +!kqml_received(Sender, askOne, fullname, ReplyWith) : true
14    <- .send(Sender,tell,"Maria dos Santos", ReplyWith). // send the
          answer
```

CHAPTER

6

Argumentative Agents

This chapter describes the *argumentative agents* API of Magentix2. This API allows agents to engage in argumentation dialogues to reach agreements about the best solution for a problem that must be solved.

First, we introduce the theory of the argumentation framework that Magentix2 argumentative agents implement. Then, the implementation details of the API are shown. Finally, a guide to run argumentative agents and an example of a call centre application are provided.

6.1 Argumentation Framework

Argumentative agents implement a case-based argumentation framework to generate arguments, to select the best argument to put forward taking into account their social context and to evaluate arguments in view of other arguments proposed in the dialogue. Also, they can use different dialogue strategies to exchange information and engage in the argumentation process. In this section we briefly introduce the framework and the dialogue strategies that Magentix2 argumentative agents use. For a more detailed explanation we refer the reader to [Heras, 2011].

6.1.1 Framework Architecture

Magentix2 argumentative agents can use a computational case-based argumentation framework to manage argumentation processes. This section outlines the main components of this framework.

We have three types of knowledge resources that the agents can use to generate, select and evaluate arguments by using our framework. These resources are implemented in Magentix2 as java clases (see 6.2):

A database of argumentation schemes with a set of schemes with the structure proposed in [Walton et al., 2008], which represent stereotyped patterns of common reasoning in the application domain where the framework is implemented. An argumentation scheme consists of a set of premises and a conclusion that is presumed to follow from them. Also, each argumentation scheme has associated a set of *critical questions* that represent potential attacks to the conclusion supported by the scheme.

A case-base with domain-cases that represent previous problems and their solutions. Agents can use this knowledge resource to generate their positions in a dialogue and arguments to support them. The *position* of an agent represents the *solution* that this agent proposes. Also, the acquisition of new domain-cases increases the knowledge of agents about the domain under discussion.

A case-base with argument-cases that store previous argumentation experiences and their final outcome. Argument-cases have three main objectives: they can be used by agents 1) to generate new arguments; 2) to select the best position to put forward in view of past argumentation experiences; and 3) to store the new argumentation knowledge gained in each agreement process, improving the agents' argumentation skills.

Argumentation Schemes

The concrete set of argumentation schemes used also depends on the application domain of our argumentation framework. The Magentix2 argumentation API only provides a basic template for them with the common components of Walton's-like argumentation schemes [Walton et al., 2008]. A user that wants to use this knowledge resource must overwrite the `ArgumentationScheme.java` class of the Magentix2 `argAgents.knowledgeResources` package.

Domain-Cases

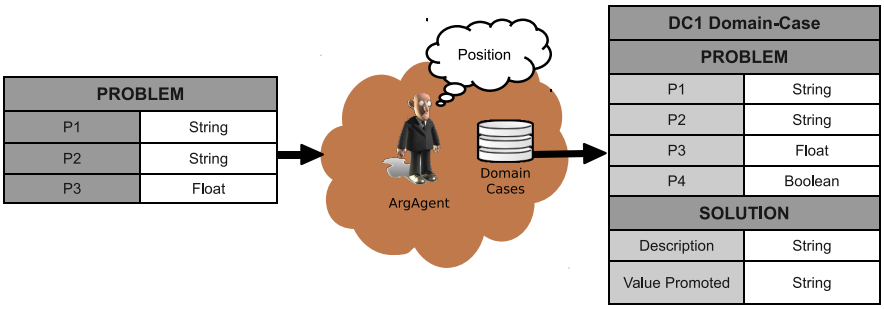

Figure 6.1: Example Structure of a Domain-Case

The structure of domain-cases that an argumentation system that implements our framework depends on the application domain. As example, Figure 6.1 shows the structure of a possible domain-case. Here, an argumentative agent must solve a problem characterised by three premises of different data types (P1, P2 and P3). In this example, the argumentative agent *ArgAgent* has found the domain-case *DC1* that matches de description of the problem to solve (has all or some of the premises of the problem with the same data values for this premises), also including an extra feature (P4). Note that here we assume that domain-cases also store the value promoted by the solution that they represent (see on the arguments structure below for a more detailed explanation). This is a design decision that can be replaced by any other assumption. Domain-cases are implemented in Magentix2 by using the `DomainCase.java` class of the `argAgents.knowledgeResources` package.

Arguments

Arguments in Magentix2 are implemented as java classes, concretely, in the `Argument.java` class of the Magentix2 `argAgents.knowledgeResources` package. In our proposal, arguments that agents interchange have the following structure:

```
1 public Argument(long id, Conclusion conclusion,
2         int timesUsedConclusion, String promotedValue,
3         SupportSet supportSet,
4         DependencyRelation proponentDepenRelation)
```

where `conclusion` is the conclusion of the argument, `timesUsedConclusion` stores the number of times that this conclusion has been used, `promotedValue` is the value that the agent wants to promote with this argument, `supportSet` is a set of elements that justify the argument and `proponentDepenRelation` is the dependency relation (power, authorisation or charity) established between the proponent of the argument and the opponent that the

argument is addressed. Thus, in our case-based argumentation framework arguments promote values. These values can be personal goods (e.g. efficiency, accuracy, etc.) or also social goods inherited from the agents' dependency relations. Preferences over values can determine the reasons that lead an agent to propose a specific argument or to accept or refuse an argument from another agent.

The support set is a knowledge resource of the Magentix2 `argAgents.knowledgeResources` package, represented by the `SupportSet.java` class, and can consist of different elements, depending on the argument purpose. On one hand, if the argument justifies a potential solution for a problem, the support set is the set of features (*premises*) that represent the context of the domain where the argument has been put forward (those premises that match the problem to solve and other extra premises that do not appear in the description of this problem but that have been also considered to draw the conclusion of the argument) and optionally, any knowledge resource used by the proponent to generate the argument (*domain-cases*, *argument-cases* and *argumentation schemes*). This type of argument is called a *support argument*. On the other hand, if the argument attacks the argument of an opponent, the support set can also include any of the allowed attack elements of our framework. These are: *distinguishing premises*, *counter-examples* or *critical questions*. This other type of argument is called an *attack argument*.

Definition 6.1.1 (Distinguishing Premise) *A distinguishing premise is a premise that does not appear in the description of the problem to solve and has different values for two cases or a premise that appears in the problem description and does not appear in one of the cases.*

Definition 6.1.2 (Counter-Example) *A counter-example for a case is a previous case (i.e. a domain-case or an argument-case that was deemed acceptable), where the problem description of the counter-example matches the current problem to solve and also subsumes the problem description of the case, but proposing a different solution.*

Definition 6.1.3 (Critical Question) *A* critical question *is a question associated to an argumentation scheme that represents a potential way in which the conclusion drawn from the scheme can be attacked. Therefore, if the opponent asks a critical question, the argument that supports this argumentation scheme remains temporally rebutted until the question is conveniently answered.*

Argument-Cases

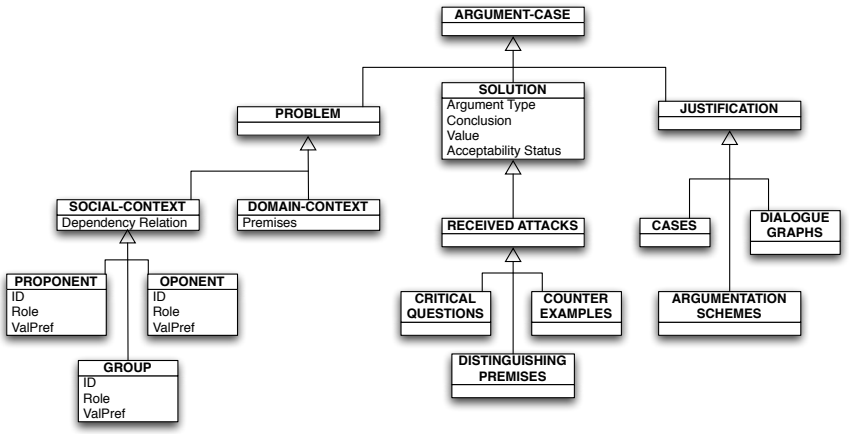

Figure 6.2: Structure of an Argument-Case

Argument-cases are the main structure that we use to computationally represent argumentation knowledge in Magentix2, concretely in the `ArgumentCase.java` class of the Magentix2 `argAgents.knowledgeResources` package. Their structure is generic and domain-independent. Figure 6.2 shows the generic structure of an argument-case.

Argument-cases have the three possible types of components that usual cases of CBR systems have: the description of the state of the world when the case was stored (*Problem*); the solution of the case (*Conclusion*); and the explanation of the process that gave rise to this conclusion (*Justification*).

The problem description has a *domain context* that consists of the *premises* that characterise the argument. In addition, if we want to store an argument and use it to generate a persuasive argument in the future, the features that characterise its *social context* must also be kept. The social context of the argument-case includes information about the *proponent* and the *opponent* of the argument and about their *group*. Moreover, we also store the preferences (*ValPref*) of each agent or group over the set of possible *values* that arguments can promote (pre-defined in the system). Finally, the *dependency relation* between the proponent's and the opponent's roles is also stored. In our framework, we consider three types of dependency relations: *power*, when an agent has to accept a request from another agent because of some pre-defined domination relationship between them; *authorisation*, when an agent has signed a contract with another agent to provide it with a service and hence, the contractor agent is able to impose its authority over the contracted agent and *charity*, when an agent is willing to answer a request from another agent without being obliged to do so.

In the solution part, the *conclusion* of the case, the *value* promoted, and the *acceptability status* of the argument at the end of the dialogue are stored. The acceptability status shows if the argument was deemed *acceptable*, *unacceptable* or *undecided* in view of the other arguments that were put forward in the agreement process. Therefore, an argument is deemed acceptable if it remains undefeated at the end of the argumentation dialogue, unacceptable if it was defeated during the dialogue and undecided if its acceptability status cannot be determined with the current information about the dialogue. In addition, the conclusion part includes information about the possible *attacks* that the argument received during the process. These attacks could represent the justification for an argument to be deemed unacceptable or else reinforce the persuasive power of an argument that, despite being attacked, was finally accepted.

Finally, the justification part of an argument-case stores the information about the knowledge resources that were used to generate the argument represented by the argument-case (the set of domain-cases and argument-cases). In addition, the justification of each argument-case has a *dialogue-graph* (or several) associated, which represents the sequence of arguments that form the dialogue where the argument was proposed. In this way, the complete conversation is stored as a directed graph that links argument-cases that represent the arguments of the dialogue. This graph can be used later to improve the efficiency in an argumentation dialogue in view of a similar dialogue that was held in the past.

6.1.2 Dialogue Strategies

In each step of an argumentation process, a Magentix2 argumentative agent can choose a specific locution to put forward and a content for it. The mechanism that agents follow to make such decisions is known as *dialogue strategy*. In our case-based argumentation framework, agents select the best locution to bring up depending on the *dialogue protocol* that they are following and the content of this locution depending on the knowledge that they have in their knowledge resources and the *tactic* that they follow to argue.

A Magentix2 argumentative agent will not initially accept any position from a peer. This agent will challenge positions of other peers when they are different from its position, even if they appear in its list of potential positions to propose. Also, it will try to generate an answer for any attack that it receives, but opposite to open-minded agents, argumentative agents do not accept the position of the peer that generated the attack if the last wins the debate. If an argumentative agent cannot generate positions, it will not participate in the dialogue. Finally, agents will accept arguments from other agents that have a power or authorisation dependency relation over them.

Depending on its dialogue protocol, the agent will choose the next locution to put forward on the dialogue game. Then, among the potential arguments that the agent may generate, it has to select one to propose. This implies to select the content of the locution to assert the argument. To make this selection, Magentix2 argumentative agents can use different tactics. From our point of view, a tactic consist on assigning more or less weight to the elements of an argument *support factor* used to select positions and arguments. The support factor estimates how suitable a current position or argument is in view of the suitability of similar arguments (to support or attack similar positions or arguments) put forward in previous argumentation dialogues, which are stored in the agent's case-base of argument-cases. In this way, the agent can select the most suitable position or argument to propose next. The support factor is computed by a linear combination of several parameters:

- **Persuasiveness Degree (PD):** is a value that represents the expected persuasive power of an argument by checking how persuasive an argument-case with the same problem description and conclusion was in the past.
- **Support Degree (SD):** is a value that provides an estimation of the probability that the conclusion of the current argument was acceptable at the end of the dialogue.
- **Risk Degree (RD):** is a value that estimates the risk for an argument to be attacked in view of the attacks received for an argument(s) with the same problem description and conclusion in the past.
- **Attack degree (AD):** is a value that provides an estimation of the number of attacks received by a similar argument(s) in the past.
- **Efficiency degree (ED):** is a value that provides an estimation of the number of steps that it took to reach an agreement posing a similar argument in the past.
- **Explanatory Power (EP):** is a value that represents the number of pieces of information that each argument covers. It is based on the number of knowledge resources were used to generate each similar argument-case retrieved.

The selection of these specific parameters to estimate the support factor of a position or argument has been determined by the nature of the elements of our argument-cases. Thus, the persuasiveness and support degrees take into account the acceptability status stored in the argument-cases, the attack and risk degrees look at the attacks received by the argument that an argument-case represents, the efficiency degree makes use of the dialogue graphs stored in

the argument-cases and the explanatory power computes the number of justification elements that argument-cases have. Therefore, the support factor is computed by using the following formula:

$$\begin{aligned} SF =& w_{PD} * PD + w_{SD} * SD + w_{RD} * (1 - RD) \\ &+ w_{AD} * (1 - AD) + w_{ED} * ED + w_{EP} * EP \end{aligned} \tag{6.1}$$

where $w_i \in [0, 1], \sum w_i = 1$ are weight values that allow the agent to give more or less importance to each parameter of the support factor. In Magentix2 argumentative agents, these weights can be set by specifying the corresponding parameters in the agent's constructor, as will be show in Section 6.3.

Thus, an agent can use the following dialogue tactics depending on the weight that it assigns to the elements of the support factor when it selects the best argument to bring up in each step of the argumentation dialogue:

- Persuasive Tactic: the agent selects such arguments which similar argument-cases were more persuasive in the past (assigns more weight to the persuasiveness degree).
- Maximise-Support Tactic: the agent selects such arguments that have higher probability of being accepted at the end of the dialogue (assigns more weight to the support degree).
- Minimise-Risk Strategy: the agent selects such arguments that have a lower probability of being attacked (assigns more weight to the risk degree).
- Minimise-Attack Tactic: the agent selects such arguments that have received a lower number of attacks in the past (assigns more weight to the attack degree).
- Maximise-Efficiency Tactic: the agent selects such arguments that lead to shorter argumentation dialogues (assigns more weight to the efficiency degree).
- Explanatory Tactic: the agent selects such arguments that cover a bigger number of cases or argumentation schemes. That is, such arguments that are similar to argument-cases that have more justification elements (assigns more weight to the explanatory power).

As pointed out before, the dialogue strategy that an agent follows determines the locution and content that it puts forward in each step of the dialogue. Thus, different strategies can be more or less suitable depending on the other agents that participate in the dialogue, the tactics that they follow and their available knowledge resources.

6.2 Argumentation API

In this section, we define the different modules that implement the argumentation API of the Magentix2 platform and their functionality. For a tutorial on how to use these modules for programming argumentative agents see Section 6.3.

- **Domain CBR module**: consists of a CBR module with domain-dependent data (previous problem-solving experiences stored in the form of domain-cases). This CBR has to be initialised with data of the application domain.
- **Argumentation CBR module**: consists of a CBR module with argumentation data (previous arguments stored in the form of argument-cases). Once an agent has a list of potential solutions for a current problem, it has to select the best position to put forward among them. Also, the agent can generate arguments to support its position and attack another agent's arguments and positions. Then, this module is used to look for previous argumentation experiences and use this knowledge to select the best positions and arguments to propose.
- **Argumentative agent**: is an agent with a Domain CBR and an Argumentation CBR able to engage in an argumentation dialogue to solve a problem. This agent learns about the domain problem and the argumentation dialogue adding and updating cases into the domain and argumentation case-bases in each CBR run.
- **Commitment Store**: is a resource of the argumentation framework that stores all the information about the agents participating in the problem solving process, argumentation dialogues between them, positions and arguments. By making queries to this resource, every agent of the framework can read the information of the dialogues that it is involved in. To facilitate the communication among argumentative agents and the commitment store, this resource has been implemented as a Magentix2 `CAgent`.

6.2.1 Argumentative Agents: ArgCAgent Class

With the class `ArgCAgent.java` Magentix2 distribution includes an implementation of an argumentative agent. This agent is an extension of the Magentix2 conversational agent (`CAgent`, see Section 4). Argumentative agents have two CBR modules: Domain CBR and Argumentation CBR. The main functionalities of the argumentative agents are the generation, selection and evaluation of positions and arguments. These are specified as methods, to facilitate a better

understanding of the code and to facilitate modifications and updates. Table 6.4 provides an overview of the main methods used to manage positions and arguments in the `ArgCAgent.java` class. Next, we explain how argumentative agents can manage positions and arguments.

Method	**Description**
addPosition	Returns an `ACLMessage` with the locution ADDPOSITION and a `Position` to send to the Commitment Store
createMessage	Creates and returns an `ACLMessage` with the message arguments. Messages are managed by the main `execution` method of `cAgents` and are sent and received in the corresponding *send* and *receive* states (see Section 4)
generateAttackArgument	Returns an attack `Argument` against the given argument of the given agent identifier
generateCEAttack	Returns a counter-example attack `Argument` against the agent of the given agent identifier, and its given premises
generateDPAttack	Returns a distinguishing premises attack `Argument` against the agent of the given agent identifier, and its given premises
generatePositions	Returns an `ArrayList` of `Position` with all generated positions to solve the specified problem, ordered from more to less suitability degree to the problem
generateSupportArguments	Returns an `ArrayList` of support `Argument` for the given `Position` against the given agent identifier
getDifferentPositions	Returns an `ArrayList` of positions that are different from the defended position and also are not asked yet
getDistinguishingPremises	Returns an `ArrayList` with distinguishing premises between the HashMaps given as arguments
getUsefulPremises	Returns a `HashMap` of the useful premises of the agent of the current problem to solve (the premises of the `Position` that are specified in the problem characterisation).
updateCBs	Adds the final solution to the current problem and adds it in the domain-cases case-base. Also, stores all the generated argumentation data in the argument-cases case-base. Finally, makes a cache of the domain CBR and the argumentation CBR

Table 6.1: `ArgCAgent.java` methods to manage positions and arguments

POSITION MANAGEMENT

A position is a solution that defends an agent as the correct one to apply to solve the problem at hand. The position generation is made in two steps. First, the agent retrieves from its Domain CBR the most similar domain-cases to the current problem to solve (by using the `retrieve` method of the `DomainCBR.java`) class. With them, the agent is able to propose its position

in view of the solutions applied to similar problems in the past. Then, the agent evaluates the suitability of each position by using its Argumentation CBR to compute the support factor parameters (by using the method `getDegrees` of the `ArgCBR.java` class). Then, each position is assigned a *suitability degree* by using the formula:

$$Suitability = w_{SimD} * SimD + w_{SF} * SF \tag{6.2}$$

where $w_i \in [0,1], \sum w_i = 1$ are weight values that allow the agent to give more or less importance to the similarity degree with the domain-cases used to generate its position or the support factor (these weights can be set in the agent's constructor). Agents sort their potential positions from most to less suitable depending on their value preference order, and for each group of positions that promote the same value, agents sort them by their suitability degree. The most suitable position is selected as the one that the agent is going to propose and defend first.

The argumentation API of Magentix2 includes several *similarity algorithms* to compute the similarity degree (*SimD*) in the `SimilarityAlgorithms.java` class. Also, the `Metrics.java` class includes several metrics to compute distances between cases, which are used in the similarity algorithms. Table 6.2 shows these methods, which belong to the `argAgents` package.

The specific algorithm that an argumentative agents uses to compute the similarity degree between two cases can be set in the `configuration.xml` file of the `configuration` package, shown below.

```
<root>
...
<domaincbr>
        <similarity>normalizedEuclidean</similarity> <!--
           normalizedEuclidean or weightedEuclidean or
           normalizedTversky -->
</domaincbr>
...
</root>
```

SUPPORT ARGUMENTS MANAGEMENT

A list of possible support arguments is generated with the method `generateSupportArguments` by using different combinations of the available support elements in the support set. This list

Class	Method	Description
Metrics.java	doDist	This method decides about which is the data type of its attributes, and return the distance between them
Metrics.java	dist	This method calculates the distance between a pair of attributes
SililarityAlgorithms.java	normalizedEuclideanSimilarity	Returns a list of the candidate domain-cases with a similarity degree to the given domain-cases. The similarity is calculated using normalized Euclidean distance among the premises
SililarityAlgorithms.java	weightedEuclideanSimilarity	Returns a list of the candidate domain-cases with a similarity degree to the given domain-cases. The similarity is calculated using weighted Euclidean distance among the premises
SililarityAlgorithms.java	normalizedTverskySimilarity	Returns a list of the candidate domain-cases with a similarity degree to the given domain-cases. The similarity is calculated using normalized Tversky distance among the premises

Table 6.2: Methods to compute the distance and similarity between cases

is ordered by using the *suitability degree* explained before and the argument that has the higher degree is proposed first as support argument to justify the agent's position.

ATTACK ARGUMENTS MANAGEMENT

To generate an attack argument, the premises that the argument to attack has and the social context between the agents that are arguing are taken into account. With this information, argumentative agents extract the argument-cases that match with the current position and have a similar social context. Then, if the dependency relation allows to attack the other agent, the agent generates the attack argument. The first type of attack that the agent will try to generate is a counter-example attack (with the method `generateCEAttack`), and if it is

not possible the agent will try to generate a distinguishing premise attack (with the method `generateCEAttack`). The counter-example attack consists on an argument that includes a domain-case or an argument-case whose conclusion contradicts the conclusion of the attacked argument. A distinguishing premise attack consists on an argument that includes a premise (or a set) that describes the problem and that the attacked agent did not consider to generate its position (and its associated support argument) or a premise that both agents have, but with different data value.

6.2.2 Argumentation Protocol

Argumentative agents need a protocol to exchange positions and arguments and engage in the argumentation dialogue. The protocol is represented by a set of locutions that the agents use to communicate with each other, and a state machine that defines the behaviour of an agent in the argumentation dialogue. The state machine has been implemented in the `ArgCAgent.java` class by overwriting the states of the argumentation protocol of the API. This protocol has been implemented in the `Argumentation_Participant.java` abstract class of the `cAgents.protocols` package. In each state of the protocol, the different locutions that can be received and generated are taken into account to act in consequence and move to the next state. Inside each state, the corresponding actions are performed using the necessary calls to the different functions of the agent (shown in Table 6.9 of Section 6.3.2). Also, argumentative agents can make queries to the commitment store and retrieve information about their argumentation dialogue. Recall that since argumentative agents are a special type of Magentix2 `cAgents`, agents queue messages and send or receive them in the corresponding *send* and *receive* states (see Section 4). The behaviour of the commitment store resource has been implemented by overwriting the abstract class `CommitmentStore_Protocol.java` of the `cAgents.protocols` package in the `CommitmentStore.java` class.

The set of allowed locutions of our argumentation protocol are codified as constants in the `Argumentation_Participant` protocol of the `cAgents.protocols` package. These locutions are the following:

- OPENDIALOGUE: with this locution an agent opens the argumentation dialogue, asking other agents to collaborate or negotiate to solve a problem that it has been presented with.
- ENTERDIALOGUE: with this locution an agent engages in the argumentation dialogue to solve the problem.

- WITHDRAWDIALOGUE: with this locution an agent leaves the argumentation dialogue.
- ADDPOSITION: with this locution an agent puts forward its position as its proposed solution to solve the problem under discussion in the argumentation dialogue.
- WHY: with this locution an agent challenges the position or the argument of another agent, asking it for a support argument.
- NOCOMMIT: with this locution an agent withdraws its position as a solution for the problem under discussion in the argumentation dialogue.
- ASSERT: with this locution an agent sends to another agent an argument that supports its position.
- ACCEPT: with this locution an agent accepts the argument or the position of another agent.
- ATTACK: with this locution an agent challenges the argument of another agent.

Also, there are other allowed locutions to manage the life cycle of argumentative agents and get information from the commitment store:

- FINISHDIALOGUE is a locution to inform an agent that it must perform the necessary actions (if any) before withdrawing from the dialogue.
- DIE is a locution to inform an agent that it must *shutdown* its execution.
- GETALLPOSITIONS is a locution to request the commitment store the list of available positions at a certain step of the dialogue. The commitment store uses the same locution to answer this request.

Figure 6.3 shows the state machine that defines the behaviour of an agent that follows the Magentix2 `Argumentation_Participant` protocol. In the figure, dotted states represent *wait states* where the argumentative agent waits for messages from other agents or the commitment store. Also, dotted lines represent transitions between states when these incoming messages, with their associated locution, are received. Therefore, the transitions between states depend on the locutions that the agent can use in each step of the dialogue. The states of the argumentation dialogue are described as follows:

1. **Begin:** this is the start state of the argumentation protocol.

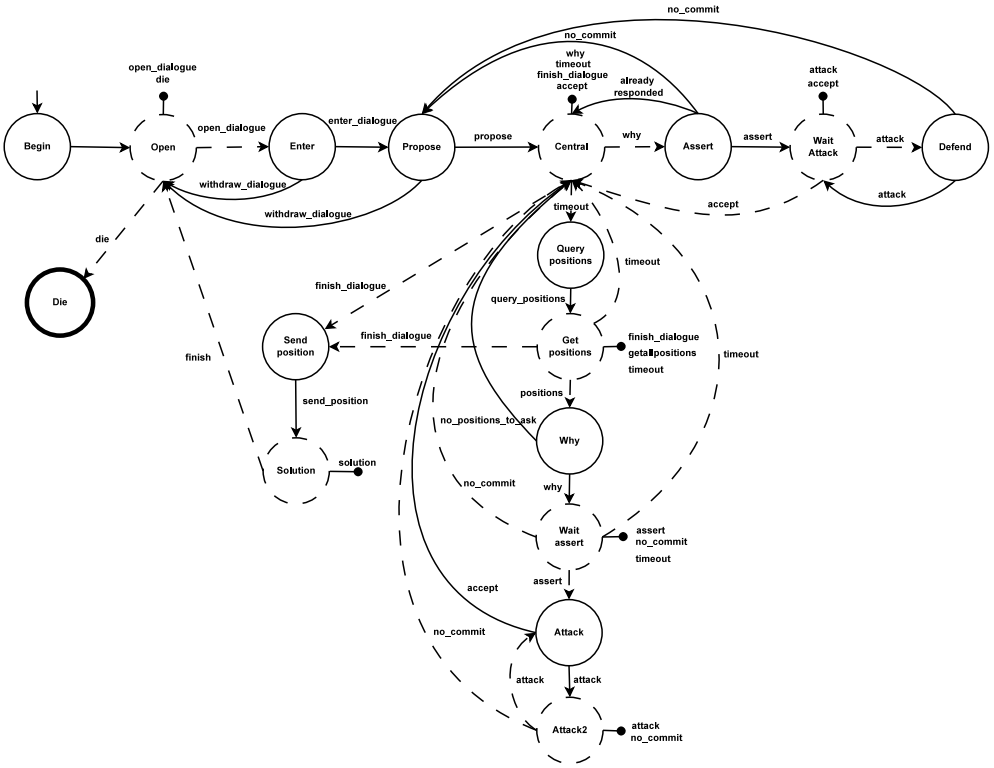

Figure 6.3: Argumentation Protocol

2. **Open:** when the agent is initialised it remains in this state waiting for an *OPENDIALOGUE* locution. The agent will move back to this state when the dialogue has finished. The *OPENDIALOGUE* locution inform the agent that a new dialogue to solve a problem has started. Also, when an agent received the *DIE* locution in this state, it must shutdown its execution.

3. **Enter:** in this state, the agent will retrieve such cases of its domain-cases case-base which features match the given problem with a similarity degree greater than a given threshold. If the agent has been able to retrieve similar domain-cases and use their solutions to propose a solution for the current problem the agent will engage in the dialogue with the locution *ENTERDIALOGUE* and will go to the state ”Propose”. The agent only engages in the dialogue if it has solutions to propose. Otherwise, the agent can refuse to engage in the dialogue with the locution *WITHDRAWDIALOGUE*.

4. **Propose:** when the agent is in this state it has retrieved a list of similar domain-cases to the current problem to propose a solution (position to defend). If there are several solutions to propose, it will select the most suitable and go to the state ”Central”. Otherwise,

the agent will leave the dialogue with the locution *WITHDRAWDIALOGUE*.

5. **Central:** this is a central state, since the agent can try to attack other positions or defend its position from the attacks of other agents. First, the agent checks if there is any *WHY* request from other agent. This locution is used to ask an argumentative agent to justify its position. In that case, the agent will go to the state "Assert" to try to generate a support argument for its position. If the agent has not received any *WHY* request before a specified *timeout*, it will go to the state "Query Positions" to challenge the positions of other agents. Also, an agent can be reported by other agent that the latter *ACCEPT*s its position. Alternatively, the agent can receive a *FINISHDIALOGUE* locution in this state, and it will go to the state "Send Position" to start the actions to leave the dialogue when it has proposed yet a position.

6. **Assert:** the agent that received the *WHY* request will *ASSERT* a support argument to the opponent if it can. This implies going to the state "Wait Attack" and wait for incoming attack arguments. If the agent is not able to provide a support argument to defend its position, it must move back to the state "Propose" with the locution *NOCOMMIT* to withdraw its position from the dialogue and if possible, propose another generated position. Also, argumentative agents do not respond to the same *WHY* query from the same opponent agent twice. In this case, the argumentative agent will move back to the state "Central" and ignore the repeated *WHY* request.

7. **Wait Attack:** in this state, the agent that has put forward a support argument for its position waits for an *ATTACK* or an *ACCEPT* locution. In the case that an *ATTACK* is received, the agent will go to the state "Defend" to try to rebut the attack. If the agent receives an *ACCEPT*, it means that the opponent agent has accepted its position and the proponent agent will move back to state "Central".

8. **Defend:** in this state, an agent that has received an *ATTACK* from an opponent agent tries to reply with another *ATTACK*. In this case, the proponent agent will move back to the state "Wait Attack" to wait for the response of the opponent agent. However, if the proponent agent is not able to counter-attack, it must move back to the state "Propose" with the locution *NOCOMMIT* to withdraw its position from the dialogue and if possible, propose another generated position.

9. **Query Positions:** in this state the agent that has decided to challenge the positions of other agents requests the commitment store the list of current positions proposed in the dialogue with the locution *GETALLPOSITIONS*.

10. **Get Positions:** the agent that has requested the commitment store for active positions in the dialogue moves to this state to wait for an answer. Here, the agent can receive the list of positions with the same locution *GETALLPOSITIONS*. Alternatively, the agent can receive a *FINISHDIALOGUE* locution in this state, and it will go to the state "Send Position" to start the actions to leave the dialogue when it has proposed yet a position. Also, if the agent does not receive any response before a specified *timeout*, it will move back again to the state "Central".

11. **Why:** in this state, the agent that has received a list of potential positions to challenge makes a random choice to challenge one of them with the locution *WHY* (from these positions that are different to its own). Otherwise, if there are no positions to challenge and ask for a justification, the agent moves back to the state "Central".

12. **Wait Assert:** in this state, the agent that has challenged a position waits for a response from the challenged agent. If it receives such a response with the locution *ASSERT*, it tries to rebut the position of the challenged agent and moves to the state "Attack". Otherwise, if the challenged agent cannot provide a justification for its position, it must withdraw such position from the dialogue with the locution *NOCOMMIT* and the challenger moves back to the state "Central". Also, if the agent does not receive any response before a specified *timeout*, it will move again to the state "Central".

13. **Attack:** in this state, if an agent has received a justification for a challenged position, it tries to generate an attack with the locution *ATTACK*. In this case, the agent moves to the state "Attack2". However, if it is not able to attack the position, it will accept it with the locution *ACCEPT* and move back to the state "Central".

14. **Attack2:** once an agent has generated an attack for the position of a proponent agent, it waits in this state for the answer of the proponent. Thus, if the proponent is able to counter-attack and sends a locution *ATTACK*, the agent will move back to the state "Attack" to try to generate a new attack to rebut the position of the proponent. Otherwise, the proponent must withdraw its position from the dialogue with the locution *NOCOMMIT* and the attacker agent moves back to the state "Central".

15. **Send Position:** an agent reaches this state when it has received a locution *FINISHDIALOGUE* to start the actions to leave the dialogue when it has proposed yet a position. Here, the agent sends its current position to update the commitment store information and avoid possible inconsistencies. After that, the agent moves to the state "Solution" to wait for the final solution applied to solve the problem.

16. **Solution:** when the dialogue has finished, the final solution to apply is reporter to all dialogue participants. When agents receive this information, they update their case-bases with the data learnt from the dialogue and move back to the state "Open".

17. **Die:** this is the final state of the argumentation protocol. Agents move to this state when they receive the locution *DIE* and shutdown their execution.

6.3 Programming Argumentative Agents

6.3.1 How to run Magentix2 Argumentative Agents

Magentix2 provides a default implementation of an argumentative agent that can be run from the *Magentix2/examples/bin* directory. Thus, the user can execute the `Start-Argumentation Example.sh` script that launches the default argumentation example:

```
cd Magentix2/examples/bin

sh Start-ArgumentationExample.sh
```

If the user wants to use a different configuration of the parameters of the example, it is necessary to modify the source code of the example (see Table 6.3). To do it, the user has to use the source code from the *Magentix2/src* directory and create a project to modify the code. The Magentix2 library must be included in the project. The steps to perform this task are explained in Section 2.2. In this case, the code to modify is located in the *Argumentation_Example* package. The main class to execute tests is *TestArgCAgent.java*. In this class, the user can modify the initial data given to the agents, the number of agents to execute, the number of problems to solve and the weights of the argumentative agents to perform different argumentation strategies. The behaviour of the argumentative agents can be changed modifying the default actions of the methods in the *ArgCAgent.java* class (see Table 6.4). Furthermore, in the argumentation example provided, we assume the existence of a *tester* agent that is in charge of requesting a group of *ArgCAgents* to solve a problem. This agent, implemented in the Magentix2 argumentation API with the class ŤesterArgCAgent extends a Magentix2 `SingleAgent`. Also, it is in charge of gathering information at the end of the protocol and convey the final solution proposed.

A standard argumentative agent can be created and executed by using these commands:

Package	Classes	Description
Argumentation_Example	AgentsCreation.java	This class has different methods to create groups of agents and some of their parameters
	ArgCAgent.java	This class implements the argumentative agent as a CAgent. It can engage in an argumentation dialogue to solve a problem.
	CreatePartitions.java	This class creates different partitions of domain-cases and argument-cases to make tests
	TestArgCAgent.java	This class launches a test with argumentative agents, including the commitment store and a tester agent that acts as initiator of the dialogue
	TesterArgCAgent.java	This class represents a tester agent in charge of sending the test domain-case to solve to a group of agents and acts as the initiator of the dialogue
TestArgumentation (datafiles)	partitionsInc	Folder with the data files to perform tests

Table 6.3: Argumentation example packages and classes

Method	Parameters	Description
generateAttackArgument	incomingArgument, opponentID	Defends and attacks other agents' positions
generatePositions	problem	Evaluates if it can propose a position
generateSupportArguments	myPosition, opponentID	Defends its position
updateCBs	solution	Stores generated data in the dialogue

Table 6.4: Main *ArgCAgent.java* methods

```
ArgCAgent agent =
new ArgCAgent(new AgentID("qpid://agentName@localhost:8080"),
socialEntity, friendsList, depenRelsList, group,
"commitmentStoreID", iniDomainCasesFilePath,
finDomainCasesFilePath, domCBRindex, domCBRthreshold,
iniArgCasesFilePath, finArgCasesFilePath,
```

```
7  wPD, wSD, wRD, wAD, wED, wEP);
8
9  agent.start();
```

where *socialEntity* represents the social entity (name, role, preference values...) created for this agent, *friendsList* is an `ArrayList` with the social entities that represent the friends of the agent (the agents that it knows), *depenRelsList* is an `ArrayList` that represents the dependency relations that the agent has with the rest of agents of the system, *group* represents the group that the agent belongs and *commitmentStoreID* represents the identifier of the commitment store resource. There are also two `ArrayList` called *iniDomainCasesFilePath* and *finDomainCasesFilePath* with the file paths that store the initial and final domain-cases case-bases. In addition, *domCBRindex* establishes an index (in the sense of a `HashTable` index) to retrieve cases from the domain-cases case-base and *domCBRthreshold* establishes a threshold over which two cases are considered as similar. The *iniArgCasesFilePath* and *finArgCasesFilePath* `ArrayLists` represent the file paths that store the initial and final argument-cases case-bases. Finally, the *wPD*, *wSD*, *wRD*, *wAD*, *wED* and *wEP* are the weights assigned to the parameters to compute the *supportFactor*. Different combinations of these weight allow argumentative agents to follow different argumentation strategies.

6.3.2 How to create your own Magentix2 Argumentative Agent

Magentix2 provides the core of the argumentation protocol, the Domain CBR, the Argumentation CBR and the knowledge resources (see Table 6.5) needed to crete your own argumentative agent (and not directly using the default argumentative agent provided in the *Argumentation_Example*).

The main tasks to perform to create an argumentative agent are:

- To use the Domain CBR to store and retrieve domain knowledge.
- To use the Argumentation CBR to store and retrieve argumentation knowledge.
- To implement the argumentation protocol methods and thus, specify how to perform the actions of the argumentative agent in the states of the designed argumentation protocol.

The Domain CBR can be used to generate and select positions to defend in the argumentation dialogue. To make a query to the domain CBR, the user has to provide a problem (codified as

Package	Classes	Description
argAgents	argCBR.ArgCBR.java	This CBR stores argument-cases that represent past argumentation experiences and their final outcome
	CommitmentStore.java	Agent that stores all the information about the argumentation dialogues
	Configuration.java	Configuration parameters
	domCBR.DomainCBR.java	This CBR stores domain knowledge of previously solved problems
	knowledgeResources	Package that contains the classes needed to manage the data of the CBRs and arguments
	Metrics.java	Used by the CBRs to measure the similarity between case attributes, based on their distance
	SimilarityAlgorithms.java	Contains algorithms to calculate similarity measures between cases
cAgents.protocols	Argumentation_Participant.java	Abstract class that defines the argumentation participant protocol to be followed by the CAgents
	CommitmentStore_Protocol.java	Abstract class that defines the protocol that the Commitment Store follows to attend the petitions of the agents

Table 6.5: Argumentative agents core packages and classes

a list of premises or a domain-case without solution) and a threshold of similarity. The domain CBR module searches the domain case-base and returns a list of similar domain-cases to the given problem. In addition, with every request attended and every CBR cycle performed, the module adds, modifies or deletes one or more domain-cases of the domain case-base. In the current version of the API, if the problem that has been solved is similar enough (over certain similarity threshold) to a case of the domain-cases case-base, the update algorithm updates this case with the new data acquired. Otherwise, a new domain-case is created and added to the case-base. The main methods of the Domain CBR are shown in Table 6.6 (for a detailed explanation of all Domain CBR methods, see the `DomainCBR.java` associated JavaDoc).

The Argumentation CBR can be used to look for previous argumentation experiences and use

Method	Parameters	Description
addCase	domain-case	Adds a new domain-case to domain case-base. Otherwise, if the same domain-case exists in the case-base, adds the relevant data to the existing domain-case
getPremisesSimilarity	premises1, premises2	Obtains the similarity between two HashMap of premises using the similarity algorithm specified in the configuration of this class
retrieve	premises, threshold	Retrieves the most similar domain-cases to the given premises with a similarity degree greater or equal than a given threshold

Table 6.6: Main *DomainCBR.java* methods

this knowledge to select the best positions and arguments to propose. Thus, argument-cases store information related to the domain and the social context where previous arguments (and their associated positions) were used. The information about the domain consists of a set of features (premises) to compare cases and information about the social context where the proposed solution was applied (e.g. the agents that participated in the dialogue to solve the problem, their roles or their value preferences). The latter information can determine if certain positions and arguments are more persuasive than others for a particular social context and hence, agents can select the best ones to propose in the current situation. As for the domain-cases case-base, if the argument-cases created during the problem solving process are similar enough to previous argument-cases stored in the argument-cases case-base, the update algorithm updates those cases with the new data acquired. Otherwise, new argument-cases are created and added to the case-base. The main methods of the Argumentation CBR are shown in Table 6.7 (for a detailed explanation of all Argumentation CBR methods see the `ArgCBR.java` associated JavaDoc).

The contents of the case-bases of the Domain CBR and the Argumentation CBR can be stored as java serialised objects. In this way, we provide a quick and simple data persistence mechanism. Thus, both `DomainCBR.java` and `ArgCBR.java` classes include different methods to load and save information about cases from/to data files, see Table 6.8.

The main functionalities of the argumentative agents (generation, selection and evaluation of positions and arguments) are specified as methods, to facilitate a better understanding of the code and to facilitate modifications and updates. The main method of the argumentative agents is the `execution` method. This method overwrites the `execution` method of

Method	Parameters	Description
addCase	argument-case	Adds a new argument-case to the case-base. Two cases are considered equal if they have the same domain context, social context, conclusion and status of acceptability
getDegrees	argumentProblem, solution, allPositions, index	Return a list with the degrees (attack, efficiency, explanatory power, persuasiveness, support and risk) of an argument-case
getSameDomainAndSocialContextAccepted	premises, solution, socialContext	Returns the argument-cases with the same domain and social context that have been accepted in the past

Table 6.7: Main *ArgCBR.java* methods

Method	Description
doCache	Stores the current domain-cases case-base in a specified file path
doCacheInc	Stores the current domain-cases case-base in a specified file path, but keeping the contents of that file
loadCaseBase	Loads a case-base stored in a specified file path

Table 6.8: Main CBR persistence methods

the Magentix2 `CAgent`, implementing the argumentative functions of the agent. Inside the `execution` method, the `myArgumentation` class extends the argumentation protocol of Magentix2. Concretely, the argumentation API of Magentix2 includes two special protocols that allow agents to argue: the `Argumentation_Participant` protocol and the `CommitmentStore_Protocol`. Both are abstract classes of the `cAgents.protocols` package that argumentative agents and the commitment store extend respectively. These classes provide a template for the behaviour of the argumentative agents and the commitment store. Each one implements a new `CFactory` (see `CAgent` JavaDoc for more information) with the graph that specifies the sequence of states and transitions among them that agents can pass through during the argumentation dialogue.

Therefore, to create a new argumentative agent, it is necessary to implement the abstract methods of the `Argumentation_Participant` protocol. The decisions and the actions to perform in each state of the argumentation protocol are codified in these methods. The main methods to implement are shown in Table 6.9. Due to the complexity of defining a new argumentative agent, the user should consider to use the *ArgCAgent.java* class of the example as a base to implement the behaviour of a new argumentative agent.

Method	Description
doAccept	actions to perform and send an *ACLMessage* accepting the other agent's position or argument
doAssert	try to assert a support argument to respond to the *WHY* received previously
doAttack	actions to perform to generate an attack argument against an attack or assert received
doDie	actions to perform when the message with locution *DIE* is received
doEnterDialogue	evaluates if the agent can enter in the dialogue offering a solution
doFinishDialogue	actions to be executed when the dialogue has to finish
doGetPositions	get the positions of the agents in the dialogue sent by the Commitment Store as an object
doMyPositionAccepted	actions to perform when the position of the agent has been accepted
doNoCommit	creates an *ACLMessage* to send with the locution *NOCOMMIT*
doOpenDialogue	takes the domain-case to solve and the dialogue ID from the received *ACLMessage* given
doPropose	proposes a position to defend in the dialogue. If it can not, it does a *WITHDRAWDIALOGUE*
doQueryPositions	creates a message to send to the Commitment Store with locution *GETALLPOSITIONS* to obtain all the positions of the dialogue
doSendPosition	sends an *ACLMessage* with the position defended by the agent
doSolution	actions to perform when the final solution to the current problem to solve arrives in an *ACLMessage*
doWhy	choose a position to send a *WHY* message if it can

Table 6.9: Methods to implement in the Argumentation Protocol

6.3.3 Example: Call Centre Application

To date, the system is implemented in the domain of a customer support application where several operators represented by software agents argue to provide the best solution for an incidence (also known as *ticket*) received. Thus, we use this example domain to show how `ArgCAgents` can be executed. Int the current version of the software, the example is implemented in the `Argumentation_Example` package, which includes the java files for creating agents (`AgentsCreation.java`), the template code of the argumentative agent (`ArgCAgent.java`), a file to create case-bases with data about example domain-cases and argument-cases (`CreatePartitions.java`), a *test* file for executing the example (`TestArgCAgent.java`) and a file with a template of the *tester* agent (`TesterArgCAgent.java`). In this section we

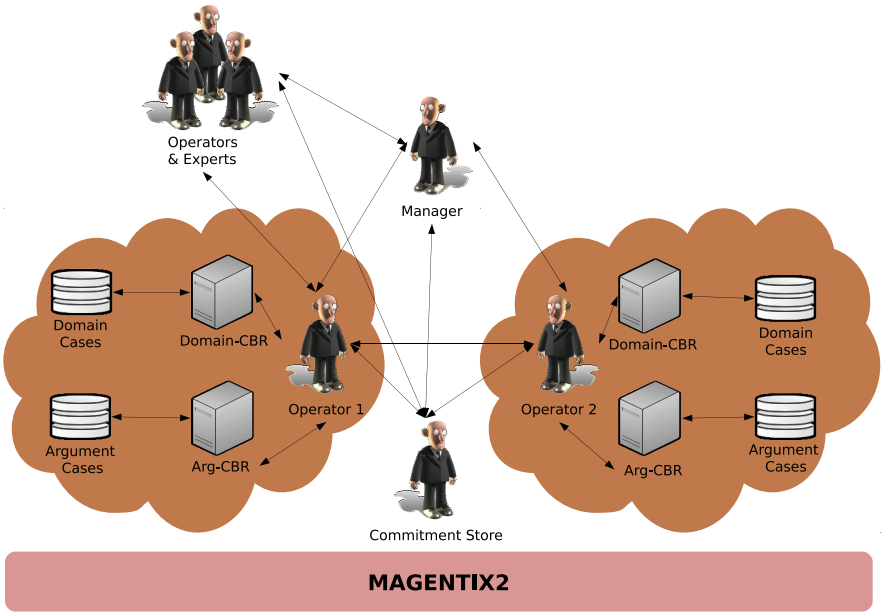

Figure 6.4: Data-flow for the argumentation process of the helpdesk application

provide a brief explanation of this example. Details of the implementation can be found in the JavaDoc of the files of the package.

In order to show how the developed system works, the data-flow for the problem-solving process (or argumentation process) to solve each problem is shown in Figure 6.4 and described below (arrows in the figure are labelled with the number of the data-flow step that they represent):

1. First, we have some argumentation agents running in the platform and representing the technicians of the call centre. The *tester* agent sends the problem to solve (also known as *ticket* in the common call centre terminology) to the group of agents.

2. Each agent evaluates individually if it can engage in the dialogue offering a solution. To do that, the agent makes a query to its domain CBR to obtain potential solutions to the ticket based on previous solutions applied to similar tickets. To compute such similarity, agents use a *weighted Euclidean* algorithm that searches their domain-cases case-bases for previous problems that semantically match the category of the current ticket to solve. Thus, the algorithm retrieves all problems of the same category and of related categories and select those that syntactically match (assign the same values to the attributes that match the ticket attributes) and overpass a defined *similarity threshold*. If one or more valid solutions can be generated from the selected domain-cases, the agent will be able

to defend a position in the dialogue. We consider a valid solution any domain case from the domain CBR with one or more solutions and with a similarity degree greater than the given threshold. Moreover, the agent makes a query to its argumentation CBR for each possible position to defend. With these queries the *suitability degree* of the positions is obtained. This degree represents if a position will be easy to defend based on past similar argumentation experiences. Then, all possible positions to defend are ordered from less to more suitability degree.

3. When the agents have a position to defend (a proposed solution), these positions are stored by the commitment store, such that other agents can check the positions of all dialogue participants. Every agent tries to attack the positions that are different from its position.

4. The argumentation process consists on a series of steps by which agents try to defend its positions by generating counter-examples and distinguishing premises for the positions and arguments of other agents. A counter-example for a case is generated by retrieving from the domain case-base another case that matches the features of the former, but has a different conclusion. Similarly, distinguishing premises are computed by selecting such premises that the agent has taken into account to generate its positions, but that other agents did not considered. If different attacks can be generated, agents select the best attack to rebut the position of another agent by making a query to their argument-cases case-base, extending the characterisation of each case with the current social context. In this way, agents can gain knowledge about how each potential attack worked to rebut the position of an agent in a past argumentation experience with a similar social context. When an agent is able to rebut an attack, the opponent agent makes a vote for its position. Otherwise, the agent must withdraw its position and propose an alternative position, if possible.

5. The dialogue finishes when no new positions or arguments are generated after a specific time. The *tester* agent is in charge of making queries to the commitment store agent to determine if the dialogue must finish. Then, this agent retrieves the active positions of the participating agents. If all agents agree, the solution associated to the agreed position is selected. Otherwise, the most frequent position wins. In case of draw, the most voted position in selected. If even in this case the draw persists, a random choice is made. Finally, the *tester* agent communicates the final solution (the outcome of the agreement process) to the participating agents.

CHAPTER

7

Tracing Service

This chapter 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.

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.

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.

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 7.1. 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 7.1: `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 7.6.1 (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.

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 7.6.3, 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 7.2.

Table 7.2: 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 7.3.1), subscription/unsubscription to/from tracing services (Section 7.3.2) and listing of tracing entities or services (Section 7.3.3).

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 7.3.

Table 7.3: 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");
```

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 7.4.

An example of tracing service subscription can be found in Section 7.6.1, 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.

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 7.4: 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 7.5.

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 7.5: 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.

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 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

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 7.6.

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 7.6.1, 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 7.7.

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 7.6.1, 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 7.8.

Table 7.6: 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).

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 7.7: 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 7.8: 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 7.9

Table 7.9: 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`

7.5 Customizable Trace Support

In the trace system of Magentix2, the trace events are customizable. Platform administrators can define the appropiate trace level of the platform. Thus, platform applications only use the necessary trace support to avoid the application overload. Some platform applications may not be interested in any events and in fact, trace support is not needed. However, other domains might need a full trace support.

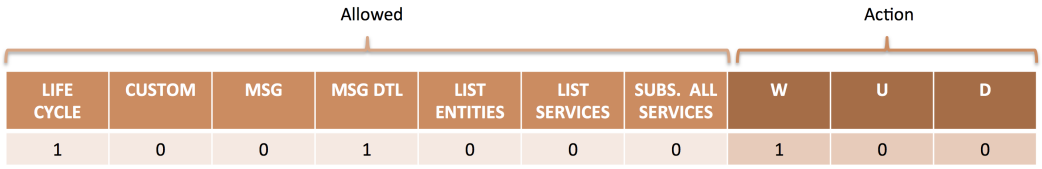

Figure 7.1: Trace Support Mask

In the trace support, it is defined a mask which allows administrators to determinate the types of events supported at the platform, by means of a bit position for each type of event (Figure 7.1). The meaning of each bit position is are detailed in Tables 7.10 and 7.11.

This definition should be incorporated to the settings xml of the platform. Thus, it could be possible to run the platform without trace support, even without an agent TM. Notice that the events related with the agents life cycle will be always generated, but the mask informs regarding if it is possible for agents to subscribe to these life cycle events or not.

Moreover, it would be possible to start the platform without trace support and then run it, launching the agent TM and informing the existing agents about the new event mask.

This mask is going to be used both by the TM (which read it from the settings) and the agents

Table 7.10: Event's types allowed to customize

LIFE_CYCLE	Subscriptions to tracing services related to agents life cycle (Agent creation and destruction).
CUSTOM	Publication and subscription to tracing services defined by users.
MSG	Subscription to the sending and receiving of ACL messages.
MSG_DTL	Subscription to the sending and receiving of ACL messages with the content included.
LIST_ENTITIES	List all the tracing entities.
LIST_SERVICES	List all the tracing services.
SUBSCRIBE_TO ALL_SERVICES	Subscription to all available tracing services.

Table 7.11: Action information about the mask reception

WELCOME(W)	The mask is sent because the TM has just launched. Another possibility is to receive it as a response of a newAgent event.
UPDATE(U)	The mask is sent because the tracing policy has changed, and the TM informs all the agents with a new mask event.
DIE(D)	The TM informs agents about it is going to shut down. So the trace support will be not available.

registered at the platform. The TM will use it to check if it is allowed to register new services or subscriptions. The agent will check the current mask in order to generate or not events[1]. By default, agents will have a mask with no events allowed. When the agent TM is created, it is activated the trace support and the agents receive the trace support mask with the allowed events.

7.6 Example: TraceDaddy

This simple example shows how to use domain independent tracing services to follow other agents' activities and to make decisions according to that 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. As the agents use only the trace events related with the agents life cycle and message detail, the optimal trace level is defined with the mask value 1001000100 (see Figure 7.1).

[1] except life cycle, that will be always generated

- **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.

7.6.1 `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      }
```

7.6.2 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   }
```

7.6.3 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            tm.setTraceMask(new TraceMask("1001000100"));
31
32            System.out.println("INITIALIZING...");
33
34            /**
35             * Instantiating Dad
36             */
37            dad = new Daddy(new AgentID("qpid://MrSmith@localhost
                  :8080"));
38
39            /**
40             * Instantiating sons
41             */
42            olderSon = new Boy(new AgentID("qpid://Timmy@localhost
                  :8080"), 7, dad.getAid());
43            youngerSon = new Boy(new AgentID("qpid://Bobby@localhost
                  :8080"), 5, dad.getAid());
44
45            /**
46             * Execute the agents
47             */
```

```
48              dad.start();
49              olderSon.start();
50              youngerSon.start();
51          } catch (Exception e) {
52              logger.error("Error  " + e.getMessage());
53          }
54      }
55  }
```

7.6.4 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

8

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 and SOA-like services. THOMAS (Methods, Techniques and Tools for Open Multi-Agent Systems) framework has been integrated with Magentx2 with this purpose.

8.1 Overview of THOMAS framework

The THOMAS framework tries to communicate agents and SOA-like 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.

Different types of virtual organizations are supported by the THOMAS framework. Each organization can contain others organizations. Furthermore, diverse roles can be assigned to

each organization. These roles are characterized by some attributes which restrict its behavior regarding THOMAS framework. Thus, agent must play those roles in order to belong to an organization.

In addition, Magentix2 offers a new communication mechanism based on the virtual organizations structure. Thus, this organizational messaging allows mass communication among agents of an organization, taking into account the type of roles agents play.

THOMAS framework 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 a yellow page service and also 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 and the roles the agents play.

8.1.1 Roles in THOMAS

Roles in THOMAS are defined with a *RoleName* that acts as an identifier. Furthermore, each role is always associated with a particular organizational unit. Also, each role has the following attributes: $< Position, Accessibility, Visibility >$.

- **Position**. The possible values that can take are: *Member*, *Supervisor*, *subordinate* or *Creator*.
  - This value must be assigned to roles when they are created.
  - Not all the values are allowed to any type of organization (see section 8.1.2).
  - Anytime an agent registers an organization, a new role with a creator position is automatically assigned him. This position gives permission to register and deregister organizations and roles. In addition, it is possible to assign a *creator* role using the services offered by the THOMAS API (see section 8.2.1).

  In Table 8.1 a summary of the role behavior taking into account its position is shown.

- **Accessibility**. This attribute allows controlling who can acquire roles. Specifically, the permitted values are:

  - External: roles can be acquired by agents who do not play any role in the organization.
  - Internal: in this case, it is necessary participate into the organization, that is, the requested agent should play some role in the organization.

- **Visibility**. By means of this attribute it is possible to control what information about roles is provided. Concretely, the allowed values for this attribute are:
  - Public: information services always provide the requested information.
  - Private: the solicited information is provided only if the requested agent belongs to the same organization where the role is registered.

The relationship between accessibility and visibility attributes can be seen in the table 8.2.

8.1.2 Units in THOMAS

THOMAS framework gives support for three types of organizations or units: *Flat*, *Team* and *Hierarchy*. Each type of organization is governed by different structural norms. In the Table 8.3 a comparative of these types of units is shown.

Furthermore, there is an organization created by default named *Virtual*. This is a flat unit which represents the THOMAS world, and it is the starting point to enter in the system. A role named *Participant* is always available in the *Virtual* organization. This role has the following attributes:

$< RoleName = Participant, Position = Creator, Visibility = Public, Accessibility = External >$

Thus, an agent who wants to enter in Thomas world only needs to play the role *Participant*.

Position	Unit Types	**Behavior Allowed**
Creator	All types	- Send organizational messages is not allowed - Register/Deregister units and roles - Allocate/Deallocate roles to other agents - Change the hierarchical relations between organizations - Acquire other roles - Request information about the roles played by other agents - Request information about organizations and its elements
Member	Flat Team	- Send organizational messages. These messages will be received for all organization members - Register/deregister roles - Allocate/deallocate roles to other agents - Acquire another role - Request information about the roles played by other agents - Request information about organizations and its elements
Supervisor	Hierarchy	- Send organizational messages. These messages will be received for all organization members - Register/Deregister units and roles - Allocate/Deallocate roles to other agents - Change the hierarchical relations between organizations - Acquire other roles - Request information about the roles played by other agents - Request information about organizations and its elements
Subordinate	Hierarchy	- Send organizational messages. These messages will be received for all supervisor agents of the organization - Acquire other roles - Request information about the roles played by other agents - Request information about organizations and its elements

Table 8.1: Agent behavior depending of its position

Visibility	Accessibility	Behavior
Public	*External*	Role fully accessible and transparent. The role is visible throughout the system: -*External*: any agent can acquire this role -*Public*: any agent can request information about the role
Public	*Internal*	Role with restricted access, although visible throughout the system: -*Internal*: in order to acquire this role, agents must participate in the organization in which the role was registered(or in its Parent Organization) -*Public*: any agent can request information about the role
Private	*External*	Role fully accessible, but with limited visibility: -*External*: any agent can acquire this role -*Private*: information about the role is only provided if requested agents participate in the organization in which the role was registered(or in its Parent Organization).
Private	*Internal*	Maximum protection of the role. The unit acts like a black box for these roles: -*Internal*: agents must participate in the organization or in its parent organization to acquire the role -*Private*: agents can request information about the role if they participate in the organization in which the role was registered(or in its Parent Organization).

Table 8.2: How visibility and accessibility attributes affect roles

	FLAT	**TEAM**	**HIERARCHY**
Allowed role positions	Creator Member		Creator Supervisor Subordinate
What agents can send organizational messages?	Any agent from this organization, except those who only play roles with *Position* = *Creator*		
What agents can receive organizational messages?	Any agent from this organization, except who what only play roles with Position = Creator		*Supervisors* receive all messages. *Subordinates* only receive messages sent by *Supervisors*. *Creators* do not receive messages.
What agents can register/deregister organizations?	Any agent from this organization which plays roles with *Position* = *Creator*		
What agents can join an organization to another?	Any agent from this organization which plays both a role with Position = Creator and a role with Position = Creator in the new Parent organization.		
What agents can register/deregister roles?	Any agent from this organization. *Creator* Agents from other organizations	Any agent from this organization	Agents from this organization with *Position*. = *Creator* or *Position* = *Supervisor*
What agents can allocate/deallocate roles?			
What agents can request "public" information?	Any agent from any organization.		
What agents can request "private" information?	Any agent from this organization.		
What agents can request information about organizations?	Any agent from any organization.	Any agent from this organization or any agent from its parent unit.	
What agents can request information about attributes of the roles?			
What agents can acquire roles with accessibility = external?	Any agent from any organization.		Any agent from any organization. But, it is not allowed to acquire a role with *Position* = *Creator* or *Position* = *Supervisor* if the agent already plays a role with Position = Subordinate in the organization.
What agents can acquire roles with accessibility = internal in the organization?	Any agent from this organization or any agent from its parent unit		Any agent from this organization or any agent from its parent unit. But, it is not allowed to acquire a role with *Position* = *Creator* or *Position* = *Supervisor* if the agent already plays a role with Position = Subordinate in the organization.

Table 8.3: Differences among the diverse organization types

8.1.3 Service Facilitator

The SF is a support mechanism 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 framework. The SF can find services searching for a given service profile. This is done using matchmaking mechanisms over the inputs, outputs and keywords of the services to search.

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.

Services registered and discovered by SF in THOMAS are demanded or offered by some agent or organization (Figure 8.1). Demanded services are services which are wished by autonomous entities, but no agent or organization provides them. Offered services are provided and registered in the THOMAS environment by agents or organizations. The final execution of this services will be finally made by a web service, an internal behavior of an agent or an agent internal call to a web 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 and removing services from the SF directory (RegisterService, DeregisterService).
- **Affordability:** they allow managing the association between providers and their services

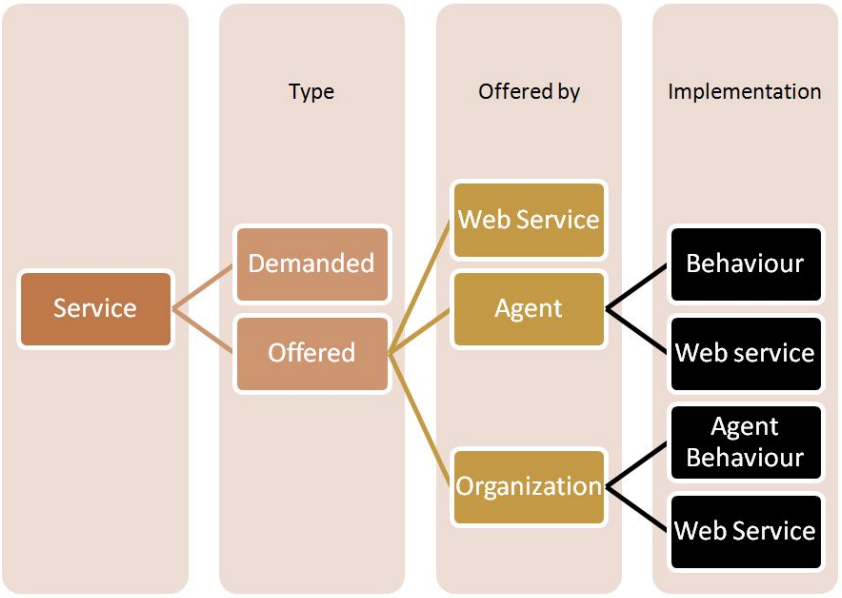

Figure 8.1: Handled Services: demanded and implementations of offered services supported

(RemoveProvider).

- **Discovery:** services in charge of searching and composing services as an answer to user requirements (SearchService, GetService).

8.1.4 Organization Manager Service

This component is in charge of organizations life cycle management, including specification and administration of their structural components (roles, organizative and units) 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 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 ([Val et al., 2009]), classified in:

- **Structural services:** which modify the structural organization specification, i.e. roles and organizational units (RegisterRole, DeregisterRole, RegisterUnit, DeregisterUnit, JointUnit).
- **Informative services:** that give information of the current state of the organization (InformRole, InformUnit, InformUnitRoles, InformAgentRole, InformMembers, QuantityMembers).
- **Dynamic services:** that allow managing role enactment and dynamic entry/exit of agents (AcquireRole, LeaveRole, AllocateRole, DeallocateRole) .

8.2 Programming agents which use THOMAS

8.2.1 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.

8.2.1.1 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 has to be specified. There are two options. In the first one, the url where OMS services are deployed is taken from the *settings.xml* configuration file (please see section 10). Notice that this is the recommended option.

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

On the contrary, in the second option, the url where OMS services are deployed (.war) should be specified in the constructor if the user does not want to use the default url from the *settings.xml* configuration file:

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

OMSProxy provides a developer with a set of methods to manage available services, this services are divided into different types and sub-types. Tables 8.4, 8.5, 8.6 and 8.7 show respectively these sub-types.

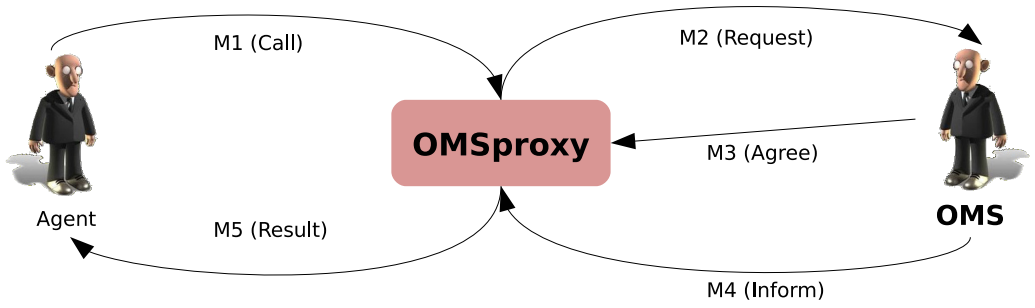

Figure 8.2: Interaction between user agent and OMS agent through the OMSProxy

8.2.1.2 SFProxy

We can find the SFProxy 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 has to be specified. There are two options. In the first one, the url where SF services are deployed is taken from the *settings.xml* configuration file (see section 10). Notice that this is the recommended option.

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

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*. **Visibility:** Is defined (*public*) or from inside (*private*). **Position:** Position inside the unit, such as *member*, *supervisor* or *subordinate*.
DeregisterRole	Removes a specific role from a unit scription from a unit	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
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*. **ParentUnit:** Identifier of the parent unit. **Creator:** The name of the new creator role.
DeregisterUnit	Removes a unit from an organization	**UnitID:** Identifier of the unit.
JointUnit	Updates the parent unit	**UnitID:** Identifier of the unit. **ParentUnit:** Identifier of the new parent unit.

Table 8.4: OMS Proxy: Structural services API

On the contrary, in the second option, the url where SF services are deployed (.war) should be specified in the constructor if the user does not want to use the default url from the *settings.xml* configuration file:

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

OMS-Service	Description	Parameters
InformRole	Provides a role description of a specific unit	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit.
InformUnit	Provides unit description	**UnitID:** Identifier of the unit.
InformUnitRoles	Used for requesting the list of roles that have been registered inside a unit.	**UnitID:** Identifier of the unit.
InformAgentRole	Requesting the list of roles and units in which an agent is in a specific moment.	**RequestedAgentID:** Identifier of the agent requested.
InformMembers	Indicates entities that are members of a specific unit. Optionally, it is possible to specify a role and position of this unit, so then only members playing this role or position are detailed.	**UnitID:** Identifier of the unit. **RoleID:** Identifier of the role. **PositionID:** Identifier of the position inside the unit, such as member, supervisor or subordinate.
QuantityMembers	Provides the number of current members of a specific unit. Optionally, if a role and position is indicated then only the quantity of members playing this roles or position is detailed.	**UnitID:** Identifier of the unit. **RoleID:** Identifier of the role. **PositionID:** Identifier of the position inside the unit, such as member, supervisor or subordinate.

Table 8.5: OMS Proxy: Informative services API

SFProxy provides a developer with a set of methods to manage available services, as Table 8.8 shows.

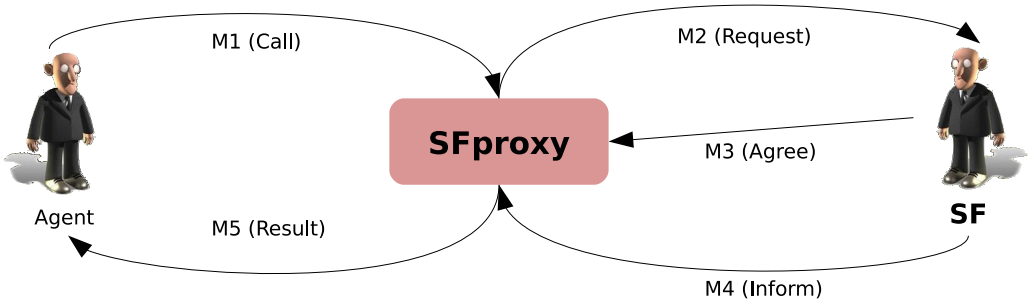

Figure 8.3: Interaction between user agent and SF agent through the SFProxy

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.
AllocateRole	Forces an agent to acquire a specific role.	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit. **TargetAgentID:** Identifier of the agent.
DeallocateRole	Forces an agent to leave a specific role.	**RoleID:** Identifier of the role. **UnitID:** Identifier of the unit. **TargetAgentID:** Identifier of the agent.

Table 8.6: OMS Proxy: Dynamic services API

OMS-Service	Description	Parameters
buildOrganizationalMessage	Builds a new organizational message.	**OrganizationID:** Identifier of the organization.

Table 8.7: OMS Proxy: Organizational messaging service API

8.2.1.3 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).

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

```
1
2 ArrayList<String> resultRegister= sfProxy.registerService("http://
    localhost:8080/testSFservices/testSFservices/owl/owls/Addition.
    owl");
```

Type	Service	Description	Inputs	Output
Registration	Register Service	Registers a service or part of it	Service URL	Description of what is registered and complete OWL-S specification
	Deregister Service	Deregisters a complete service	Service profile	Ok if it is deregistered. Else error description
Affordability	Remove Provider	Removes a provider (agent, organization or web service) of a service	(Service profile, provider name or grounding ID)	Ok if it is removed. Else error description
Discovery	Get Service	Gets the OWL-S description of the service	Service profile	OWL-S specification of the service or error description
	Search Service	Searches a service	(Inputs, Outputs, Keywords)	Weighted services list or error description

Table 8.8: SF Proxy API

The Register Service tries to register the service that is specified as parameter. The parameter is the original URL of the OWL-S specification of the service. The contents of the OWL-S specification of a service must have the profile that describes the service and the process that describes its implementation. If one or more providers (agents or organization) are specified in the *profile:contactInformation* of the service, it means that the service is provided by agents or/and organizations. If there is one or more groundings, it means that the service is provided by a web service.

The Register Service returns a response describing if the service has been entirely registered or the number of providers and groundings added to an already registered service profile. In all cases, it returns a description of what is registered or not, and an OWL-S specification of the registered services or all data of the already registered service.

An agent can register its own implementation of an existing profile or create a new service from the scratch. In all cases, the OWL-S specification must have the profile and the process, and the Register Service will detect if the profile is already registered. In this case, the providers (agents, organizations, or groundings/web services) of the new specification will be added to the registered service in the SF database.

8.2.1.4 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, providers, list of roles or organizational units).

Oracle needs just the OWL-S string containing the service specification to analyze it (as a parameter in the constructor of the class, see the example). This specification can be obtained with the method `GetService` of the `SFProxy` class.

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

```
1  //obtain the the service OWL-S specification
2  String serviceOWLS = sfProxy.getService(ServiceProfile);
3
4  //load in the oracle the service OWL-S specification and parse it
5  Oracle oracle = new Oracle(serviceOWLS);
6
7  //access to the service OWL-S information through the oracle
8  ArrayList<String> service_inputs = oracle.getOwlsProfileInputs();
9  ArrayList<Provider> providers = oracle.getProviders();
10 ArrayList<String> providersGroundingWSDL = oracle.
      getProvidersGroundingWSDL();
```

8.3 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.

8.3.1 Acquire Role

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

```
1
```

Figure 8.4: Agent interaction protocol to acquire a role.

```
2  omsProxy.acquireRole("participant", "virtual");
```

8.3.2 Service Registration

- **Registering a new service**. An agent wants to register its own service in the SF (Figure 8.5). The OWL-S specification of the service must be available in a url. This specification has to follow the OWL-S standard, having the profile and process of the service. It also needs to specify the provider or providers of the service in the following way:

  – If the provider is an agent or an organization, the profile has to be specified as a *provider* in the OWL-S specification. The provider is defined following the *provider.owl* ontology, located in the *webapps* folder of the Apache Tomcat in Magentix2 installation folder (concretely in *webapps/ontologies*). The parameters of the provider are: entity id, entity type (agent or organization), communication language performative to use in the petition. An example of this structure is shown below:

```
1
2    <profile:contactInformation>
3      <provider:Provider rdf:ID="AdditionAgent">
4          <provider:entityID rdf:datatype="^^xsd;string">
               AdditionAgent</provider:entityID>
5          <provider:entityType rdf:datatype="^^xsd;string">
               Agent</provider:entityType>
6          <provider:language rdf:datatype="^^xsd;string">FIPA-
               ACL</provider:language>
7          <provider:performative rdf:datatype="^^xsd;string">
               REQUEST</provider:performative>
8      </provider:Provider>
9    </profile:contactInformation>
```

– If the provider is a web service, the corresponding grounding has to be specified as standard specification of the OWL-S services including its WSDL document URL to execute the service.

Figure 8.5: Agent interaction protocol to register a service.

Thus, the agent can make a call to register a service specifying the url of the service OWL-S specification, as we can see in this code example:

```
1
2 ArrayList<String> resultRegister = sfProxy.registerService("http
    ://localhost:8080/testSFservices/testSFservices/owl/owls/
    Addition.owl");
```

- **Registering new providers**. When a service is already registered, it is possible to add new providers (agents, organizations or web services). The *RegisterService* of the SF detects automatically if the profile of the given OWL-S specification is already registered in the SF and just adds the new providers to the SF database. The given OWL-S specification must have the same profile so the SF recognize that is the same service and it associates the new providers to the registered service (see Figure 8.6).

Figure 8.6: Agent interaction protocol to register new providers.

8.3.3 Provide services

In THOMAS, there are different ways of providing a service, as represented in Figure 8.1. On the one hand, in the cases that the providers are registered in the SF as agents or organizations, the requests to that services will be sent to the corresponding agent or organization as specified in the OWL-S registered in the SF. The agents or organizations can execute their service as an internal behaviour or as a web service, but this is not visible to the requester.

On the other hand, if the providers of a service are registered in the SF as web services (groundings), the requester of these services will have to execute the service by itself, using the class *ServiceTools* of the package *es.upv.dsic.gti_ia.organization* (see Section 8.4.3).

8.4 Programming Agents that Request Services

This section describes how an agent that requires services from other agents can search and request services in THOMAS.

8.4.1 Acquire Role

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

Figure 8.7: Agent interaction protocol to acquire role.

```
1 omsProxy.acquireRole("participant", "virtual");
```

8.4.2 Service Search Process

- **Search of a service.** Once the agent is registered, it can use the `SearchService` service to find required services. The method *SearchService* of the SFproxy makes a request to this service with the following parameters:

- List of inputs of the service to search. These inputs has to be specified as types of an ontology. In *ServiceTools* class the main types (integer, float, double, string, boolean) are specified as constants to be easier to the developer.
- List of outputs of the service to search. These inputs has to be specified as types of an ontology, as the inputs explained before.
- List of keywords of the text description of the service to search. Each *String* added to this list will compute to find it in the text description of the service profiles registered in the SF.

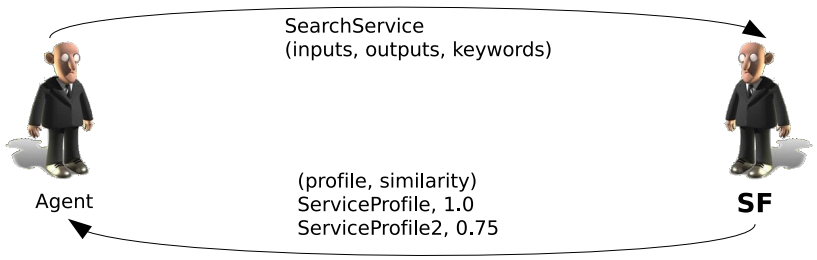

Figure 8.8: Agent interaction protocol to search service.

The following example shows how to search for a service. The desired service should have the word ”addition” in its description, two double inputs an one double output.

```
1
2  ArrayList<String> searchInputs = new ArrayList<String>();
3  ArrayList<String> searchOutputs = new ArrayList<String>();
4  ArrayList<String> searchKeywords = new ArrayList<String>();
5
6  searchInputs.add(ServiceTools.OntologicalTypesConstants.DOUBLE);
7  searchInputs.add(ServiceTools.OntologicalTypesConstants.DOUBLE);
8
9  searchOutputs.add(ServiceTools.OntologicalTypesConstants.DOUBLE)
       ;
10
11 searchKeywords.add("addition");
12
13 ArrayList<ArrayList<String>> foundServices;
14
15 do {
16         // Waiting for services
17         try {
18                 Thread.sleep(2 * 1000);
```

```
19          } catch (InterruptedException e) {
20
21                  e.printStackTrace();
22          }
23          foundServices = sfProxy.searchService(searchInputs,
               searchOutputs, searchKeywords);
24
25  } while (foundServices.isEmpty());
```

8.4.3 Service Request Process

- **Request a service.** After the agent has found the desired service, it has to get the main information of this service to make a request. So the agent has to use the *GetService* to obtain the required data to specify the inputs of the service and to know the provider to ask and its parameters. In the obtained OWL-S specification with *GetService* is specified the providers of the service and their parameters. To make a request of the service, the provider has to be an agent or organization. In case that the providers are web services (with a grounding in the specification), the service has to be executed by the requester agent.

  Using the *Oracle* class provided in the package *es.upv.dsic.gti_ia.organization* the OWL-S specification is parsed to obtain the information of the service, including its inputs and its providers. There are two different ways to request a service depending on the type of the providers:

  – If the providers are **agents** or **organizations**, their parameters (entity ID, entity type, language and performative to make the request) are obtained from the OWL-S specification using the *Oracle* class. Then, the requester agent should properly build the message to make the request and wait for the response of the provider to obtain the result. In *ServiceTools* class it is possible to find the method *buildServiceContent* which returns an XML description, receiving as parameters the inputs and the name of the requested service. This description is understood by all THOMAS services and the services of the provided examples.

  – If the providers are **web services**, the URL of the WSDL documents are obtained from the OWL-S specification using the *Oracle* class. Then, the requester agent has to execute the service by itself. To perform this task, the class *ServiceTools* of the package *es.upv.dsic.gti_ia.organization* provides the method *executeWebService* to

facilitate the execution of web services to the developers. This method receives as parameters the url of the service WSDL document to execute it, and the inputs of the service. The inputs can be specified in two ways: in a *HashMap* giving the name of the input and its value, or with an XML string in the following form:

```
<inputs>
 <inputX>valueX</inputX>
 <inputY>valueY</inputY>
</inputs>
```

In the following example, the requester agent tries to make a request of the found service to the agent or organization providers if there are (providers list, lines 28-51). If there are not agent or organization providers (providers list is empty), then it tries to obtain the Web Services (groundings WSDL documents) that provide the service (providersGroundingWSDL list, lines 52-60). In this case, the requester agent has to execute the service by itself using the *ServiceTools* class provided by THOMAS.

```
// Requesting the execution of the Addition service

// get the first service found because it is the most suitable
String serviceOWLS = sfProxy.getService(foundServices.get(0).get(0))
    ;

Oracle oracle = new Oracle(serviceOWLS);

// get service inputs
ArrayList<String> serviceInputs = oracle.getOwlsProfileInputs();

// put the service inputs values
HashMap<String, String> agentInputs = new HashMap<String, String>();

for (String input : serviceInputs) {
        if (input.equalsIgnoreCase("x"))
                agentInputs.put(input, "3");
        else if (input.equalsIgnoreCase("y"))
                agentInputs.put(input, "4");
        else
                agentInputs.put(input, "0");
}
```

```
22
23  // agents or organizations providers
24  ArrayList<Provider> providers = oracle.getProviders();
25  // web services providers
26  ArrayList<String> providersGroundingWSDL = oracle.
        getProvidersGroundingWSDL();
27
28  if (!providers.isEmpty()) {
29          System.out.println("[" + this.getName() + "]" + " Requesting
                Addition Service (3+4)");
30
31          // Building the ACL message
32          ACLMessage msg = new ACLMessage();
33          msg.setReceiver(new AgentID(providers.get(0).getEntityID()))
                ;
34          msg.setLanguage(providers.get(0).getLanguage());
35          msg.setPerformative(providers.get(0).getPerformative());
36          msg.setSender(getAid());
37
38    // Building the content of the message in XML
39          String content = st.buildServiceContent(oracle.
                getServiceName(), agentInputs);
40
41          // ACL message content is formed by XML format with service
                name
42          // and
43          // inputs
44          msg.setContent(content);
45
46          this.send_request(msg);
47
48          ServiceTools st = new ServiceTools();
49          HashMap<String, String> outputs = new HashMap<String, String
                >();
50          st.extractServiceContent(requestResult, outputs);
51          resultEquation = outputs.get("Result");
52  } else if (!providersGroundingWSDL.isEmpty()) {
53          System.out.println("[" + this.getName() + "]" + " Executing
                Addition Service (3+4)");
54
```

```
55          HashMap<String, Object> resultExecution = st.
                executeWebService(providersGroundingWSDL.get(0),
56                          agentInputs);
57
58          Double resultDouble = (Double) resultExecution.get("Result")
                ;
59          resultEquation = resultDouble.toString();
60  } else {// no providers for this service
61          System.out.println("[" + this.getName() + "]" + " No
                providers found for Addition Service (3+4)");
62  }
```

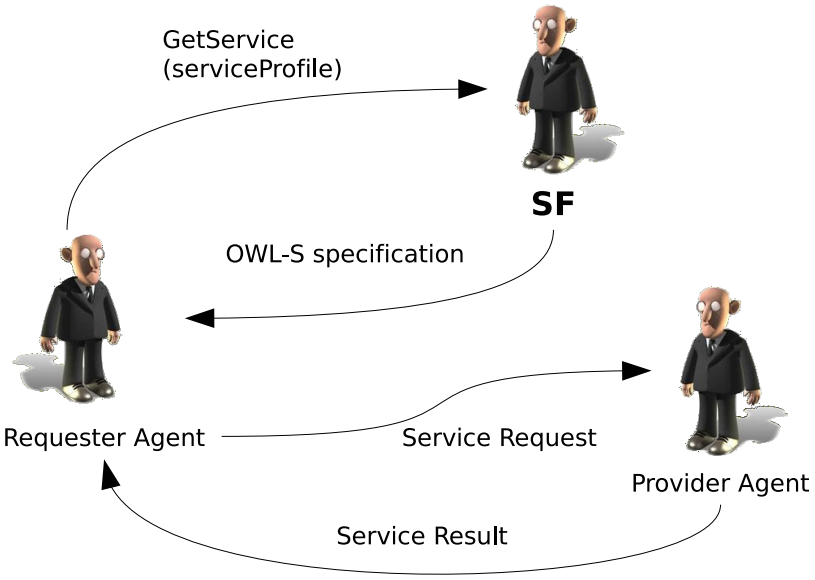

Figure 8.9: Agent interaction protocol to request a service.

8.5 Running THOMAS Example

The examples folder of the Magentix2 packages contains a basic THOMAS example. In this example there are four agents:

- **Initiator**. This agent build the organizational framework needed. First, it registers the *Calculator* and *School* organizations. Then, it registers the roles needed in each organi-

zation. As this agent register each unit, it also plays the role *Creator* of them. At the end of the example, it deallocates all roles and deregister all units, roles and services.

- **Addition**. This agent acquires the role *Operation* of the *Calculator* organization. It also registers and provides the service *Addition*.
- **Product**. This agent acquires the role *Operation* of the *Calculator* organization. It also registers the services *Product* and *Square*. It only provides the service *Product*.
- **James**. It represents an student which needs computing some operations and demand the other their services. This agent acquires the role *Student* of the *School* organizations. It looks for the different services it needs (Addition, Product and Square) to calculate a simple mathematical operation. Then either ask the service providers in carrying out such services, or run them directly if no provider.

Services in the THOMAS example are deployed in the *webapps* folder of the Apache Tomcat in Magentix2 installation folder. The OWL-S specifications are located in:

```
webapps/testSFservices/testSFservices/owl/owls/
```

The WSDL documents can be found in:

```
webapps/testSFservices/WEB-INF/services/service_name/META-INF/
    service_name.wsdl
```

The following services are used in the THOMAS example:

- **Product**: This service multiplies two input numbers (doubles) and returns the product (double). It is provided by an agent behavior.
- **Addition**: This service adds two input numbers (doubles) and returns the addition (double). It is provided by agent that calls to a web service.
- **Square**: This service squares an input number (double) and returns the result (double). It is provided by a web service.

Following, we briefly explain the actions of the THOMAS example shown in the diagram of Figure 8.10:

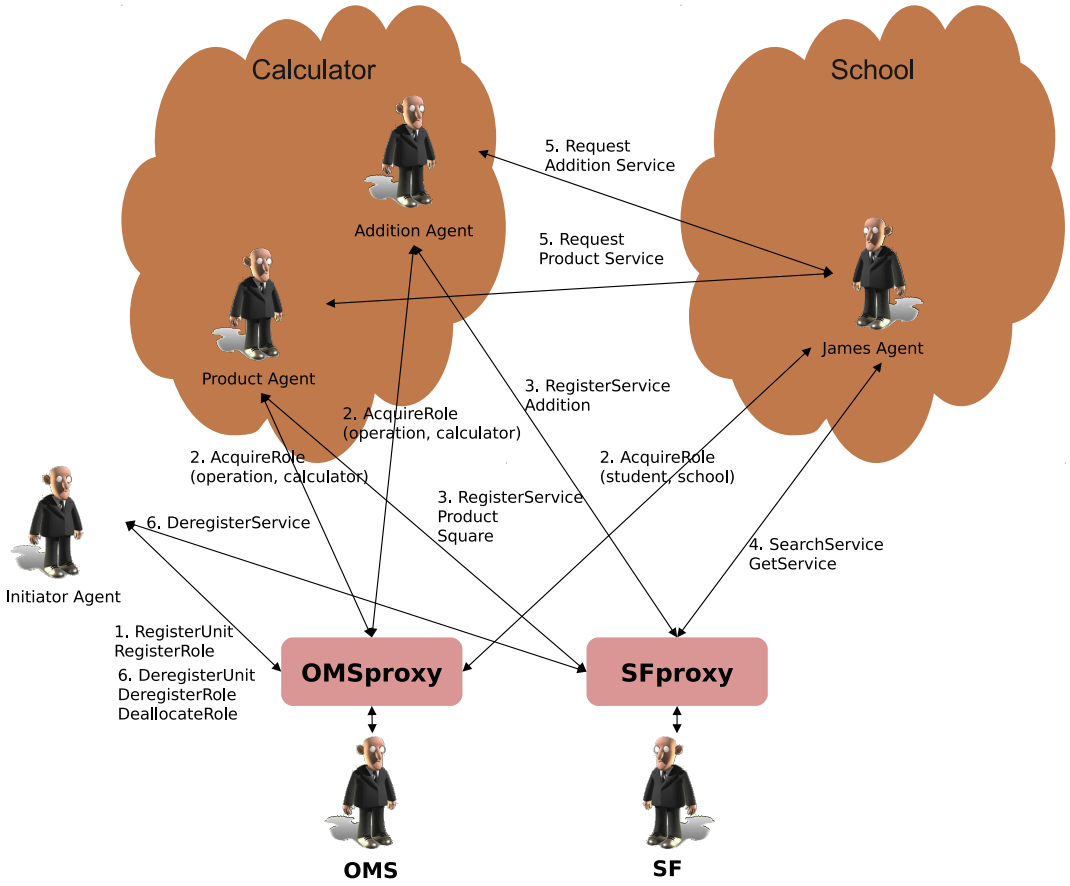

Figure 8.10: Thomas Example diagram

1. The Initiator acquires the role *participant* of the default unit *virtual*. Then, it registers the organizations *Calculator* and *School*. *Calculator* is a team unit, its parent unit is virtual and the name of the default role is *creator*. On the other hand, the unit *School* is a flat organization, its parent unit is virtual and the name of the default role is also *creator*. Once the agent registers the units, it also creates the roles needed. So, first the Initiator agent registers a role named *operator* (member, public and external) inside the unit *Calculator*. Secondly, it registers a role named *student* (member, public and external) inside the unit *School*.

2. The Addition agent and the Product agent acquire the role *operation* of the *Calculator* organization. Also, the James agents acquires the role *student* of the *School* organization.

3. The Addition agent registers the Addition service in the SF. Also, the Product agent registers the Product and Square services in the SF.

4. The James agent wants to calculate the equation $(5(3+4))^2$. Firstly, it searches for a service to add two numbers and it finds the Addition service. It makes a *GetService* of this service to obtain the required information to request it. It also searches for a service to make the product and another one to square. In these cases, it finds the Product service and the Square service. It makes a *GetService* of the two services to obtain their information.

5. The James agent requests the Addition service to add 3 and 4. Then, with this result requests the Product service to multiply it by 5. Finally, the result of this operation is used to square it with the Square service, that is provided by a web service. This implies that the James agent has to execute it itself by using the Service Tools provided in the Magentix2 API. The James agent has obtained the final result of the equation that it wanted to solve. Then, it sends a message to the Initiator agent to inform that the example has ended.

6. The Initiator agent receives the message informing that the example has ended. It deallocates all the roles and deregisters all services, roles and units.

To run the example the user has to enter in the Magentix2/examples folder where it has been installed. Then, the user has to execute the *Start-ThomasExample.sh* script with administrator privileges:

```
cd Magentix2/examples/bin/
sudo sh Start-ThomasExample.sh
```

8.6 Programming agents which use organizational messaging

This section describes how an agent that wants to send a message to an organization can build, fill the fields of an ACLMessage and send the organizational message. The following example is used in order to illustrate this section. It is formed by: the sender agent that acquires role member in the unit team and two receivers agents with the role member in the unit team (Figure 8.11). The unit virtual is parent of the unit team.

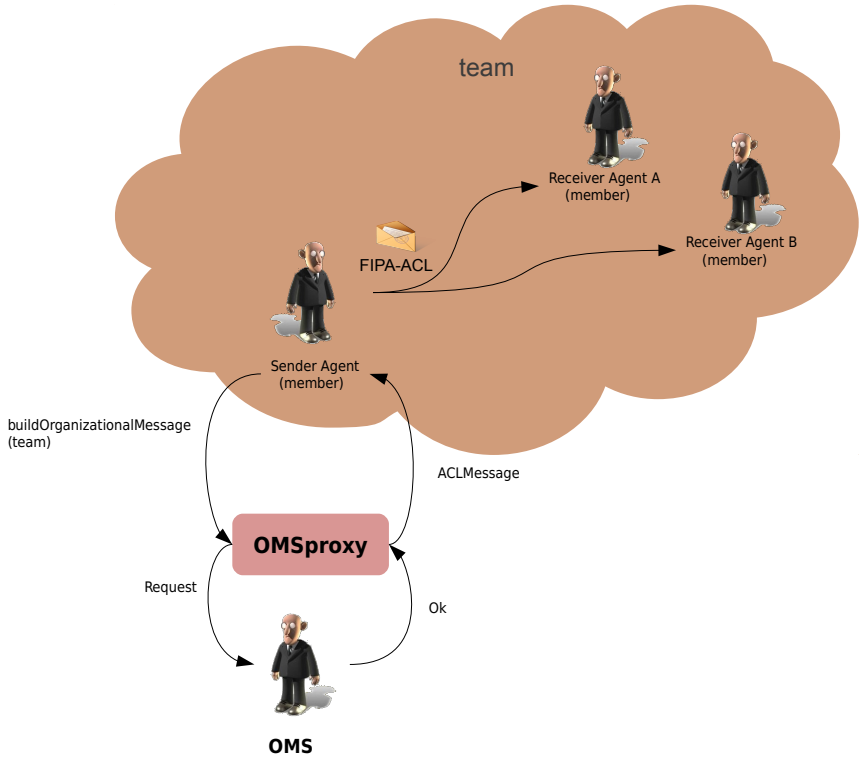

Figure 8.11: Organizational messaging: Example diagram.

8.6.1 Acquire Role

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

Figure 8.12: Agent interaction protocol to acquire role.

```
1 omsProxy.acquireRole("participant", "virtual");
```

- **Registration of an agent inside the unit.** The agents request to be registered as a *member* inside the unit *team*.

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

8.6.2 Message build process

- **Build an organizational message.** Once the agent is inside the unit with an specific role, it has to use the *buildOrganizationalMessage* to obtain the required organizational message specifying the identifier of the organization:

```
1  ACLMessage msg = omsProxy.buildOrganizationalMessage("team");
```

8.6.3 Message complete process

- **Add ACL message fields.** After the message is built, it can add the fields of an ACLMessage, such as content, performative or language:

```
1  msg.setContent(6+" "+3);
2  msg.setPerformative(ACLMessage.REQUEST);
3  msg.setLanguage("ACL");
```

8.6.4 Message send process

- **Send an organizational message.** When a message is formed, it can send the message using the *send()* method:

```
1  send(msg);
```

or it reached a state of type send in CFactories:

```
1  class REQUEST_Method implements SendStateMethod {
2
3
4    public String run(CProcessor myProcessor, ACLMessage
       messageToSend) {
5
```

```
6
7       OMSProxy omsProxy = new OMSProxy(myProcessor);
8       ACLMessage msg;
9       String state = "WAIT";
10      try {
11          msg = omsProxy.buildOrganizationalMessage("team");
12          messageToSend.copyFromAsTemplate(msg);
13
14          messageToSend.setPerformative(ACLMessage.REQUEST);
15          messageToSend.setLanguage("ACL");
16          messageToSend.setContent(6+" "+3);
17
18      } catch (THOMASException e) {
19          e.printStackTrace();
20          state="FINAL";
21      }
22      return state;
23    }
24  }
```

CHAPTER

9

HTTP Interface

In order to allow interaction between a Magentix2 agent and the outside world, an HTTP interface service has been developed. The service is automatically started when the StartMagentix.sh script is executed, so no special action is required. A common use for the HTTP Interface service is a webpage that allows its users to monitor and interact with the agents running in Magentix2. The examples in this section show how to interact with Magentix2 agents using a web page and using Javascript and PHP. However, the use of the HTTP interface is not exclusive for web pages and can be used in other scenarios.

9.1 Framework

The functionality of the HTTP Interface service is quite simple. It listens to the port 8081 and expects to get an HTTP POST request. The HTTP interface extracts the target agent from the HTTP POST body message and sends that body message as the content of an ACLMessage to the target agent.

The information that the HTTP POST request contains must be a well formed JSON[1] object. Besides, the JSON object has to obbey to the following guideline:

[1] `http://en.wikipedia.org/wiki/JSON`

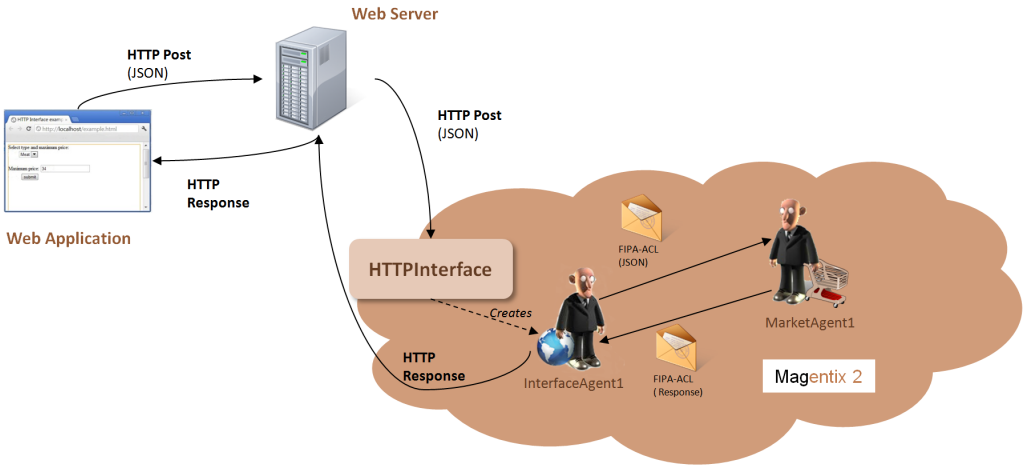

Figure 9.1: HTTP Interface framewok

- The JSON object must contain a field called `agent_name`. The content of this field is the name of the target agent that will receive the content of the HTTP request.
- The object must contain a field called `conversation_id`. The content of this field should be a unique identifier. This identifier will be used in the messages sent between the HTTP interface service and the target agent.
- The object must also contain a field called `content`. The content of this field is the information that the target agent will manage.

When the HTTP Interface gets the HTTP POST request, it reads the JSON object included in the message body of the HTTP POST request. The HTTP Interface extracts from the JSON object the field `agent_name`. The content of this field specifies the target agent. The content of the field `conversation_id` of the JSON object will be used as the `conversation_id` of the ACLMessage that will be sent to the target message. Finally the entire JSON is used as the content of the ACLMessage. Please note that the entire JSON is used as content, therefore the target agent has to be capable to deal with the fields `agent_name` and `conversation_id`. Magentix2 includes the XStream library, which is able to manage JSON objects in java. For more information about XStream and JSON, please, go to `http://xstream.codehaus.org/json-tutorial.html`. For better understanding, an example of a valid JSON obect is given below. In this example a web page allows users to ask for supermarket products to an agent. The user has to specify the type of the product and the maximum price she is willing to pay.

```
{``agent_name'':``Mike'', ``conversation_id'':``conv1'',``content
   '':{``type'':``Fruit'', ``max_price'':23.0}}
```

The HTTP interface does not send the ACLMessage directly to the target agent, instead, it creates a dummy agent that will send the ACLMessage. This functionality allows the HTTP interface to manage multiple and concurrent HTTP POST requests. The name of a dummy agent is InterfaceAgentX, where X will be a number. As an example, the receiver, conversation_id, sender and content fields of the ACLMessage that the target agent would recive after the HTTP interface would receive the previous JSON object would be the following:

```
ACLMessage.receiver = Mike
ACLMessage.conversation_id = conv1
ACLMessage.sender = InterfaceAgent1
ACLMessage.content = {``agent_name'':``Mike'', ``conversation_id
   '':``conv1'',``content'':{``type'':``Fruit'', ``max_price
   '':23.0}}
```

The target agent can respond to the HTTP request sending an ACLMessage to the InterfaceAgent that sent the ACLMessage with the HTTP request. The content of the ACLMessage sent to the InterfaceAgent will be used as the message body of the reply to the HTTP request. Continuing with the previous example, the ACLMessage that the agent Mike would send to the InterfaceAgent1 could be:

```
ACLMessage.receiver = InterfaceAgent1
ACLMessage.conversation_id = conv1
ACLMessage.sender = Mike
ACLMessage.content = {``name'':``Banana'', ``id'':123, ``price
   '':21.0, }}
```

As you may note, the content of this message is a JSON object but it does not follow the guideline explained previously. This is not a problem, the reply content is free and may contain any type of data.

9.2 Tools

Magentix2 provides two tools to facilitate the communication between Magentix2 agents and a webpage: *Magentix2.js* and *Redirect.php* . This tools can be obtained from the examples folder of the Magentix2 distribution.

9.2.1 Magentix2.js

Magentix2.js is a Javascript library that facilitates the creation of JSON objects that are accepted by the HTTP interface. The two main functions of the library are createJSONObject(agent_name, conversation_id, content) and createJSONObject(agent_name, content). The first one returns a JSON object with the conversation_id field set to the value passed as parameter. The second one returns a JSON object with a new conversation_id, this id is a UUID[2]. The parameter content has to be an associative array of elements that will be used as the content of the resulting JSON object.

9.2.2 Redirect.php

This php can redirect an HTTP POST request to any address. You can use it to redirect HTTP POST requests to your HTTP interface. This php requires the Pear HTTP_Request2 library, you can get this library from `http://pear.php.net/package/HTTP_Request2/redirected`.

9.3 Example

In this section only a brief explanation of how to interact with a Magentix2 agent from a webpage is shown. For the complete example, please, refer to the folder examples/httpinterface of your Magentix2 distribution.

The example is composed by two parts:

- example.html: This web page contains a form. The user specifies a type of product and a maximum prize. The user will get a product that satisfies her specifications.

[2] `http://en.wikipedia.org/wiki/Universally_unique_identifier`

- MarketAgent: This agent receives the request from the web page example.html and returns an appropriate product.

The example.html webpage uses several javascript libraries in order to send the HTTP POST request asynchronously. These libraries are included in the head section of the html code. The html file also uses our javascript library Magentix2.js. When the user submits the form the javascript function transformData is called.

```
 function transformData(){
        var pp=$("#f1").serializeArray();
        var data=createJSONObject("MarketAgent", pp);
        requestAppData(data);
}
```

This function takes every input of the form and constructs an associative array with them. Then it calls to the function createJSONObject of the Magentix2.js library. The parameters are "MarketAgent" the name of our agent and the array with the content of the form. The resulting JSON object will be something like this:

```
{"agent_name":"MarketAgent","conversation_id":"5a1a2c78-91b2-465c
   -818f-bd750ea45f4f","content":{"type":"fruit","max_price":"10"}}
```

Finally the transformData calls the function requestAppData with the JSON object as parameter.

```
function requestAppData(data){
        $.post("redirect.php", data, function(result) {alert("Data
           Loaded: " + JSON.parse(result));}, "json");
}
```

This function calls the jquery function post with the following parameters. Redirect.php is the php script that will redirect our HTTP POST to our HTTP interface. Data is the JSON object previously shown. The next parameter is the function that will be executed when we receive the reply to our request, in this case we will show the result. The last parameter is the type of the data. For more information about jquery and the method jquery.post, please refer to `http://jquery.com/`.

MarketAgent is a very simple agent, it receives a query from our webpage and sends a product

encapsulated in a JSON object. In the following code, the MarketAgent receives the JSON containing the query and transforms it into a Java object.

```
ACLMessage msg = receiveACLMessage();
XStream xstream = new XStream(new JettisonMappedXmlDriver());
xstream.alias("jsonObject", JSONMessage.class);
JSONMessage query = (JSONMessage)xstream.fromXML(msg.getContent());
```

Once the JSON object is a Java object it is possible to work easily with it. The MarketAgent does some calculations with the input data and prepares the response.

```
 XStream xstream2 = new XStream(new JsonHierarchicalStreamDriver() {
   @Override
   public HierarchicalStreamWriter createWriter(Writer writer) {
      return new JsonWriter(writer, JsonWriter.DROP_ROOT_MODE);
   }
});
String result = xstream2.toXML(product);
response.setContent(result);
this.send(response);
```

This time the MarketAgent transforms a Java object into a JSON object. As in the previous step it uses the library XStream. Finally we assign the JSON object as the content of the response message.

Before we can test this example it is important to make sure that the example.html file and the redirect.php script are both in the same web server. By security reasons, cross communication between different hosts is forbidden. Besides, we have to specify in redirect.php the address where our HTTP interface service is running. We can indicate the address modifying the following line in redirect.php:

```
$request = new HTTP_Request2('http://localhost:8081', HTTP_Request2
   ::METHOD_POST);
```

In this example the HTTP interface is running in the localhost.

CHAPTER 10

Advanced platform administration

10.1 Advanced 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)
- 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 10.1

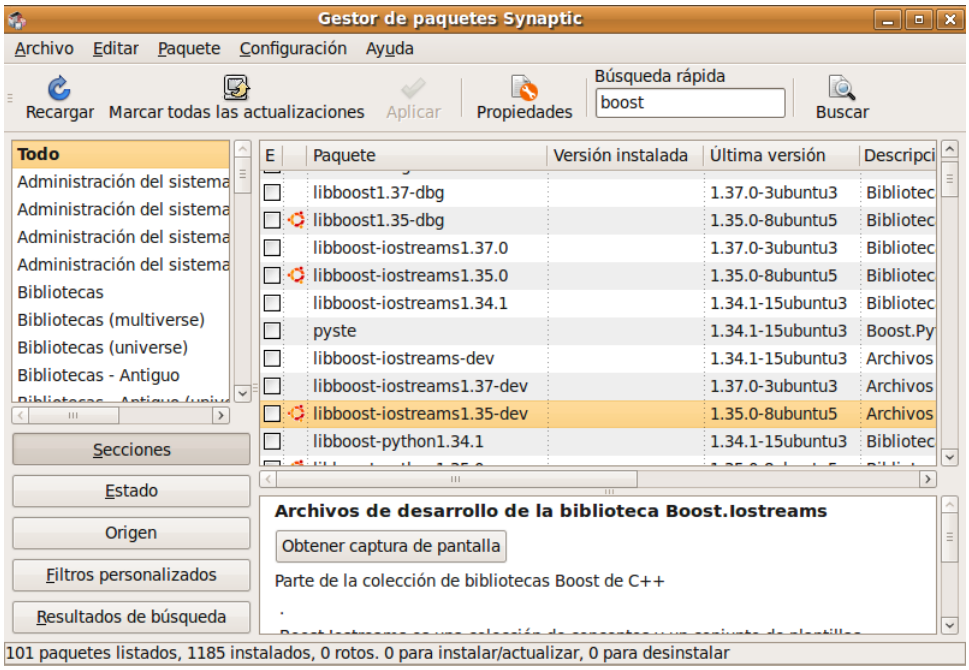

Figure 10.1: Installing libboost-iostreams 1.35-dev library with Synaptic tool

shows how to install the libboost-iostreams 1.35-dev library. Once all the required libraries have been installed, 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= /opt/qpid` → Using the prefix option when configuring, the location where the Qpid binaries are installed can be specified (in this example case, /opt/qpid).
- `$ 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, /opt/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.

10.2 Advanced MySQL

MySQL is a main component of the THOMAS framework. This section explains how to properly configure MySQL to work in conjunction with THOMAS. This section helps you 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 *magentix-setup.py*. This script is located in the main directory of the Magentix2 installation folder. The commands needed to execute this script are the following (from the Magentix2 root directory):

```
$ python magentix-setup.py
```

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 *db-schema.sql* and *grants.sql* files. These files are located into the directory `bin/sql/`. In order to load these files, the *MySQL Administrator* tool should be opened, and then the *Restore Backup* option must be selected, choosing the *db-schema.sql* backup file to be restored. An example of this procedure can be shown in the figure 10.2

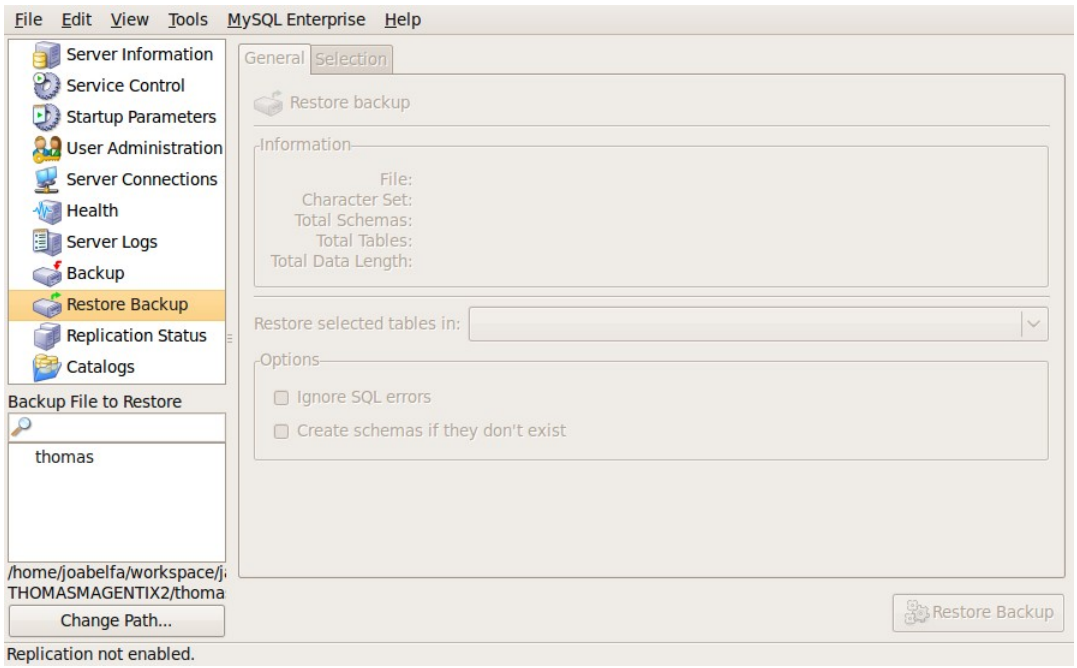

Figure 10.2: Restoring the *db-schema.sql* backup file in the *Restore Backup* option of the *MySQL Administrator*

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 10.3). The required fields must be fulfilled with the following information:

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

You can automate this step by loading the *grants.sql* file.

The *ServerName*, *databaseName*, *userName* and *password* entries must be also configured in the Settings.xml file located in the `configuration/` directory. The parameters that affect MySQL configuration in the Settings.xml file are the following:

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

On the other hand, the THOMAS framework uses Apache Jena for manage the semantic de-

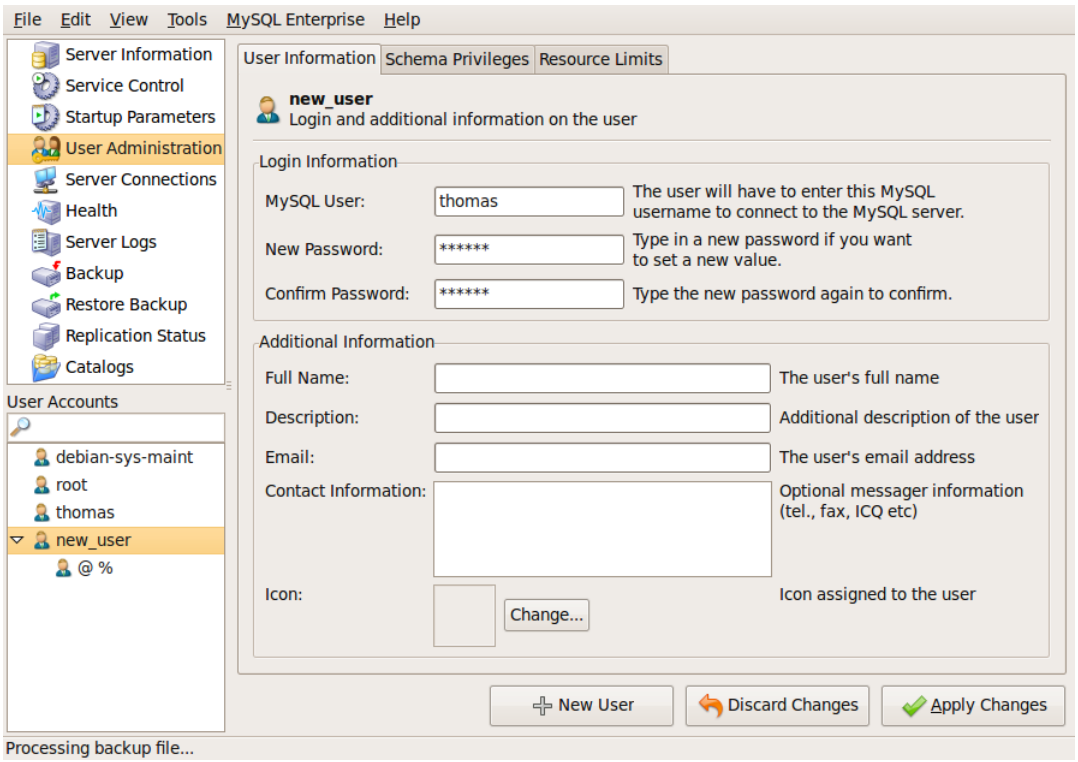

Figure 10.3: Adding the necessary user information into the THOMAS schema in the *User Administrator* option of the *MySQL Administrator* tool

scription of the services. In order to specify the required parameters for Jena, the Settings.xml file located in the `configuration/` directory will be configured. The parameters that affect Jena configuration in the Settings.xml file are the following:

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

Check if the direction where agents OMS and SF are running is different host from where the MySQL is configured with the data base schema employed by the THOMAS framework. If the OMS and SF are running in the same host, the configuration by default is correctly.

Finally, all available privileges for THOMAS tables must be assigned to the *thomas* user (Figure 10.4)

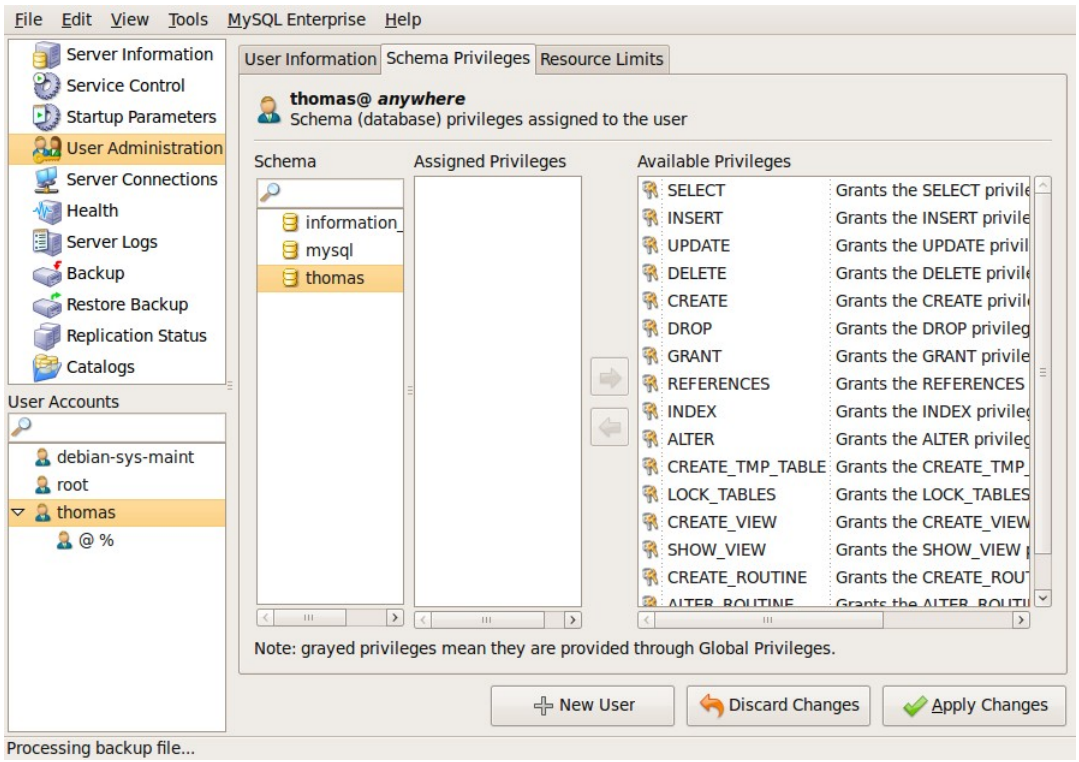

Figure 10.4: Assigning privileges to the *thomas* user in the *User Administration* option of the *MySQL Administration tool*

10.3 Advanced 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 other hand, this section helps to configure it properly.

THOMAS platform is based on services (chapter 8), so SF and OMS service implementations have to be available as standard web services. Moreover, Magentix2 offers another service named MMS (section **??**), 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 the `webapps/` directory of the Magentix2 installation to the subdirectory
`webapss/` of the Tomcat installation directory.

Then, the path where web services are deployed is required. In order to specify these parameters, the Settings.xml file located in the configuration/ directory will be configured The parameters that affect THOMAS configuration in the Settings.xml file are the following:

```
<!-- Properties thomas -->
<entry key="OMSServiceDescriptionLocation">
        http://localhost:8080/omsservices/services/
</entry>
<entry key="SFServiceDescriptionLocation">
        http://localhost:8080/sfservices/services/
</entry>
```

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.

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.

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.

10.4 Advanced 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.

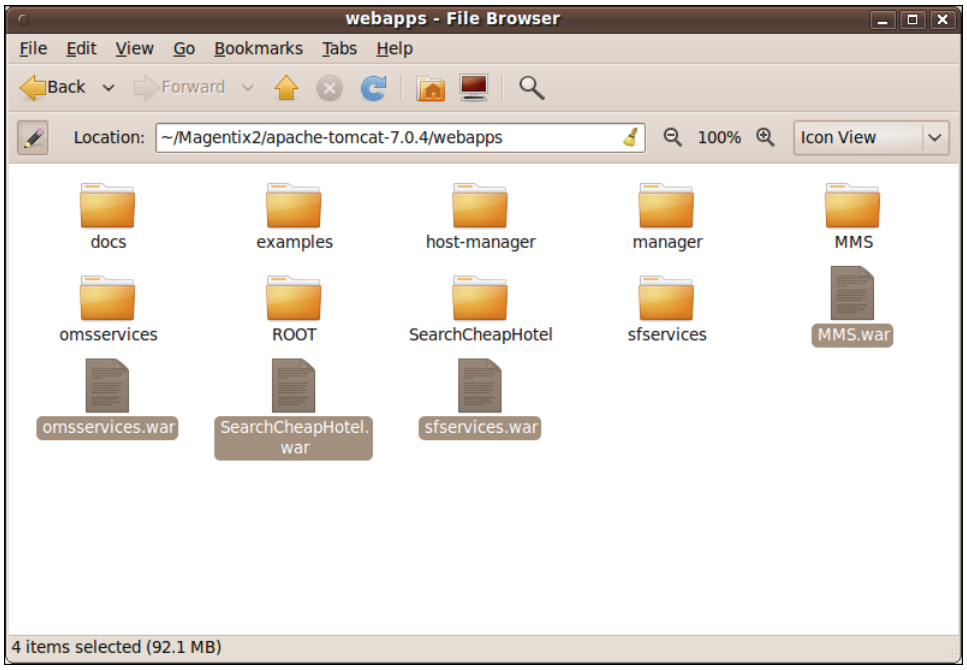

Figure 10.5: Location of web services files (*.war)

10.4.1 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* 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.

10.4.2 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.

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.

Heras, S. (2011). *Case-Based Argumentation Framework for Agent Societies*. PhD thesis, Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València. http://hdl.handle.net/10251/12497.

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.

Walton, D., Reed, C., and Macagno, F. (2008). *Argumentation Schemes*. Cambridge University Press.