2. MAGENTIX2 AGENTS

Creating a Magentix2 agent is as easy as defining a class extending the es.upv.dsic.gti_ia.core.BaseAgent, create a unique identifier (AgentID) for that agent can be referenced in our platform and implement the logic of the agent in the execute() method as illustrate in the next code. Let us to explain AgentID structure later.

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

public class SenderAgent extends BaseAgent {

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

        public void execute() {
                logger.info("Hello world");
        }
}
```

Note that although the example we use log4j, an output like "System.out.println("Hello World!")" also would be accepted.

The execute() method is used in order to define the logic of the agent. Therefore, in this first version of magentix2 platform the actual job that an agent has to do, its carried out by this method.

2.1 Agents identifiers

Each agent is identified by a AgentID tuple as follows: <Protocol>://<Agent-Name>@Host-Name>:<Port>, or more commonly, and only for internal agents: **<Agent-Name>@Host-Name>:<Port>**
Thus we allow agents on any server machine thrown all within the same platform.

The agents identifiers are represented as an instance of es.upv.dsic.gti_ia.core.AgentID, where the set of allowed protocols is {"http", "qpid"}, the Agent-Name is any name that is not already registered on the platform, Host identifies the server machine on which the agent will stay and finally it is important to specify the port on the server machine where the agent is going to launch.

For instance, an agent called "Consumer", living on a "qpid" platform, having been launched in "localhost" and whose port corresponds with "8080" has the form qpid://Consumer@localhost:8080. So this agent will have Consumer@localhost:8080 as globally unique name. Note that unlike other not distributed platforms, this address is quite important when another agent living on a different server machine needs to communicate with this agent.

It is not always necessary to specify the four fields of the AgentID tuple. Only Agent-Name field is always required, the rest take the following default values if they are not specified as follows:

Protocol = "qpid",
Host = "localhost",
Port = "8080".

Please note, we understand as internal agents, agents of our platform that can be launched on any machine, on any operating system and implemented in any programming language, but all of them must be working on the same qpid broker communications. The Qpid broker can be in the same or in a different machine. Also although it is recommended that the broker works on UNIX, you can also do in Windows XP. Let us to explain all the details of a agent connection in the next section.

The toString() method of the es.upv.dsic.gti_ia.core.AgentID allows retrieving the agent identifier as a string.

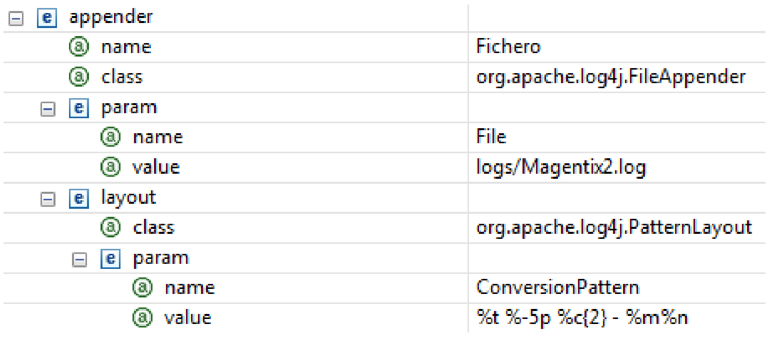
2.2 Initialization tasks

Our platform have been developed with log4j, for this reason is necessary it will be initialized as follows:

i. DOMConfigurator.configure("configuration/loggin.xml");
ii. Logger logger = Logger.getLogger(Run.class);

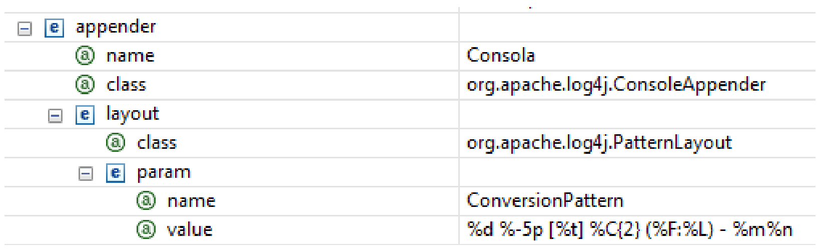
Regarding i. logging.xml is a file that you can use to specify what level of log messages are written to the log files for that component. Moreover, different appenders can be defined in logging.xml. Each one indicates a place where the platform writes its output messages. For instance, in the configuration file we provide two appenders have been defined.

Appender 1



e appender	
a name	Fichero
a class	org.apache.log4j.FileAppender
e param	
a name	File
a value	logs/Magentix2.log
e layout	
a class	org.apache.log4j.PatternLayout
e param	
a name	ConversionPattern
a value	%t %-5p %c{2} - %m%n

Appender 2



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

The first indicates a file and the second console as standard output. Learn more: http://logging.apache.org/log4j/1.2/index.html

Note: If you modify the logging.xml file associated with the platform, you must restart the platform before the changes take effect.
Respecting ii. logger represents a new instance logger for the class. And remember that this step is not required for classes that extend from es.upv.dsic.gti_ia.core.BaseAgent, because of an instance of Logger has been defined within it.

2.3 Connecting to Qpid Broker

Always before launch any agent, a connection to the Qpid broker must have been established. Thus any agent in our platform will use this communication.
Remember that at this point we assume that you have a Qpid broker launched on a machine and running properly.

The following parameters must be specified in any connection to the broker. <QpidHost>,<QpidPort>,<QpidVhost>,<QpdidUser>,<QpidPassword>, <QpidSSL>.

<QpidHost> indicates the machine where is living Qpid, <QpidPort> refers to the port where the broker works, <QpidVhost> is a path which acts as a namespace,<QpdidUser> and <QpidPassword> are the user credentials to access the broker and <QpidSSL> indicates if we use SSL encryption on the connection.
Note that if you require client authentication, the clients certificate needs to be signed by a CA trusted by the broker. Previously, the broker must have been installed with option authentication.

Although this is not important to develop agents, as a curiosity, internally Magentix2 defines a single connection as a url following the AMQP specification.

amqp://[<user>:<pass>@][<clientid>]<virtualhost>[?<option>='<value>'[&<option>='<value>']]

The connection url defines the values that are common across the cluster of brokers. The virtual host is second in the list as the AMQP specification demands that it start with a '/' otherwise it be more readable to be swapped with clientid. There is currently only one required option and that is the **brokerlist** option. In addition the following options are recognized.

For instance, Magentix2 could use a URL which looks something like this:

amqp://guest:guest@client1/development?brokerlist='tcp://localhost:5672'

Breaking this example down, here's what it all means:

- amqp = the protocol we're using
- guest:guest@localhost = username:password@clientid where the clientid is the name of your server (used under the covers but don't worry about this for now). Always use the guest:guest combination at the moment.
- development = the name of the virtualhost, where the virtualhost is a path which acts as a namespace. You can effectively use any value here so long as you're consistent throughout. The virtualhost must start with a slash "/" and continue with names separated by slashes. A name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_+!=:].
- brokerlist = this is the host address and port for the broker you want to connect to. The connection factory will assume tcp if you don't specify a transport protocol. The port also defaults to 5672. Naturally you have to put at least one broker in this list.

Note that for this first version of Magentix2, is not allow use failover so only provides one host for the broker. If you do wish to connect using failover you should wait for future releases.

Forgetting what internally makes Magentix2 , when the programmers want to launch agents, only first they must create a Qpid connection. All agents to be launched after will make use of this connection in an implicit way.

There are three different ways to establish a connection with Qpid broker implemented in es.upv.dsic.gti_ia.core.AgentsConnection:

- Take the input connection parameters from settings.xml file. Connect(). Let us to explain this file later.
- Take into account all the parameters specified as input. Connect(qpidHost, qpidPort, qpidVhost, qpdidUser, qpidPassword, qpidSSL)
- Take into account the qpidhost parameter and considering the rest as default parameters. Connect(qpidHost)

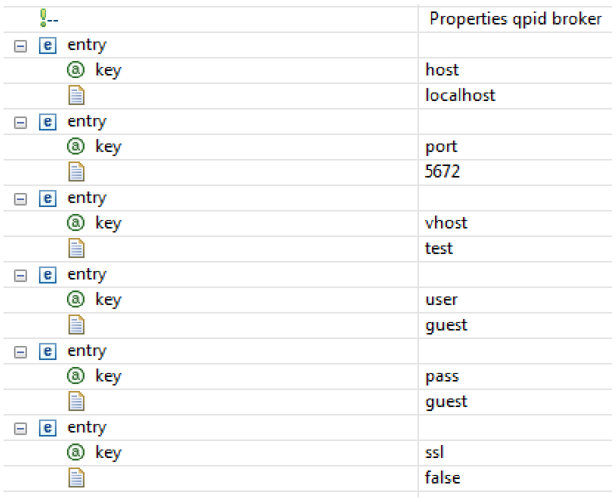
For instance:

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

Finally on Qpid connection, we will explain how to specify parameters for connection to the broker via a configuration file called Settings.xml. (Remember that there are only two configuration files, loggin.xml and Setting.xml).

Simply we can specify on Setting.xml file all the connection parameters, and so when we want to establish a connection to the Qpid broker, we can use the method Connect (). Note that if you do not specify all parameters in Setting.xml file, you must not use the Connect method.

For instance:



!--	Properties qpid broker
entry	
key	host
	localhost
entry	
key	port
	5672
entry	
key	vhost
	test
entry	
key	user
	guest
entry	
key	pass
	guest
entry	
key	ssl
	false

2.4 Running Agents

Once implemented agents, it can be instantiated and can be launched. Please note that the platform not allow several agents with the same name.

As noted on agents identifiers subsection, there are several ways to create an agent ID:

- AgentID(String Name, String Protocol, String Host, String Port)
- AgentID(String Identifier), where Identifier may be only the name of the new agent and assuming all other parameters by default, or Identifier can be a URL as qpid://<Agent-Name>@Host-Name>:<Port>.

In order to execute the agents start method must be called.

For instance, creating a new Instantiation:

i. SenderAgent agent1 = new SenderAgent(new AgentID("qpid://emisor@localhost:8080"));

ii. ConsumerAgent agent2 = new ConsumerAgent(new AgentID("consumer"));

For instance, running agents:

i. agent2.start();

ii. agent.start();

2.5 Optional Agent Initialization

Although init and finalize methods have not mentioned during the explanation, these methods can be overwritten when we extend the es.upv.dsic.gti_ia.core.BaseAgent.

The initialization corresponds to the init method. Here the programmer can define activities such as initialization resources, and every task necessary before execution of execute procedure.

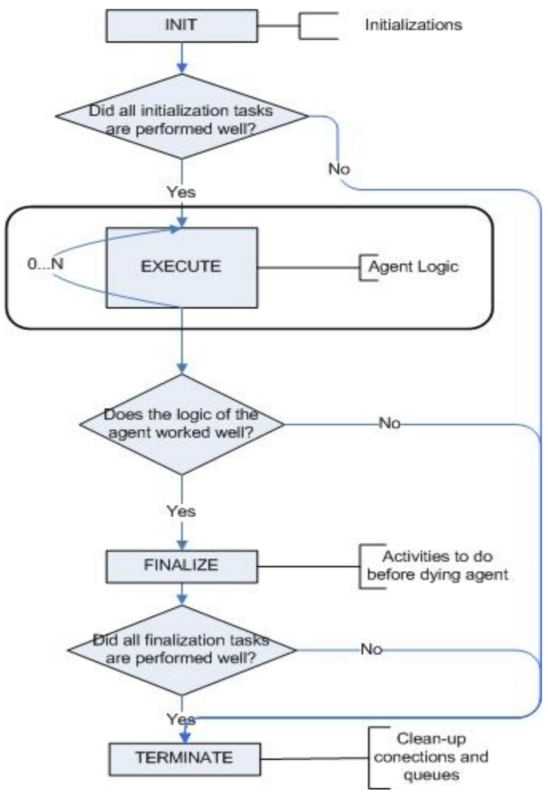
2.6 Optional Agent Termination

The termination is matched with the finalize method. Here the programmer can define activities that must be closed before killing an agent. For instance, before dying the agent it must inform to AMS Agent.

2.7 Agent Finalize

This method is already implemented by es.upv.dsic.gti_ia.core.BaseAgent. It is in charge of closing the connections that the agent has, moreover to delete its associated queue in the Qpid broker.

Summarizing the logic of an agent is reduced to:


INIT
Initializations
Did all initialization tasks are performed well?
No
Yes
0...N
EXECUTE
Agent Logic
Does the logic of the agent worked well?
No
Yes
FINALIZE
Activities to do before dying agent
Did all finalization tasks are performed well?
No
Yes
TERMINATE
Clean-up conections and queues
Agent Thread path of execution.

Attention is important to understand that these methods can be redefined by the programmer, but in no case should be invoked. This is because es.upv.dsic.gti_ia.core.BaseAgent defines run method, which is responsible for invoking in sequence each of these methods. BaseAgent implements Runnable class.

2.7 Other Considerations

Despite that throughout this section, we have always been referring to class es.upv.dsic.gti_ia.core.BaseAgent, we must not forget that BaseAgent is just a meta-template of agent. That is, different templates of agents have been tried, all have in common that they extend from BaseAgent.

For example:

- es.upv.dsic.gti_ia.architecture.QueueAgent defines an new agents template, extending of BaseAgent. This type of agent's implements a set of protocols.
- es.upv.dsic.gti_ia.core.SingleAgent defines an new agent template, extending BaseAgent. It defines a new reception method that takes into account a blocking reception.
- es.upv.dsic.gti_ia.core.BridgeAgentInOut works as a mediator agent receives all messages whose recipient is another platform (protocol = http), encapsulates the entire message and sends the message via http.

- es.upv.dsic.gti_ia.core.BridgeAgentOutIn works in contrast to BridgeAgentInOut. Their work is expected messages to internal agents which have been sent by external agents via http. Therefore the objective is the decapsulation of a http message to make a ACLMessage message. After this message will be sent to the recipient's mailbox.

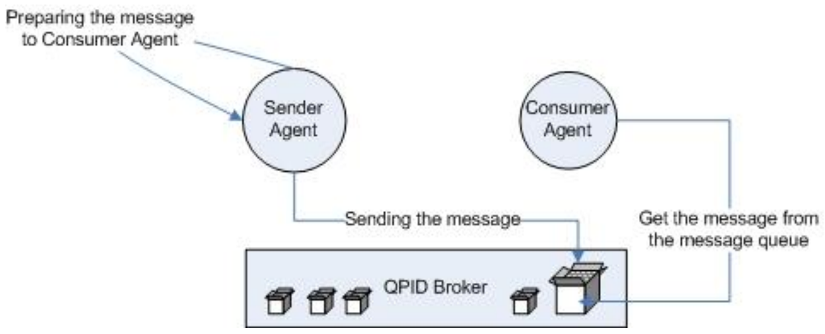
3. AGENT COMMUNICATION

As with any other agents platform, the ability to communicate between agents not only is an important task, is a crucial task that in most cases can make the success of a platform of intelligent agents.

In Magentix2 we can ensure at least among internal agents, communication is really successful. It is incredibly fast and can be very safe. Although we do not give more details in this manual, various performance tests have been widely tested in detail, and the results are really successful.

Each agent has a sort of mailbox on Qpid Broker, where other agents can post messages.

It is important to say that in this first version of Magentix2 does not support to organizations, therefore all the mailboxes of the agents are associated with a single exchange of Qpid Broker called amq.direct


Preparing the message to Consumer Agent
Sender Agent
Consumer Agent
Sending the message
Get the message from the message queue
QPID Broker

3.1 FIPA ACL Language

Messages exchanged by Magentix2 agents have a format specified by the ACL language defined by the FIPA (http://www.fipa.org) international standard for agent interoperability. This format comprises a number of fields as:

- The **sender** of the message.
- The list of **receivers**
- The communicative intention (also called “**performative**”) indicating what the sender intends to achieve by sending the message. The performative can be REQUEST, if the sender wants the receiver to perform an action, INFORM, if the sender wants the receiver to be aware a fact, QUERY_IF, if the sender wants to know whether or not a given condition holds, CFP (call for proposal), PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL, if the sender and receiver are engaged in a

negotiation. Although we will not explain the rest of the constants that can identify the FIPA performative, all of them have been defined.

- The **content** is the actual information included in the message
- The **language** as the syntax used to express the content.
- The **ontology** as the vocabulary of the symbols used in the content and their meaning.
- Some fields used to control several concurrent conversations and to specify timeouts for receiving a reply such as **conversation-id**, **reply-with**, **in-reply-to**, **reply-by**.

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

3.2 Sending message

Sending a message to another agent is as simple as filling 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 next message "Hello I'm Consumer".

```
/**
 * Building a ACLMessage
 */
ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
msg.setReceiver(receiver);
msg.setSender(this.getAid());
msg.setLanguage("ACL");
msg.setContent("Hello, I'm " + getName());

/**
 * Sending a ACLMessage
 */
        send(msg);
```

Let us to explain how the send(ACLMessage msg) method from BaseAgent works. First, it is important to know that any message to be sent, must have been coded as an instance of es.upv.dsic.gti_ia.core.ACLMessage class. The task for this send method is to construct an instance of class org.apache.qpid.transport.MessageTransfer. Although we will not give many details of MessageTransfer, it message is formed by Destination, AcceptMode, AcquireMode, Header, BodyString.

A message will be sent to as many recipients as you have the message.

Note that BaseAgent class has a method called MessageTransfertoACLMessage, which is capable of converting a

MessageTransferto to ACLMessage. Remember that all this work has been necessary to integrate in our platform Qpid broker. But in any case all this is internal to our platform and is not accessible by programmers.

3.2 Receiving message

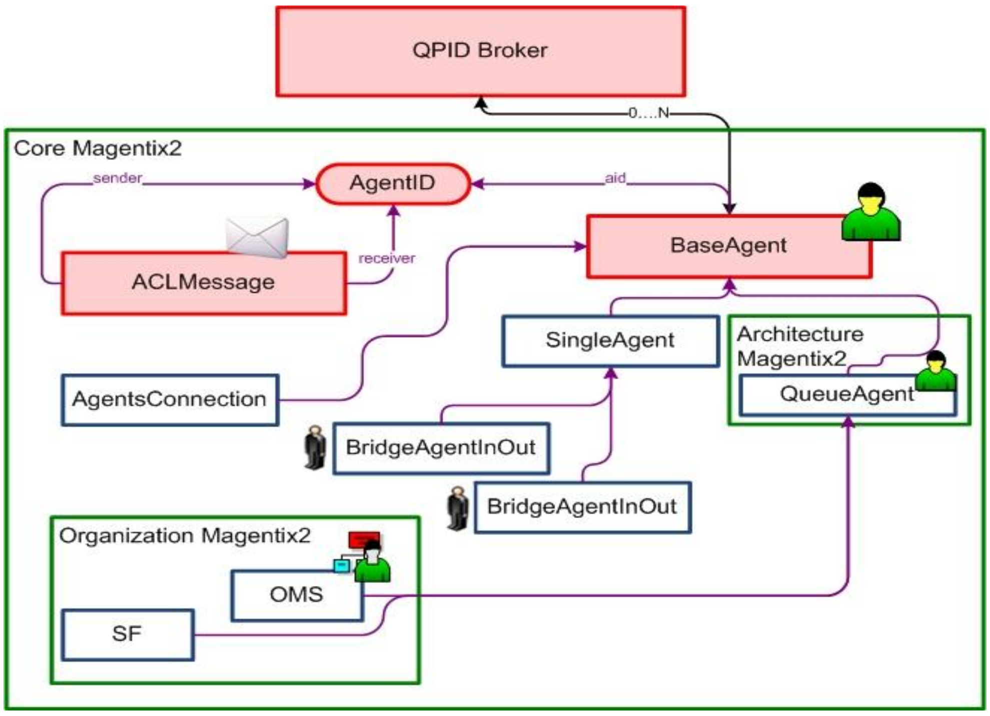
Whenever a message is posted in the message queue the receiving agent is notified by onMessage(ACLMessage msg) method. This method allows it to receive any message instantly. Note that the agent could keep all the notifications received in turn in another internal list, and read it at another time.

It is therefore important to redefine the method onMessage when we are implementing a new agent, so that when it receive a message, it can be read. Let us to consider an example:

```
public void onMessage(ACLMessage msg) {
                /**
                 * When a message arrives, its shows on screen
                 */
        logger.info("Mensaje received in " + this.getName()
                                + " agent, by onMessage: " + msg.getContent());
        }
```

4. FULL EXAMPLE

Before considering a full example let us summarize the main components of Magentix2 by the following figure, as we illustrate...... falta.....


QPID Broker
0....N
Core Magentix2
sender
AgentID
aid
receiver
ACLMessage
BaseAgent
SingleAgent
Architecture Magentix2
QueueAgent
AgentsConnection
BridgeAgentInOut
BridgeAgentInOut
Organization Magentix2
OMS
SF