JADE TUTORIAL

JADE PROGRAMMING FOR BEGINNERS

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 04 December 2003. JADE 3.1

Authors: Giovanni Caire (TILAB, formerly CSELT)

TABLE OF CONTENTS

JADE PROGRAMMING FOR BEGINNERS

This tutorial shows how to create simple JADE agents and how to make them executing tasks and communicate between each other. JADE is completely written in Java and JADE programmers work in full Java when developing their agents. Therefore the reader is assumed to be familiar with the Java programming language.

This tutorial is structured as follows. Chapter 1 gives a brief overview of JADE and introduces the concepts of Platform, Container, AMS and DF. Chapter 2 presents a simple example that will be used throughout this tutorial to illustrate the steps required to develop simple agent-based applications with JADE. Chapter 3 focuses on creating agents and explains the basic features of the `Agent` and `AID` classes. Chapter 4explains how to make JADE agents execute tasks and introduces the `Behaviour` class. Chapter 5 describes how to make agents communicate and introduces the `ACLMessage` and `MessageTemplate` classes. Chapter 6 illustrates how to exploit the Yellow Pages service provided by the DF agent through the `DFService` class.

Besides the basic features illustrated in this tutorial JADE provides a number of advanced features such as the support for complex interaction protocols and the usage of user defined ontologies. For these features readers are reminded to the JADE Programmer's guide and Administrator's guide available on the JADE web site.

1 JADE OVERVIEW

JADE is a middleware that facilitates the development of multi-agent systems. It includes

- A **runtime environment** where JADE agents can "live" and that must be active on a given host before one or more agents can be executed on that host.
- A **library** of classes that programmers have to/can use (directly or by specializing them) to develop their agents.
- A suite of **graphical tools** that allows administrating and monitoring the activity of running agents.

1.1 Containers and Platforms

Each running instance of the JADE runtime environment is called a ***Container*** as it can contain several agents. The set of active containers is called a ***Platform***. A single special ***Main container*** must always be active in a platform and all other containers register with it as soon as they start. It follows that the first container to start in a platform must be a main container while all other containers must be "normal" (i.e. non-main) containers and must "be told" where to find (host and port) their main container (i.e. the main container to register with).

If another main container is started somewhere in the network it constitutes a different platform to which new normal containers can possibly register. Figure 1 illustrates the above concepts through a sample scenario showing two JADE platforms composed of 3 and 1 container respectively. JADE agents are identified by a unique name and, provided they know each other's name, they can communicate transparently regardless of their actual location: same container (e.g. agents A2 and A3 in Figure 1), different containers in the same platform (e.g. A1 and A2) or different platforms (e.g. A4 and A5).

You don't have to know how the JADE runtime environment works, but just need to start it before executing your agents. **Starting JADE as a main or normal container and executing agents on it, is**

described in the JADE Administrative Tutorial available on the JADE website.

1.2 AMS and DF

Besides the ability of accepting registrations from other containers, a main container differs from normal containers as it holds two special agents (automatically started when the main container is launched).

The **AMS** (Agent Management System) that provides the naming service (i.e. ensures that each agent in the platform has a unique name) and represents the authority in the platform (for instance it is possible to create/kill agents on remote containers by requesting that to the AMS). This tutorial does not illustrate how to interact with the AMS as this is part of the advanced JADE programming.

The **DF** (Directory Facilitator) that provides a Yellow Pages service by means of which an agent can find other agents providing the services he requires in order to achieve his goals. Using the Yellow Pages service provided by the DF agent is illustrated in chapter 6.

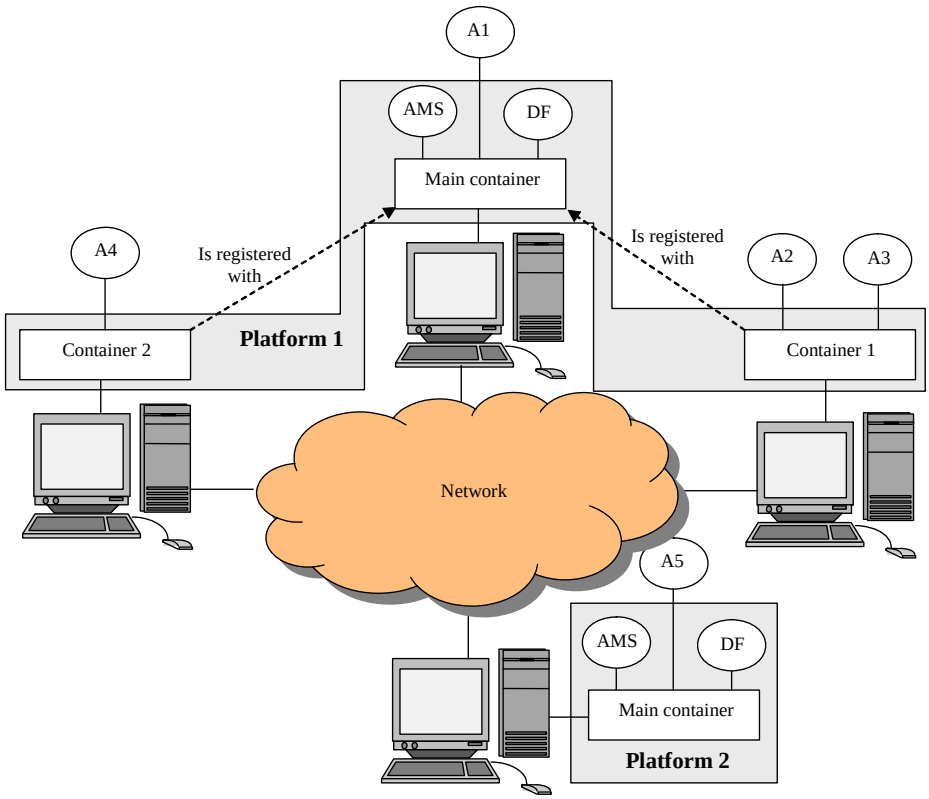

Figure 1Containers and Platforms

2 THE "BOOK TRADING" EXAMPLE

This chapter introduces a simple example that will be used throughout this tutorial to illustrate the steps required to develop agent-based applications with JADE. The scenario considered in this example includes some agents selling books and other agents buying books on behalf of their users.

Each buyer agent receives the title of the book to buy (the “target book”) as a command line argument and periodically requests all known seller agents to provide an offer. As soon as an offer is received the buyer agent accepts it and issues a purchase order. If more than one seller agent provides an offer the buyer agent accepts the best one (lowest price). Having bought the target book the buyer agent terminates.

Each seller agent has a minimal GUI by means of which the user can insert new titles (and the associated price) in the local catalogue of books for sale. Seller agents continuously wait for requests from buyer agents. When asked to provide an offer for a book they check if the requested book is in their catalogue and in this case reply with the price. Otherwise they refuse. When they receive a purchase order they serve it and remove the requested book from their catalogue.

All issues related to electronic payment are outside the scope of this tutorial and are not taken into account.

The complete sources of this example are available among the examples provided with JADE in the `examples.bookTrading` package.

3 CREATING JADE AGENTS – THE AGENT CLASS

Creating a JADE agent is as simple as defining a class extending the `jade.core.Agent` class and implementing the `setup()` method as shown in the code below.

```
import jade.core.Agent;

public class BookBuyerAgent extends Agent {
  protected void setup() {
    // Printout a welcome message
    System.out.println(“Hallo! Buyer-agent “+getAID().getName()+” is ready.”);
  }
}
```

The `setup()` method is intended to include agent initializations. The actual job an agent has to do is typically carried out within “behaviours” as will be described in chapter 4.

3.1 Agent identifiers

Each agent is identified by an “agent identifier” represented as an instance of the `jade.core.AID` class. The `getAID()` method of the `Agent` class allows retrieving the agent identifier. An `AID` object includes a globally unique name plus a number of addresses. The name in JADE has the form *<nickname>@<platform-name>* so that an agent called *Peter* living on a platform called *P1* will have *Peter@P1* as globally unique name. The addresses included in the AID are the addresses of the platform the agent lives in. These addresses are only used when an agent needs to communicate with another agent living on a different platform. Developers need to care about them only in particular cases that are outside the scope of this tutorial.

Knowing the nickname of an agent, its AID can be obtained as follows:

```
String nickname = “Peter”;
AID id = new AID(nickname, AID.ISLOCALNAME);
```

The `ISLOCALNAME` constant indicates that the first parameter represents the nickname (local to the platform) and not the globally unique name of the agent.

3.2 Running agents

The created agent can be compiled as follows.

```
javac –classpath <JADE-classes> BookBuyerAgent.java
```

In order to execute the compiled agent the JADE runtime must be started and a nickname for the agent to run must be chosen:

```
java –classpath <JADE-classes>;. jade.Boot buyer:BookBuyerAgent
```

More details on compiling and running agents can be found in the JADE Administrative Tutorial or in the JADE Administrator's Guide available on the JADE website. The result of the typed command is as follows.

```
C:\jade>java –classpath <JADE-classes> jade.Boot buyer:BookBuyerAgent
    This is JADE snapshot - 2003/10/24 13:43:39
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/

IOR:000000000000001149444C3A464950412F4D54533A312E3000000000000000001000000000000
0060000102000000000A3132372E302E302E310009A600000019AFABCB0000000002BCE5528F0000
0008000000000000000000A000000000000001000000010000002000000000000010001000000020501
000100010020000101090000000100010100
Agent container Main-Container@JADE-IMTP://IBM8695 is ready.
Hallo! Buyer-agent buyer@IBM8695:1099/JADE is ready.
```

The first part of the above output is the JADE disclaimer that is printed out each time the JADE runtime is started. The IIOP address (in the form of an IOR) of the newly started platform follows. Finally the indication that a container called "Main-Container" is ready completes the JADE runtime startup. When the JADE runtime is up our agent is started and prints its welcome message. The nickname of the agent is "buyer" as we specified on the command line. The platform name "IBM8695:1099/JADE" is automatically assigned on the basis of the host and port you are running JADE on (see the JADE Administrator's guide for assigning a name to the platform).

3.3 Agent termination

Even if it does not have anything else to do after printing the welcome message, our agent is still running. In order to make it terminate its `doDelete()` method must be called. Similarly to the `setup()` method that is invoked by the JADE runtime as soon as an agent starts and is intended to include agent initializations, the `takeDown()` method is invoked just before an agent terminates and is intended to include agent clean-up operations.

3.4 Passing arguments to an agent

Agents may get start-up arguments specified on the command line. These arguments can be retrieved, as an array of `Object`, by means of the `getArguments()` method of the `Agent` class. As mentioned in chapter 2, we want our `BookBuyerAgent` to get the title of the book to buy as a command line argument. To achieve that we modify it as follows[1].

```
import jade.core.Agent;
import jade.core.AID;

public class BookBuyerAgent extends Agent {
  // The title of the book to buy
```

[1] Note that the list of known seller agents to send requests to is fixed. In chapter 6 we will describe how to dynamically discover them.

```
  private String targetBookTitle;
  // The list of known seller agents
  private AID[] sellerAgents = {new AID("seller1", AID.ISLOCALNAME),
                                new AID("seller2", AID.ISLOCALNAME)};

  // Put agent initializations here
  protected void setup() {
    // Printout a welcome message
    System.out.println("Hallo! Buyer-agent "+getAID().getName()+" is ready.");

    // Get the title of the book to buy as a start-up argument
    Object[] args = getArguments();
    if (args != null && args.length > 0) {
      targetBookTitle = (String) args[0];
      System.out.println("Trying to buy "+targetBookTitle);
    }
    else {
      // Make the agent terminate immediately
      System.out.println("No book title specified");
      doDelete();
    }
  }

  // Put agent clean-up operations here
  protected void takeDown() {
    // Printout a dismissal message
    System.out.println("Buyer-agent "+getAID().getName()+" terminating.");
  }
}
```

Arguments on the command line are specified included in parenthesis and separated by spaces[2].

```
C:\jade>java jade.Boot buyer:BookBuyerAgent(The-Lord-of-the-rings)
    This is JADE snapshot - 2003/10/24 13:43:39
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/

IOR:000000000000001149444C3A464950412F4D54533A312E3000000000000000001000000000000
006000010200000000A3132372E302E302E310009A600000019AFABCB0000000002BCE5528F0000
000800000000000000000A000000000000001000000010000002000000000000100010000000020501
000100010020000101090000000100010100
Agent container Main-Container@JADE-IMTP://IBM8695 is ready.
Hallo! Buyer-agent buyer@IBM8695:1099/JADE is ready.
Trying to buy The-Lord-of-the-rings
```

4 AGENT TASKS – THE BEHAVIOUR CLASS

As mentioned in chapter 3, the actual job an agent has to do is typically carried out within "behaviours". A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends `jade.core.behaviours.Behaviour`. In order to make an agent execute the task implemented by a behaviour object it is sufficient to add the behaviour to the agent by means of the `addBehaviour()` method of the `Agent` class. Behaviours can be added at any time: when an agent starts (in the `setup()` method) or from within other behaviours.

Each class extending `Behaviour` must implement the `action()` method, that actually defines the operations to be performed when the behaviour is in execution and the `done()` method (returns a `boolean`

[2] When using JADE with the LEAP add-on, arguments are separated by colon (';') instead of spaces.

value), that specifies whether or not a behaviour has completed and have to be removed from the pool of behaviours an agent is carrying out.

4.1 Behaviours scheduling and execution

An agent can execute several behaviours concurrently. However it is important to notice that scheduling of behaviours in an agent is not pre-emptive (as for Java threads) but cooperative. **This means that when a behaviour is scheduled for execution its `action()` method is called and runs until it returns.** Therefore it is the programmer who defines when an agent switches from the execution of a behaviour to the execution of the next one.

Though requiring a small additional effort to programmers, this approach has several advantages.

- Allows having a single Java thread per agent (that is quite important especially in environments with limited resources such as cell phones).
- Provides better performances since behaviour switch is extremely faster than Java thread switch.
- Eliminates all synchronization issues between concurrent behaviours accessing the same resources (this speed-up performances too) since all behaviours are executed by the same Java thread.
- When a behaviour switch occurs the status of an agent does not include any stack information and is therefore possible to take a "snapshot" of it. This makes it possible to implement important advanced features e.g. saving the status of an agent on a persistent storage for later resumption (agent persistency) or transferring it to another container for remote execution (agent mobility). Persistency and mobility are advanced JADE features and are outside the scope of this tutorial however.

The path of execution of the agent thread[3] is depicted in Figure 2.

[3] In JADE there is a single Java thread per agent. Since JADE agents are written in Java, however, programmers may start new Java threads at any time if they need. If you do that, remember to pay attention since the advantages mentioned in this section are no longer valid.

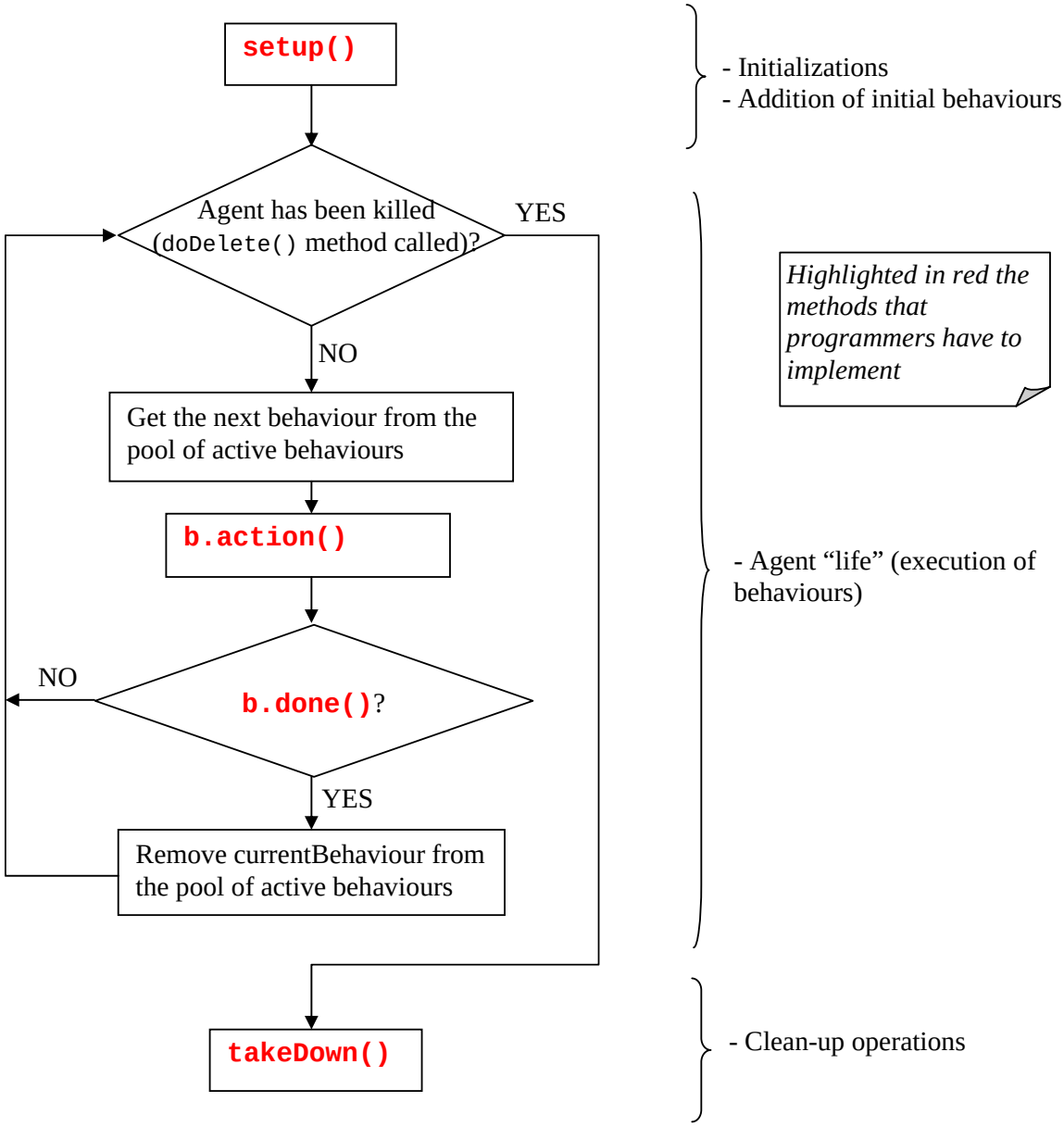

Figure 2. Agent Thread path of execution

Taking into account the described scheduling mechanism it is important to stress that a behaviour like that reported below prevents any other behaviour to be executed since its `action()` method never returns.

```
public class OverbearingBehaviour extends Behaviour {
  public void action() {
    while (true) {
      // do something
    }
  }

  public boolean done() {
    return true;
  }
}
```

When there are no behaviours available for execution the agent's thread goes to sleep in order not to consume CPU time. It is waken up as soon as there is again a behaviour available for execution.

4.2 One-shot behaviours, cyclic behaviours and generic behaviours

We can distinguish among three types of behaviour.

1) "One-shot" behaviours that complete immediately and whose `action()` method is executed only once. The `jade.core.behaviours.OneShotBehaviour` already implements the `done()` method by returning `true` and can be conveniently extended to implement one-shot behaviours.

```
public class MyOneShotBehaviour extends OneShotBehaviour {
  public void action() {
    // perform operation X
  }
}
```

Operation X is performed only once.

2) "Cyclic" behaviours that never complete and whose `action()` method executes the same operations each time it is called. The `jade.core.behaviours.CyclicBehaviour` already implements the `done()` method by returning `false` and can be conveniently extended to implement cyclic behaviours.

```
public class MyCyclicBehaviour extends CyclicBehaviour {
  public void action() {
    // perform operation Y
  }
}
```

Operation Y is performed repetitively forever (until the agent carrying out the above behaviour terminates).

3) Generic behaviours that embeds a status and execute different operations depending on that status. They complete when a given condition is met.

```
public class MyThreeStepBehaviour extends Behaviour {
  private int step = 0;
  public void action() {
    switch (step) {
    case 0:
      // perform operation X
      step++;
      break;
    case 1:
      // perform operation Y
      step++;
      break;
    case 2:
      // perform operation Z
      step++;
      break;
    }
  }

  public boolean done() {
    return step == 3;
  }
}
```

Operations X, Y and Z are performed one after the other and then the behaviour completes.

JADE provides the possibility of combining simple behaviours together to create complex behaviours. This feature is outside the scope of this document however. Refer to the Javadoc of the `SequentialBehaviour`, `ParallelBehaviour` and `FSMBehaviour` for the details.

4.3 Scheduling operations at given points in time

JADE provides two ready-made classes (in the `jade.core.behaviours` package) by means of which it is possible to easily implement behaviours that execute certain operations at given points in time.

1) The `WakerBehaviour` whose `action()` and `done()` methods are already implemented in such a way to execute the `handleElapsedTimeout()` abstract method after a given timeout (specified in the constructor) expires. After the execution of the `handleElapsedTimeout()` method the behaviour completes.

```
public class MyAgent extends Agent {
  protected void setup() {
    System.out.println("Adding waker behaviour");
    addBehaviour(new WakerBehaviour(this, 10000) {
      protected void handleElapsedTimeout() {
        // perform operation X
      }
    } );
  }
}
```

Operation X is performed 10 seconds after the "Adding waker behaviour" printout appears.

2) The `TickerBehaviour` whose `action()` and `done()` methods are already implemented in such a way to execute the `onTick()` abstract method repetitively waiting a given period (specified in the constructor) after each execution. A `TickerBehaviour` never completes.

```
public class MyAgent extends Agent {
  protected void setup() {
    addBehaviour(new TickerBehaviour(this, 10000) {
      protected void onTick() {
        // perform operation Y
      }
    } );
  }
}
```

Operation Y is performed periodically every 10 seconds.

4.4 Behaviours required in the book trading example

Having described the basic types of behaviour, let's move now to analyse which behaviours have to be carried out by the Book-buyer agent and Book-seller agent of our book trading example.

4.4.1 Book-buyer agent behaviours

As described in chapter 2, a Book-buyer agent periodically requests seller agents the book it was instructed to buy. We can easily achieve that by using a `TickerBehaviour` that, on each tick, adds another behaviour that actually deals with the request to seller agents. Here is how the `setup()` method of our `BookBuyerAgent` class can be modified.

```
protected void setup() {
  // Printout a welcome message
  System.out.println("Hallo! Buyer-agent "+getAID().getName()+" is ready.");

  // Get the title of the book to buy as a start-up argument
  Object[] args = getArguments();
  if (args != null && args.length > 0) {
    targetBookTitle = (String) args[0];
    System.out.println("Trying to buy "+targetBookTitle);

    // Add a TickerBehaviour that schedules a request to seller agents every minute
```

```
    addBehaviour(new TickerBehaviour(this, 60000) {
      protected void onTick() {
        myAgent.addBehaviour(new RequestPerformer());
      }
    } );
  }
  else {
    // Make the agent terminate
    System.out.println("No target book title specified");
    doDelete();
  }
}
```

Note the use of the `myAgent` protected variable: each behavior has a pointer to the agent that is executing it.

The `RequestPerformer` behaviour actually dealing with the request to seller agents will be described in chapter 5 where we will discuss agent communication.

4.4.2 Book-seller agent behaviours

As described in chapter 2, each Book-seller agent waits for requests from buyer agents and serves them. These requests can be requests to provide an offer for a book or purchase orders. A possible design to achieve that is to make a Book-seller agent execute two cyclic behaviours: one dedicated to serve requests for offer and the other dedicated to serve purchase orders. How actually incoming requests from buyer agents are detected and served is described in chapter 5 where we will discuss agent communication. Moreover we need to make the Book-seller agent execute a one-shot behaviour updating the catalogue of books available for sale whenever the user adds a new book from the GUI. Here is how the `BookSellerAgent` class can be implemented (the `OfferRequestsServer` and `PurchaseOrdersServer` classes will be presented in chapter 5).

```
import jade.core.Agent;
import jade.core.behaviours.*;

import java.util.*;

public class BookSellerAgent extends Agent {
  // The catalogue of books for sale (maps the title of a book to its price)
  private Hashtable catalogue;
  // The GUI by means of which the user can add books in the catalogue
  private BookSellerGui myGui;

  // Put agent initializations here
  protected void setup() {
    // Create the catalogue
    catalogue = new Hashtable();

    // Create and show the GUI
    myGui = new BookSellerGui(this);
    myGui.show();

    // Add the behaviour serving requests for offer from buyer agents
    addBehaviour(new OfferRequestsServer());

    // Add the behaviour serving purchase orders from buyer agents
    addBehaviour(new PurchaseOrdersServer());
  }

  // Put agent clean-up operations here
  protected void takeDown() {
```

```
    // Close the GUI
    myGui.dispose();
    // Printout a dismissal message
    System.out.println("Seller-agent "+getAID().getName()+" terminating.");
  }

  /**
     This is invoked by the GUI when the user adds a new book for sale
   */
  public void updateCatalogue(final String title, final int price) {
    addBehaviour(new OneShotBehaviour() {
      public void action() {
        catalogue.put(title, new Integer(price));
      }
    } );
  }
}
```

The `BookSellerGui` class is a simple Swing GUI and is not presented here since it is outside the scope of this tutorial. Its code is available among the sources packaged with this tutorial.

5 AGENT COMMUNICATION – THE ACLMESSAGE CLASS

One of the most important features that JADE agents provide is the ability to communicate. The communication paradigm adopted is the **asynchronous message passing**. Each agent has a sort of mailbox (the agent message queue) where the JADE runtime posts messages sent by other agents. Whenever a message is posted in the message queue the receiving agent is notified. If and when the agent actually picks up the message from the message queue to process it is completely up to the programmer however.

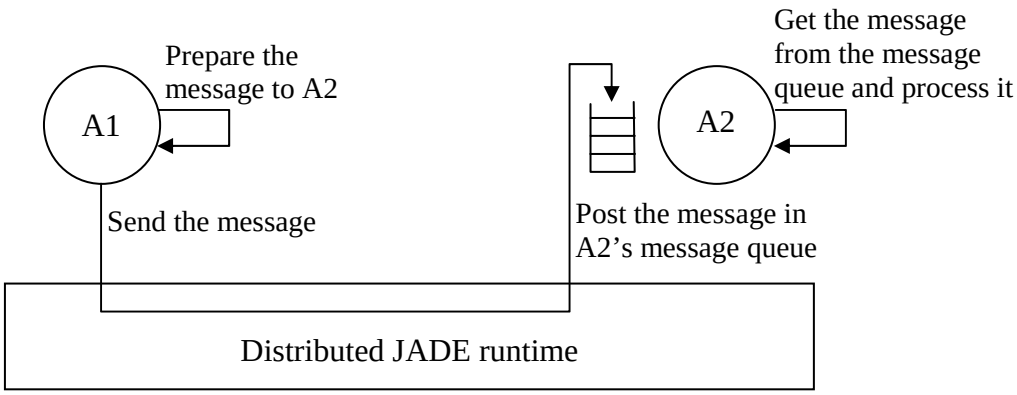

Figure 3. The JADE asynchronous message passing paradigm

5.1 The ACL language

Messages exchanged by JADE 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 and in particular:

- 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, and more.

- The ***content*** i.e. the actual information included in the message (i.e. the action to be performed in a REQUEST message, the fact that the sender wants to disclose in an INFORM message …).

- The content ***language*** i.e. the syntax used to express the content (both the sender and the receiver must be able to encode/parse expressions compliant to this syntax for the communication to be effective).

- The ***ontology*** i.e. the vocabulary of the symbols used in the content and their meaning (both the sender and the receiver must ascribe the same meaning to symbols for the communication to be effective).

- 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 JADE is implemented as an object of the `jade.lang.acl.ACLMessage` class that provides `get` and `set` methods for handling all fields of a message.

5.2 Sending messages

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 `Agent` class. The code below informs an agent whose nickname is *Peter* that *today it's raining*.

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-forecast-ontology");
msg.setContent("Today it's raining");
send(msg);
```

5.3 The book trading example messages

Considering our book trading example we can conveniently use the `CFP` (call for proposal) performative for messages that Buyer-agents send to Seller-agents to request an offer for a book. The `PROPOSE` performative can be used for messages carrying seller offers, and the `ACCEPT_PROPOSAL` performative for messages carrying offer acceptance, i.e. purchase orders. Finally the `REFUSE` performative will be used for messages sent by seller agents when the requested book is not in their catalogue. In both types of messages sent by buyer agents we assume that the message content is the title of the book. The content of `PROPOSE` messages will be the price of the book. As an example, here is how a `CFP` message can be created and sent.

```
// Message carrying a request for offer
ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.lenght; ++i) {
  cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
myAgent.send(cfp);
```

5.4 Receiving messages

As mentioned above the JADE runtime automatically posts messages in the receiver's private message queue as soon as they arrive. An agent can pick up messages from its message queue by means of the `receive()` method. This method returns the first message in the message queue (removing it) or `null` if the message queue is empty and immediately returns.

```
ACLMessage msg = receive();
if (msg != null) {
  // Process the message
}
```

5.5 Blocking a behaviour waiting for a message

Very often programmers need to implement behaviours that process messages received by other agents. This is the case for the `OfferRequestsServer` and `PurchaseOrdersServer` behaviours referenced in 4.4.2 where we need to serve messages from buyer agents carrying requests for offer and purchase orders. Such behaviour must be continuously running (cyclic behaviours) and, at each execution of their `action()` method, must check if a message has been received and process it. The two behaviours are very similar. Here we present the `OfferRequestsServer` behaviour. Look at the sources available among the JADE examples for the code of the `PurchaseOrdersServer`.

```
/**
   Inner class OfferRequestsServer.
   This is the behaviour used by Book-seller agents to serve incoming requests
   for offer from buyer agents.
   If the requested book is in the local catalogue the seller agent replies
   with a PROPOSE message specifying the price. Otherwise a REFUSE message is
   sent back.
 */
private class OfferRequestsServer extends CyclicBehaviour {
  public void action() {
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
      // Message received. Process it
      String title = msg.getContent();
      ACLMessage reply = msg.createReply();

      Integer price = (Integer) catalogue.get(title);
      if (price != null) {
        // The requested book is available for sale. Reply with the price
        reply.setPerformative(ACLMessage.PROPOSE);
        reply.setContent(String.valueOf(price.intValue()));
      }
      else {
        // The requested book is NOT available for sale.
        reply.setPerformative(ACLMessage.REFUSE);
        reply.setContent("not-available");
      }
      myAgent.send(reply);
    }
  }
}  // End of inner class OfferRequestsServer
```

We decided to implement the `OfferRequestsServer` behaviour as an inner class of the `BookSellerAgent` class. This simplifies things as we can directly access the catalogue of books for sale; it is not mandatory however.

The `createReply()` method of the `ACLMessage` class automatically creates a new `ACLMessage` properly setting the receivers and all the fields used to control the conversation (conversation-id, reply-with,

in-reply-to) if any.

If we look at Figure 2, however we may notice that, when we add the above behaviour, the agent's thread starts a continuous loop that is extremely CPU consuming. In order to avoid that we would like to execute the `action()` method of the `OfferRequestsServer` behaviour only when a new message is received. In order to do that we can use the `block()` method of the `Behaviour` class. This method marks the behaviour as "blocked" so that the agent does not schedule it for execution anymore. When a new message is inserted in the agent's message queue all blocked behaviours becomes available for execution again so that they have a chance to process the received message. The `action()` method must therefore be modified as follows.

```
public void action() {
  ACLMessage msg = myAgent.receive();
  if (msg != null) {
    // Message received. Process it
    ...
  }
  else {
    block();
  }
}
```

The above code is the typical (and strongly suggested) pattern for receiving messages inside a behaviour.

5.6 Selecting messages with given characteristics from the message queue

Considering that both the `OfferRequestsServer` and `PurchaseOrdersServer` behaviours are cyclic behaviour whose `action()` method starts with a call to `myAgent.receive()`, you may have noticed a problem: how can we be sure that the `OfferRequestsServer` behaviour picks up from the agent's message queue only messages carrying requests for offer and the `PurchaseOrdersServer` behaviour only messages carrying purchase orders? In order to solve this problem we must modify the code we have presented so far by specifying proper "templates" when we call the `receive()` method. When a template is specified the `receive()` method returns the first message (if any) matching it, while ignores all non-matching messages. Such templates are implemented as instances of the `jade.lang.acl.MessageTemplate` class that provides a number of factory methods to create templates in a very simple and flexible way.

As mentioned in 5.3, we use the `CFP` performative for messages carrying requests for offer and the `ACCEPT_PROPOSAL` performative for messages carrying proposal acceptances, i.e. purchase orders. Therefore we modify the `action()` method of the `OfferRequestsServer` so that the call to `myAgent.receive()` ignores all messages except those whose performative is `CFP`.

```
public void action() {
  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
    // CFP Message received. Process it
    ...
  }
  else {
    block();
  }
}
```

5.7 Complex conversations

The `RequestPerformer` behaviour mentioned in 4.4.1 represents an example of a behaviour carrying out a "complex" conversation. A conversation is a sequence of messages exchanged by two or more agents with well defined causal and temporal relations. The `RequestPerformer` behaviour has to send a `CFP` message to several agents (the known seller agents), get back all the replies and, in case at least a `PROPOSE` reply is received, send a further `ACCEPT_PROPOSAL` message (to the seller agent that made the proposal) and get back the response. Whenever a conversation has to be carried out it is a good practice to specify the conversation control fields in the messages exchanged within the conversation. This allows to easily and un-ambiguously create templates matching the possible replies.

```
/**
   Inner class RequestPerformer.
   This is the behaviour used by Book-buyer agents to request seller
   agents the target book.
 */
private class RequestPerformer extends Behaviour {
  private AID bestSeller; // The agent who provides the best offer
  private int bestPrice;  // The best offered price
  private int repliesCnt = 0; // The counter of replies from seller agents
  private MessageTemplate mt; // The template to receive replies
  private int step = 0;

  public void action() {
    switch (step) {
    case 0:
      // Send the cfp to all sellers
      ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
      for (int i = 0; i < sellerAgents.length; ++i) {
        cfp.addReceiver(sellerAgents[i]);
      }
      cfp.setContent(targetBookTitle);
      cfp.setConversationId("book-trade");
      cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
      myAgent.send(cfp);
      // Prepare the template to get proposals
      mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),
                               MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
      step = 1;
      break;
    case 1:
      // Receive all proposals/refusals from seller agents
      ACLMessage reply = myAgent.receive(mt);
      if (reply != null) {
        // Reply received
        if (reply.getPerformative() == ACLMessage.PROPOSE) {
          // This is an offer
          int price = Integer.parseInt(reply.getContent());
          if (bestSeller == null || price < bestPrice) {
            // This is the best offer at present
            bestPrice = price;
            bestSeller = reply.getSender();
          }
        }
        repliesCnt++;
        if (repliesCnt >= sellerAgents.length) {
          // We received all replies
          step = 2;
        }
      }
      else {
```

```
        block();
      }
      break;
    case 2:
      // Send the purchase order to the seller that provided the best offer
      ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
      order.addReceiver(bestSeller);
      order.setContent(targetBookTitle);
      order.setConversationId("book-trade");
      order.setReplyWith("order"+System.currentTimeMillis());
      myAgent.send(order);
      // Prepare the template to get the purchase order reply
      mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),
                               MessageTemplate.MatchInReplyTo(order.getReplyWith()));
      step = 3;
      break;
    case 3:
      // Receive the purchase order reply
      reply = myAgent.receive(mt);
      if (reply != null) {
        // Purchase order reply received
        if (reply.getPerformative() == ACLMessage.INFORM) {
          // Purchase successful. We can terminate
          System.out.println(targetBookTitle+" successfully purchased.");
          System.out.println("Price = "+bestPrice);
          myAgent.doDelete();
        }
        step = 4;
      }
      else {
        block();
      }
      break;
    }
  }

  public boolean done() {
    return ((step == 2 && bestSeller == null) || step == 4);
  }
}  // End of inner class RequestPerformer
```

Complex conversations are typically carried out following a well defined interaction protocol. JADE provides a rich support for implementing conversations following interaction protocols in the `jade.proto` package. In particular the conversation we implemented follows a "Contract-net" protocol and could have been very easily implemented exploiting the `jade.proto.ContractNetInitiator` class. The support for interaction protocols is outside the scope of this tutorial however. Refer to the JADE Programmer's guide and to the Javadoc for the details.

5.8 Receiving messages in blocking mode

Besides the `receive()` method, the `Agent` class also provides the `blockingReceive()` method that, as the name suggests, is a blocking call: it does not return until there is a message in the agent's message queue. The overloaded version taking a `MessageTemplate` as parameter (it does not return until there is a message matching the specified template) is also available.

It is important to stress that the `blockingReceive()` methods actually blocks the agent thread. Therefore if you call `blockingReceive()` from within a behaviour, this prevents all other behaviours to run until the call to `blockingReceive()` returns. Taking into account the above consideration a good programming practice to receive messages is: use `blockingReceive()` in the `setup()` and `takeDown()` methods; use `receive()` in combination with `Behaviour.block()` (as shown in 5.5) within behaviours.

In the code we have written so far we have assumed that there is a fixed set of seller agents (*seller1* and *seller2*) and that each buyer agent already knows them (see the `AID[] sellerAgents` member variable of the `BookBuyerAgent` class in the code presented in 3.4). In this chapter we describe how to get rid of this assumption and exploit the yellow pages service provided by the JADE platform to make buyer agents dynamically discover seller agents available at a given point in time.

6.1 The DF agent

A "yellow pages" service allows agents to publish one or more services they provide so that other agents can find and successively exploit them as described in Figure 4.

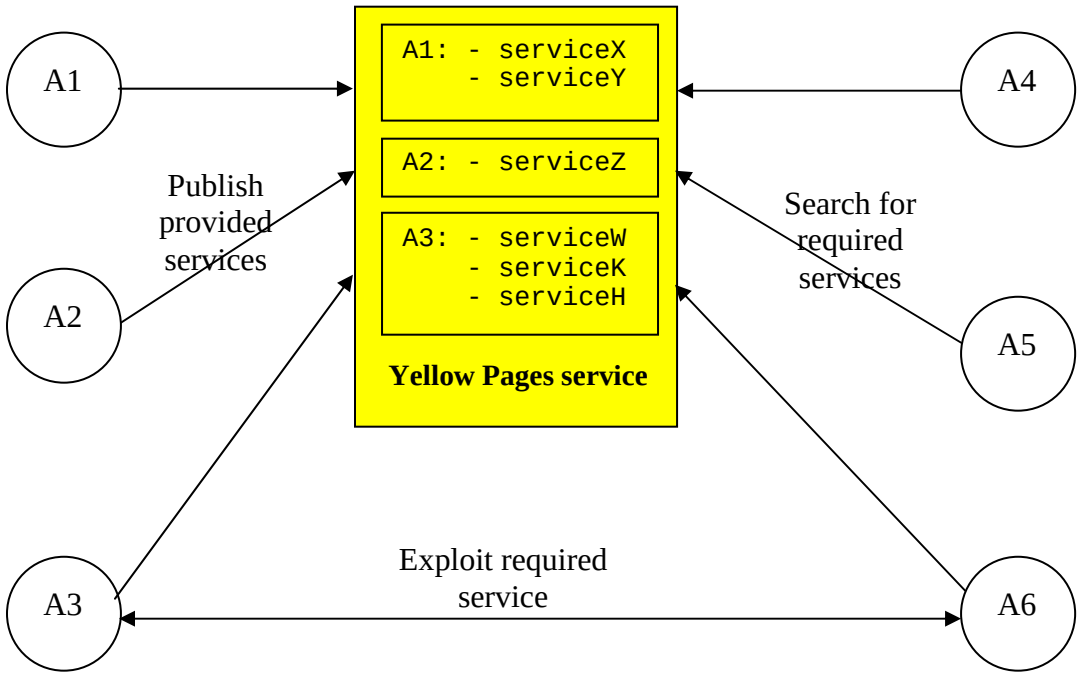

Figure 4. The Yellow Pages service

The yellow pages service in JADE (according to the FIPA specification) is provided by an agent called DF (Directory Facilitator). Each FIPA compliant platform hosts a default DF agent (whose local name is "`df`"). Other DF agents can be activated and several DF agents (including the default one) can be federated so that to provide a single distributed yellow pages catalogue.

6.2 Interacting with the DF

Being the DF an agent it is possible to interact with it as usual by exchanging ACL messages using a proper content language (the SL0 language) and a proper ontology (the FIPA-agent-management ontology) according to the FIPA specification. In order to simplify these interactions, however JADE provides the `jade.domain.DFService` class by means of which it is possible to publish and search for services through method calls.

6.2.1 Publishing services

An agent wishing to publish one or more services must provide the DF with a description including its

AID, possibly the list of languages and ontologies that other agents need to know to interact with it and the list of published services. For each published service a description is provided including the service type, the service name, the languages and ontologies required to exploit that service and a number of service-specific properties. The `DFAgentDescription`, `ServiceDescription` and `Property` classes, included in the `jade.domain.FIPAAgentManagement` package, represent the three mentioned abstractions.

In order to publish a service an agent must create a proper description (as an instance of the `DFAgentDescription` class) and call the `register()` static method of the `DFService` class. Typically, but not necessarily, service registration (publication) is done in the `setup()` method as shown below in the case of the Book seller agent.

```
protected void setup() {
  ...
  // Register the book-selling service in the yellow pages
  DFAgentDescription dfd = new DFAgentDescription();
  dfd.setName(getAID());
  ServiceDescription sd = new ServiceDescription();
  sd.setType("book-selling");
  sd.setName("JADE-book-trading");
  dfd.addServices(sd);
  try {
    DFService.register(this, dfd);
  }
  catch (FIPAException fe) {
    fe.printStackTrace();
  }
  ...
}
```

Note that in this simple example we do not specify any language, ontology or service-specific property. When an agent terminates it is a good practice to de-register published services.

```
protected void takeDown() {
  // Deregister from the yellow pages
  try {
    DFService.deregister(this);
  }
  catch (FIPAException fe) {
    fe.printStackTrace();
  }
  // Close the GUI
  myGui.dispose();
  // Printout a dismissal message
  System.out.println("Seller-agent "+getAID().getName()+" terminating.");
}
```

6.2.2 Searching for services

An agent wishing to search for services must provide the DF with a template description. The result of the search is the list of all the descriptions that match the provided template. A description matches the template if all the fields specified in the template are present in the description with the same values.

The `search()` static method of the `DFService` class can be used as exemplified in the code used by the Book buyer agent to dynamically find all agents that provide a service of type "book-selling".

```
public class BookBuyerAgent extends Agent {
  // The title of the book to buy
  private String targetBookTitle;
  // The list of known seller agents
  private AID[] sellerAgents;
```

```
// Put agent initializations here
protected void setup() {
  // Printout a welcome message
  System.out.println("Hallo! Buyer-agent "+getAID().getName()+" is ready.");

  // Get the title of the book to buy as a start-up argument
  Object[] args = getArguments();
  if (args != null && args.length > 0) {
    targetBookTitle = (String) args[0];
    System.out.println("Trying to buy "+targetBookTitle);

    // Add a TickerBehaviour that schedules a request to seller agents every minute
    addBehaviour(new TickerBehaviour(this, 60000) {
      protected void onTick() {
        // Update the list of seller agents
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("book-selling");
        template.addServices(sd);
        try {
          DFAgentDescription[] result = DFService.search(myAgent, template);
          sellerAgents = new AID[result.length];
          for (int i = 0; i < result.length; ++i) {
            sellerAgents[i] = result[i].getName();
          }
        }
        catch (FIPAException fe) {
          fe.printStackTrace();
        }
        // Perform the request
        myAgent.addBehaviour(new RequestPerformer());
      }
    } );
    ...
```

Note that the update of the list of known seller agents is done before each attempt to buy the target book since seller agents may dynamically appear and disappear in the system. The `DFService` class also provides support for subscribing to the DF to be notified as soon as an agent publishes a given service (see the `searchUntilFound()` and `createSubscriptionMessage()` methods), but this is outside the scope of this tutorial.