CentraleSupélec

Multiagent systems
Introduction to JADE

N. Sabouret and W. Ouerdane

January the 16th, 2019

Preamble

This practical session is timed for mode that 1h30min. Do not worry if you cannot finish all the exercises today. You will have some time to finish it next week.

1 General information

The Java Agent Development Environment (JADE) is a distributed multi-agent platform. It supports multiple hosts and efficient computing. It consists of a Java API that lets you program autonomous agents and interactions quite easily.

Agents in the platform run asynchronously: each agent is a different Java *thread*.

Agents' actions and interactions are defined through a set of *behaviours*. These behaviours are at the very heart of JADE programming. All important operations performed by the agent is programmed as part of one or several *behaviours*. Understanding how to design and implement such *behaviours* is the most important part in learning how to use JADE.

The JADE platform is FIPA-compliant. It comes with a powerful Agent Management System (AMS) that handles multiple hosts. Each host must have its own *container* to handle agents and all containers refer to the MAS main *container*. While you will probably not use this feature for this course, know that the complete runtime can be distributed among several computers.

JADE also offers a Directory Facilitator (DF) for agents to register and search for services. This is the most convenient way for having agents know who to interact with.

1.1 Installation

Download the `jade.jar` Java library. You can find it on the JADE website:

```
http://jade.tilab.com/
```

If you are using an IDE such as Eclipse, NetBeans or any other (and this is most certainly a good idea when programming), you must include this library into the "build path" of your project.

1.2 Runing JADE from command line

The main class to start the JADE platform is the `jade.Boot` class. The command line to run the platform is as follows:

```
java -cp jade.jar:. jade.Boot -local-host 127.0.0.1 \
   -agents "test1:AgentTest;test2:AgentTest"
```

As you can see, the `jade.Boot` main method requires several arguments:

- `local-host` to indicate where is the main JADE container located (`127.0.0.1` means that you intend to run it locally);
- `agents` to indicate the platform to load one or several agents based on their class names in the Java classpath. The list of all agents is included between double quotes, separated by a semicolon. Each agent is defined by its name and its Java class, using the `name:class` syntax as in the above example.

If you are using an IDE, you will want to modify the *Runtime Configuration* for it to start the `jade.Boot` class instead of one of your Java class with the required parameters. Again, this is only possible once you have added the `jade.jar` library to your *build path*.

1.3 Documentation

During the first 1h30min of this course, you will learn how to program Agents in JADE. You will most certainly want to bookmark the JADE API documentation available at:

```
http://jade.tilab.com/doc/api/index.html
```

2 Agents

This section is about writing a simple agent and running it with the JADE platform.

2.1 Your very first agent

To create an agent, you need to implement a subclass of the `jade.core.Agent` class. In order to run any code in your agent, it is recommended that you override the `setup` method that is called when the agent is started by the AMS. **Do not use a constructor**.

Exercise 1

1. Write a `TestAgent` class. Initialise each agent with a unique integer value and print this number upon agent's startup (in the `setup` method, not in the constructor).
2. Configure your IDE (or use the command line) to start two instances of this class `a1` et `a2`. The main class must be `jade.Boot` and the options given to as "program arguments" should be, as explained previously:

   ```
   -local-host 127.0.0.1 -agents "a1:TestAgent;a2:TestAgent"
   ```

3. Test your program. You should see, in addition to the red messages telling you that the JADE platform is running, the two expected outputs from your `setup` method.

Warning: Although nothing more happens once the two agents have written their messages, your agents and your platform are still running! You need to stop the JADE platform manually by using CTRL+C in the command line or by clicking on the red button in your IDE. This is **mandatory** after every test, so that you release the Jade network ports.

If you try to start the platform again before shutting it down, you shall see the following error message:

```
jade.core.IMTPException: No ICP active
```

2.2 Starting agents from Java code

It is not convenient to start agents manually via the command line and the `jade.Boot` class. One limit of this method is that you need to manually enter the name and classes of all agents, which is not something you want to do when you project requires hundreds of agents. The solution is to write Java code that can start the agents.

To create agents from Java code, you need to get access to the JADE *agents container* and to use the `createNewAgent` method. This can be done:

- By starting the platform manually;
- By using the `getContainerController` method from an agent's code.

Starting the JADE platform and to have it load agents directly from Java is done using the following code:

```
import jade.core.Runtime;
import jade.core.Profile;
import jade.core.ProfileImpl;
import jade.wrapper.AgentContainer;
import jade.wrapper.AgentController;
import jade.wrapper.StaleProxyException;

methode() throws StaleProxyException {
  Runtime rt = Runtime.instance();
  rt.setCloseVM(true);
  Profile pMain = new ProfileImpl("localhost", 8888, null);
  AgentContainer mc = rt.createMainContainer(pMain);
}
```

Warning: import the correct classes (the same name can appear in different packages) and do not accidentally replace «`createMainContainer`» with «`createAgentContainer`». These are the most common beginner errors.

Starting the agent

Once you have started a platform (either using the above code or via the `jade.Boot` class) and obtained the agents container, starting an agent is done with the following code:

```
AgentController test =
  container.createNewAgent("test",TestAgent.class.getName(), new Object[0]);
test.start();
```

The first argument to the `createNewAgent` medhod is the local name of the agent; the second one is the class; the third one is a list of parameters, that will be discussed in the next session.

Exercise 2

1. Write a Java class `Launcher` with a `main` method that starts the JADE platform with two instances of `TestAgent`.
2. Create a new Runtime Configuration with this class and test your program. Do not forget to «kill» the JADE process after your test to release the ports, unless you wan to see the famous «Cannot bind server socket» with the «No ICP active» exception...

2.3 Setup and parameters

A JADE agent must always have a default constructor with no parameter (which is the case if you do not write any constructor). This is the method that will be invoked by the agent creation code in the AMS.

However, you can also initialise your agents with some parameters that are being computed by the platform. These parameters are given in the `Object` array as the last parameter of the `createNewAgent` method and can be accessed by the agent in its `setup` method with the `Agent.getArguments` method.

Keep in mind that 1) you will need to cast the elements, knowing their initial class from the agent launcher code; 2) you cannot access these arguments from the constructor (because the argument list is provided to the agent *after* its creation): you need to do this in the `setup` method.

You never write an agent's constructor method in JADE.

Exercise 3

1. Modify your previous code so that the unique ID is provided to the `TestAgent` by the `Launcher`.

3 Behaviours

The actions of agents are defined in *behaviours*. A behaviour is an instance of any subclass of `jade.core.behaviours.Behaviour`. Each behaviour implements two methods:

- The `action` method defines what the behaviour actually does;
- The `done` method returns `true` when the behaviour is over.

The whole idea of JADE programming is that an agent's behaviours must be interleaved for running. All behaviours of an agent are stored in a queue. The scheduler selects the first behaviour, calls the `action` method to run **a small part** of this behaviour, and then calls `done` to check if the behaviour is over. If it is not, the behaviour is added to the end of the queue.

As a consequence, you **have to** write `action` methods that quickly gives back the control, knowing that it will be very called again soon. The `done` method simply returns `true` as long as the behaviour need to be continued. The code of the `action` method thus generally defines a **finate-state machine** through a series of self-exclusive instructions:

```
if state=x:
  do this and that
  return
```

One additional note before writing behaviours

Behaviours are at the core of the JADE agents architecture. Each agent is a separate Java *thread* with its own actions described in *behaviours*. These behaviours are not threads but the agent's runtime will interleaves them by using successive calls to the `action` and `done` methods. There is no real procedural loop, such as in the PRS architectures, but the JADE programmer has to recreate a similar mechanism with very small «perceive, compute, act» instructions sets in the `action` method.

Behaviours will be used to:

- Compute changes in the agent's internal state;
- Receive and send messages;
- Start new behaviours (by adding them to the agent's behaviours queue);
- Create other agents.

In any behaviour, the `myAgent` attribute refers to the owner agent. It can be used (with the propre cast) to call ad-hoc methods or attributes.

3.1 You very first behaviour

Adding behaviours to an agent is done using the `addBehaviour` method. You will need to do this in the `setup` method to provide your agent with initial behaviours, but you might add new behaviours later as the execution goes on.

Note that when an agent has no active behaviour in its queue, it does no use any computational resources. However, it still "runs" on the platform[1].

Exercice 4

1. Write a new class named `InitialTestBehaviour` that has the following behaviour for your `TestAgent`:
   - First, write its unique ID followed by the text «starting», then returns.
     The ID can be obtained by a getter method in your `TestAgent` class. You need to cast the `myAgent` attribute in the `Behaviour` class to your `TestAgent` class.
   - Performs N steps (N an arbitrary value) that consist in writing its ID followed by «at step X». It returns between each of these steps.
   - Last, writes its ID followed by «finished» and then marks the behaviour as done.
2. Add this behaviour using the `setup` method and test with two agents.

If N is small enough, you will notice that the behaviours of the two agents are not interleaved. The agent first created finishes its behaviour before the second agent starts. This can be changing by asking the agent to make a pause during each execution step.

3.2 Pausing the execution

There exists four different solutions to suspend the agent runtime for a given period of time:

- The first one consists in calling the `Thread.sleep` method, as you would do for any Java program. Since JADE agents are threads, this would stop **all execution of the agent's** (not only the current behaviour).

  While this is a good idea if you want to reduce the CPU load or slow down the entire simulation, it blocks all the behaviours, including the message processing. This is generally not what you want to do: agents should always be able to answer messages they receive!

- The second approach to suspending the code consists in using the `Agent.blockingReceive` method, which also blocks the whole agent's execution until it receives a message. This method might be required in some specific cases that will be seen later in this tutorial.

- The **the most common solution**, however, consists in using the `Behaviour.block` method. Only the calling behaviour will be suspended, for a fixed time or until it receives a message. This allows you to slow down the execution of some steps of the simulation **and** to reduce the CPU load (keep in mind that an agent with no active behaviour does not use any CPU time) **while still having an agent that can react to external messages**.

  However, you must understand that:

  – Writing `block()` in your code not magically ends the `action` method. It simply changes an attribute's value in the behaviour. Thus, the behaviour will be actually blocked only after the `action` method returns.

  It is highly recommended that `block` is you last instruction before the return of the `action` method.

[1] To remove an agent from the platform, you need to call the `doDelete` method from a behaviour, but we will probably not need it.

  - All behaviours are re-activated everytime the agent receives a message: you will need to re-block your behaviour if it is not concerned by the message.

- Another method that is commonly used is to rely on **specific types of behaviours** such as `TickerBehaviour` or `WakerBehaviour`. We will see these behaviours later in the tutorial.

In any case, it is very important to ensure that the pausing the agents **does not impact the CPU load**. In particular, you must ensure that you did not accidentally implement any "active waiting" solution that consists in having a behaviour looping on a given state. Watch your CPU usage!

Exercice 3

1. Using the `block` method, introduce random waiting times at each step of the previously created behaviour.

2. Test your simulation with different values of N.

3.3 Combining two different behaviours

In the `jade.core.behaviours` package, you can find a complete set of `Behaviour` subclasses. Each subclass correspond to a specific behaviour. For instance, the `CyclicBehaviour` class has a `done` method that always answers `false` (the behaviour never stops). This is useful for message listening behaviours.

Exercice 4

1. Move the "step" number (the variable that is displayed by the behaviour) from the `InitialTestBehaviour` class to your `TestAgent` class, if this has not already been done.

   You can still access this variable from the behaviour using getters and setters, and a proper cast on `myAgent`.

   It is generally a good idea, in JADE, to have the **state variables be managed at the agent level**, rather than inside a behaviour. This allows you to share this variable between different behaviours, as in the next question.

2. Use the `TickerBehaviour` class to add a new behaviour to your agents, name `SecondTestBehaviour`. At regular time periods (defined manually in your behaviour constructor), the agent must it ID and the number of the next step.

3. Test this new behaviour with the two agents. You should see interleaving between the agents and between the behaviours of a single agent.

3.4 And where is the Environment in all this?

JADE agent-oriented programming relies exclusively on direct interactions. There is no such thing as a shared environment.

Of course, you can still define a class or an object (given to the agents via the parameters) that is shared by several agents and have them modify this variables. This is a bad idea, generally speaking. First, you will have to deal with concurrent access, using synchronisation monitors (Java's `synchronized` keyword). The role of an agents platform is to avoid the programmer to deal with such difficulties. Second, JADE's philosophy is to keep things separated between the different pieces of code and to use messages to deal with information exchange. The communication mechanism can be very efficient when used wisely.

As a consequence, if you need some "shared variables", Artifact or other blackboard-based mechanism in your MAS design, please consider this as an Agent that will process messages from other agents.

4 Message sending and receiving

In this section, we will learn how to handle JADE messages.

4.1 General principle

Agents can (and must) communicate using ACL messages and the `send` and `receive` methods. The data structure for messages is defined in `jade.lang.acl.ACLMessage`.

Messages **must** have a performative. That is require by the default constructor. Performatives are simply integer values that you can define using `static final` variables in your code. The `ACLMessage` comes with a set of 22 pre-defined FIPA performatives. Do not hesitate to define yours! The semantics of the performative is up to you!

You can add a content to the message when this is required by your message type (*i.e.* the performative). This can be done using the `setContent` or `setContentObject` methods. If you build an homogeneous MAS, it is highly recommended to use the `setContentObject` method so that you do not have to parse the contents manually (but make sure that your contents are serializable objects).

The recipients of the message are added using the `addReceiver` method. This method takes as argument an object of type `jade.core.AID` (Agent ID). This identifier, unique for each agent, allows the AMS to carry messages to the agents. It is build after the name of the agent and the JADE container that runs it.

Note

When you define your own performatives, the `toString` method in the ACLMessage class print NOT-UNDERSTOOD for the performative name. This is not a problem, except for debugging.

4.2 Sending and receiving

As explained in the lecture, the `send` method is a non-blocking one: the agent' runtime resumes immediately. For message reading, JADE offers two different methods:

- The `blockingReceive` method stops **the agent's runtime** (and not only the behaviour) until a message arrives. It should not be used in the general case, unless you really know what you are doing.
- The **most frequently used** `receive` method does not wait to receive a message. It simply looks into the mailbox for one and returns `null` if there is none.

The usual way to receive message in JADE is to use the following code in the `action` method of a `CyclicBehaviour`:

```
ACLMessage m = myAgent.receive();
if (m==null)
  block();
else { ... message processing ... }
```

Exercise 5

1. Using the constructor from the `AID` class with the option `isGUID` set to `false`, add to each of your test agents an attributes that contains the agent ID of the other one.

   This can be done by either one of the two proposed solutions:

   (a) Add a parameter in the agent creation with the name of the other agent;

   (b) Access the agent's local name via the `getLocalName` method when necessary and deduce the name of the other agent.

   The third method consists in using the DF. We will learn how to do this very soon.

2. Modify the initial behaviour of your test agents so that they send a message with performative "Inform" and content "done" to the other agent once the last step is finished.

3. Write (and add to your agents) a `CyclicBehaviour` class that deals with the message reception and processing. Each time such a message arrives, the agent write on the standard output its own name, the name of the message sender and the text "said it has finished counting".

   Note: you must not use the `blockingReceive` method because you want your agent to continue running the initial counting behaviour while receiving messages.

4. Test this new program and **check the CPU usage**.

4.3 Using filters

The `receive` method, when used without any parameter, checks for any message in the mailbox. It really catches anything.

However, you do not want to write one big reception behaviour to catch all messages and decide what to do with it. The JADE approach consists in using several reception behaviours, each one having its own purpose. In this context, each behaviour needs to **filter** the messages upon reception to avoid "stealing" the message intended for another one.

The `receive` is thus generally used with filters to capture only messages that can be processed by the behaviour. Writing filters is a bit tricky when you are not used to it. You first need to implement the `MessageTemplate.MatchExpression` interface and its abstract `match` method. The code in this method can be anything, from comparing the performative and the content of the message with some expected value to more complex checks on the message's fields. Once this is done, you can build a filter as an instance of `jade.lang.acl.MessageTemplate` using the following code:

```
MessageTemplate mt = new MessageTemplate(new A());
```

where `A` is your class that implements `MatchExpression`. We recommend that you do this using the **Factory design pattern**, i.e. a static variable or method in Jave, to avoid creating new objects every time you want to filter a message.

Exercise 6

1. Write a filter for the "done" messages;

2. Modify the reception behaviour so that it only captures these messages.

3. Modify the `SecondTestBehaviour` class (i.e. the ticker behaviour) so that it also sends messages, with a different performative or content, to the other agent.

4. Write a second reception behaviour that only processes these messages.

5. Test your program and ensure that messages are properly processed by each corresponding behaviour.

What we have done here is implementing in Jade two very simple *communication protocols*. Each protocol consists in sending and receiving a simple message. The next subsection explains how to write more complex protocols.

4.4 Communication protocols

Let us imagine that we need to write agents that negotiate with each other using specific protocols. This is what we are going to do in the next practical session.

To implement the communication protocol in the JADE platform, the simplest method is to implement a behaviour for each AUML role. Each behaviour will be in charge of:

1. Following the progress of the protocol on its corresponding role's side;

2. Sending an receiving messages.

As an example, we want to write a more complex protocol. When an agent receives a "done" message, it answers either with "already done" or "still going" depending on its own current status.

Exercise 7

1. Draw the AUML diagram corresponding to the proposed above protocol;
2. Modify the first reception behaviour so that it answers when receiving a "done" message;
3. Create a new behaviour that sends a "done" message and then switches to a waiting stage (using `receive` and `block`) until it receives the confirmation, and only writes its finishing message once this protocol is over.
4. Test your code.

4.5 Using the Directory Facilitator

It is not reasonable to believe that agents will know *a priori* the local names or AIDs of all the peers they need to communicate with. Imagine that you want to scale up the above program to a hundred agents, or simply that the MAS is open and that new agents can appear later on during the simulation.

This is the reason JADE agents use the Directory Facilitator when they need to know the list of agents that can fulfil a given role in a protocol. This is done using the `DFAgentDescription` and `ServiceDescription` from the `jade.domain.FIPAAgentManagement` package.

4.5.1 Registering the agent

The first thing to do when implementing a protocol is to define a `ServiceDescription` that correspond to each role. As for message templates, you shall use factories to avoid unnecessary object creation.

Each `ServiceDescription` must have a name (`setName`) and a type (`setType`). These two `String` elements can be identical (in our examples, we will not use the difference between the two).

Upon creation (in the `setup` method), each agent will create a `DFAgentDescription` and to add the different possible roles (defined ad several `ServiceDescription` objects) to this agent description. It then registers to the DF using the `jade.domain.DFService.register` static method.

4.5.2 Searching for an agent

At any time of its execution, an agent can search for the list of agents that correspond to a given role (i.e. a given `ServiceDescription`). This is achieved with the `jade.domain.DFService.search` static method. The first argument of this method is the requesting agent (*i.e.* the `myAgent` variable in the `Behaviour` class) and the second argument is the `DFAgentDescription` of the agents, seen as a pattern. This is the reason why the service description should be a factory.

Exercise 8

1. Have your test agents register to the DF as "counters".
2. Modify the sending behaviours so that the agents search for the list o other counters' AIDs in the DF before sending a message.
3. Ensure that your behaviour awaits for an answer for all other agents that have been contacted.
4. Test your program with **3, 4 or 5 agents**.

5 More of JADE

Mastering the JADE platform will require some practice. In this section, we give you some information to help you working with its debugging features.

5.1 Using the JADE *RMA* and *Sniffer* agents

JADE comes with several handy agents that can help you programming a complex MAS. One of these agents is the *Remote Monitoring Agent* (RMA). It allows controlling the life cycle of all the agents. It is provided with a Graphical User Interface agents that serves as a user control panel to the platform.

Another handy agent is the *Sniffer* agent that tracks the messages and draws sequence diagrams to help you see what happens in your MAS.

5.1.1 The RMA agent

The RMA agent is an instance of the `jade.tools.rma.rma` class. It can be started directly from the command line using the `-gui` option to the `jade.Boot` class:

```
java -jar jade.jar:. jade.Boot -gui -local-host 127.0.0.1 \
  -agents "a1:TestAgent;..."
```

Or simply loaded as an any other agent via the *agents contrainer* in the command line:

```
java -jar jade.jar:. jade.Boot -local-host 127.0.0.1 \
  -agents "gui:jage.tools.rma.rma;a1:TestAgent;..."
```

Or using the following Java code in the launcher class or in any agent's setup or behaviour:

```
mc.createNewAgent("gui",jade.tools.rma.rma.class.getName(), \
  new Object[] {} ).start();
```

5.1.2 The Sniffer agent

The Sniffer agents is an instance of the `jade.tools.sniffer.Sniffer` class. It should be started as any other agents via the *agents contrainer*, either in the command line:

```
java -jar jade.jar:. jade.Boot -gui -local-host 127.0.0.1 \
  -agents "sniffer:jage.tools.sniffer.Sniffer;a1:TestAgent;..."
```

Or using the Java code in the launcher or in an agent's code:

```
mc.createNewAgent("gui",jade.tools.sniffer.Sniffer.class.getName(), \
  new Object[] {} ).start();
```

You can also use the GUI to load the agent from its class name!

5.2 Synchronisation of the agent's activation

One difficulty when running simulations in Jade is that you have no power on the agent's startup moment. The agents start immediately once the `start` method is called or the `jade.Boot` class loads them, depending or your starting method.

For instance, you do not want your agents to start running *before* the sniffer agent is ready to catch the bypassing messages!

In some situation, just starting the agents in the right order is not sufficient: you might want for some agent to have done something before others activate.

Several solutions exist at this point

5.2.1 Launcher agent

Keep in mind that loading and starting an agent can easily be done from any instance of `Behaviour` of any instance of `Agent`! This is probably the easiest, if not the cleanest solution to make sure that a given agent did not start before another.

However, it does not support the synchronisation of your agents, *e.g.* if you want a piece of code to be run (like DF registering) for all agents before another is done.

5.2.2 Synchroniser agent

The best solution is to write a synchronisation agent that waits for a signal from all agents that they are ready and then orders all agents to start running for real.

This agent must have a hard-coded local name that you will give as an argument (or as a `public static final` attribute) to all your agents, so that they know its AID. This agent has must also know (arguments or attribute) the number of agents that will be started in the simulation, so as to wait for them.

The `setup` code of this agent must only start one behaviour whose role is to collect the N agent adresses (AID) by receiving messages (*e.g.* with performative *Inform* and content "*ready*"). Once the all agents have send a message, the behaviour answers to them all (*e.g.* with a performative *Request* and content "*start*").

All other agents should perform the required operations (DF registering, initialisation instructions and MAS preparation) in their setup method, send a message to the synchroniser agent (still in their `setup` code) and end by initiating a simple `oneShotBehaviour` with a `blockingReceive` instruction that waits for the *start* message.

When the message is received, the behaviour will add all the required behaviours using `addBehaviour` as you would have normally done in your `setup` method.