Java Agent DEvelopment Framework is an Open Source platform for peer-to-peer agent based applications

http://jade.tilab.com/
JADE

- A framework for developing multi-agent systems
- FIPA-compliant
- Written in JAVA, consists of an API with several packages

Agent platform:
AMS and DF

- **Agent Management System (AMS)**
  - Supervises access and usage of the agent platform
  - Maintains a directory of agent identifiers (AID) and agent state
  - Each agent registers in the AMS to get a valid AID
  - In a JADE platform, there is only one AMS

- **Directory Facilitator (DF)**
  - Yellow page service

When a JADE platform is launched, both the AMS and the DF are automatically created
Containers

- A platform may have several containers, where agents live
- Containers are not necessarily in the same machine
- 1 JVM per machine, 1 thread per agent
- There is a special container in each platform: main container, which includes the AMS and the DF

Communication
- Receiver in the same container: event-notification
- Receiver in the same platform, but in a different container: RMI
- Receiver in a different platform: IIOP
Mobility and Cloning

- Supports intra-platform mobility and cloning
  
  - JADE platform may be distributed among several machines (each one having its own container)
  
  - Agents may migrate between containers
    
    ```java
    doMove(Location)
    beforeMove() / afterMove()
    ```
  
  - Agents may be cloned
    
    ```java
    doClone(Location, String)
    beforeClone() / afterClone()
    ```
JADE Execution

- Change the CLASSPATH to include:
  
  
  `%JADE%\lib\jade.jar;`  
  `%JADE%\lib\commons-codec\commons-codec-1.3.jar;`

- Execute the command:
  
  ```
  java jade.Boot [options] [agentlist]
  ```

  - launches a *main container*, including AMS and DF

  ```
  [options]
  -container
  -host HostName (default is localhost)
  -port PortNumber (default is 1099)
  -gui (Remote Monitoring Agent)
 ...
  ```

  ```
  [agentlist]  semicolon (;) separated agent identifiers
  identifier = name: class_name (comma_sep_arguments)
  ```

- NOTE: it is also possible to launch JADE from Java code!
Launching Agents

• Launching JADE agents
  i. Using the **RMA** (*Remote Monitoring Agent*)
     • select a container and choose “Start New Agent”
  
  ii. From the command line, when starting JADE or launching a new container
     • `java jade.Boot name:class_name(arguments)`
       - `java jade.Boot ping0:examples.pingAgent.PingAgent`

  iii. Programmatically, using the **ContainerController** class
     • `acceptNewAgent(name, agent)`
     • `createNewAgent(name, class, arguments)`
Launching Containers

- Multiple containers
  - In the same machine
    - Start JADE
      ```
      java jade.Boot
      ```
    - Add a new container with an agent
      ```
      java jade.Boot -container a:ClasseA
      ```
  - In different machines
    - Start JADE in machine PCAna
      ```
      java jade.Boot
      ```
    - Add new container in the machine PCJohn with an agent
      ```
      java jade.Boot -host PCAna -container a:ClasseA
      ```

⇒ agent $a$ lives in the machine PCJohn, but belongs to the platform running in the machine PCAna; its GUID (Globally Unique ID) is $a@PCAna:1099/JADE$
JADE Tools

- Tools that simplify platform administration and application development
  - Remote Monitoring Agent
  - Dummy Agent
  - Sniffer Agent
  - Introspector Agent
  - Directory Facilitator GUI
Remote Monitoring Agent (RMA)
  - Graphical console to monitor the agent platform activity
  - Visualization of container and agent states
  - There can be more than one RMA in the same platform

Launching RMA:
  - As a regular agent
    java jade.Boot myConsole:jade.tools.rma.rma
  - Using the “-gui” option
    java jade.Boot -gui
Dummy Agent

• Useful tool for debugging
• Allows interaction with other agents by sending, receiving and viewing ACL messages

• How to launch:
  – From the command line
    • `java jade.Boot da:jade.tools.DummyAgent.DummyAgent`
  – From the RMA GUI
    • choose “Tools → Start DummyAgent”
    • or choose “Start New Agent” and provide `jade.tools.DummyAgent.DummyAgent` as the class name
Dummy Agent
Sniffer Agent

• When you sniff an agent (or a group), any messages sent to/by the agent are visualized in a kind of UML sequence diagram.

• When an agent or a container is created or destroyed, the Sniffer Agent is informed by the AMS.

• How to launch:
  – From the command line
    \[\text{java} \ \text{jade.Boot snif:} \text{jade.tools.sniffer.Sniffer}\]
  – From the RMA GUI
    • choose “Tools → Start Sniffer”
    • or choose “Start New Agent” and provide \text{jade.tools.sniffer.Sniffer} as the class name
Sniffer Agent
• DF is a yellow page service: agents can register their services or search the DF
• In each platform, there is at least one DF

• How to launch the GUI:
  – From the RMA GUI
    • choose “Tools → Show the DF GUI”
Introspector Agent

- Monitors the life cycle of an agent: messages sent/received, behaviour queue
- How to launch:
  - From the RMA GUI
    - choose "Tools → Start IntrospectorAgent"
JADE Agents

- Class **Agent**
  base class to build agents

- Agent **execution**
  1. Constructor is executed
  2. Agent is given an identifier and is registered in the AMS, and is put in the ACTIVE state
     - identifier=lastname+“@”+platformmachine:port+“/JADE”
  3. The **setup()** method is executed
     - used to:
       - (optionally) modify the data registered with the AMS
       - (optionally) attribute services to the agent, and register with one or more domains (DFs)
       - add *behaviours* (tasks) which will be scheduled as soon as the setup() method returns
JADE Agents

• **Stop agent execution**
  - `Agent.doDelete()` – stops the agent execution
  - `Agent.takeDown()` – method to be overridden for any cleanup (e.g. unregister from DF) before the agent is destroyed

• **Communication**
  - Asynchronous message passing
  - Usage of objects of the `ACLMessage` class
  - `Agent.send(msg)` – sends a message
  - `Agent.receive()` – gets a message from the message queue
    • another version with `MessageTemplate`
  - `Agent.blockingReceive()` – gets a message from the queue but suspends all agent activity until a message is actually received
    • other versions with `MessageTemplate`, `timeout`
Behaviours

• **Implementing behaviours**
  – Agent tasks are implemented as `Behaviour` objects
    • `Agent.addBehaviour(Behaviour)`
    • `Agent.removeBehaviour(Behaviour)`
  – The API includes several `Behaviour` subclasses
  – There is a behaviour queue; scheduling follows a *round-robin* policy
    • Pick the first behaviour
    • Execute its `action()` method
    • Invoke its Boolean `done()` method to check if the behaviour is finished
      – if yes, the behaviour is removed from the queue
      – if no, the behaviour goes back to the end of the queue
    • Pick the next behaviour in the queue…
setup()

Agent has been killed (doDelete() method called)?

NO

Get the next behaviour from the pool of active behaviours

NO

b.done()?

YES

Remove currentBehaviour from the pool of active behaviours

takeDown()

- Initializations
- Addition of initial behaviours

Highlighted in red the methods that programmers have to implement

- Agent “life” (execution of behaviours)

- Clean-up operations
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

// classe do agente
public class PingPong extends Agent {

    // classe do behaviour
    class PingPongBehaviour extends SimpleBehaviour {
        private int n = 0;

        // construtor do behaviour
        public PingPongBehaviour(Agent a) {
            super(a);
        }
    }
}
public void action() {
    ACLMessage msg = blockingReceive();
    if (msg.getPerformative() == ACLMessage.INFORM) {
        System.out.println(++n + " " + getLocalName() + ": recebi " + msg.getContent());
        // cria resposta
        ACLMessage reply = msg.createReply();
        // preenche conteúdo da mensagem
        if (msg.getContent().equals("ping"))
            reply.setContent("pong");
        else reply.setContent("pong");
        // envia mensagem
        send(reply);
    }
}

public boolean done() {
    return n==10;
}

} // fim da classe PingPongBehaviour
// método setup
protected void setup() {
    String tipo = "";
    // obtém argumentos
    Object[] args = getArguments();
    if (args != null && args.length > 0) {
        tipo = (String) args[0];
    } else {
        System.out.println("Não especificou o tipo");
    }

    // regista agente no DF
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setName(getName());
    sd.setType("Agente " + tipo);
    dfd.addServices(sd);
    try {
        DFServiceservice.register(this, dfd);
    } catch (FIPAException e) {
        e.printStackTrace();
    }
}
// cria behaviour
PingPongBehaviour b = new PingPongBehaviour(this);
addBehaviour(b);

// toma a iniciativa se for agente "pong"
if (tipo.equals("pong")) {
  // pesquisa DF por agentes "ping"
  DFAgentDescription template = new DFAgentDescription();
  ServiceDescription sdl = new ServiceDescription();
  sdl.setType("Agente ping");
  template.addServices(sdl);
  try {
    DFAgentDescription[] result = DFService.search(this, template);
    // envia mensagem "pong" inicial a todos os agentes "ping"
    ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
    for (int i=0; i<result.length; ++i)
      msg.addReceiver(result[i].getName());
    msg.setContent("pong");
    send(msg);
  } catch(FIPAException e) { e.printStackTrace(); }
}

} // fim do metodo setup
Example

```java
// método takeDown
protected void takeDown() {
    // retira registo no DF
    try {
        DFService.deregister(this);
    } catch (FIPAException e) {
        e.printStackTrace();
    }
}
} // fim da classe PingPong
```
Jess
the Rule Engine for the Java Platform
http://www.jessrules.com/
JESS – Java Expert System Shell

• A rule engine that very efficiently applies rules to data.

• Developed at Sandia National Laboratories in late 1990s, created by Dr. Ernest J. Friedman-Hill.

• Inspired by the AI production rule language CLIPS.

• Fully developed Java API for creating rule-based expert systems.

• How does Jess work?
  – Jess matches facts in the fact base to rules in the rule base.
  – The rules contain function calls that manipulate the fact base and/or other Java code.
  – Jess uses the Rete algorithm to match patterns.
Jess Architecture Diagram

- Working Memory
- Rule Base
- Inference Engine
  - Pattern Matcher
  - Agenda
- Execution Engine
The Jess Language

- Architecturally inspired by CLIPS
- LISP-like syntax.
- Basic data structure is the list.
- Can be used to script Java API.
  - JAVA --> JESS
- Can be used to access JavaBeans.
  - JESS --> JAVA
  - you can create any Java object and access its methods from within Jess!
- Easy to learn and use.
• Jess rules to respond to ACM messages
  – use a Jess engine within a agent *behaviour*
JESS and JADE – integration

• JESS may be used to implement a reasoning module of a JADE agent
  – in one of the several agent *behaviours*

  **JADE** offers the environment and facilitates message sending/receiving
  **Jess** enables a declarative implementation of the decision module

• Further information: