JADE: Java Agent DEvelopment Framework Basics

Stefano Mariani
s.mariani@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2014/2015
1 Back to JADE Architecture

2 JADE Agents

3 JADE Messaging
Disclaimer

All the material presented in these slides is rearranged by the author from a collection of documents kindly made available by the Jade team.

Credits for all the stuff (text & images) go to the Jade team, in particular to Giovanni Caire.

Credits for all the mistakes go to the author.
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
Recap on JADE Architecture II

Containers

- agents runtimes, the environments without which agents cannot exist
- one main container for each JADE platform...
- ... but many peripheral containers may coexist in the same platform and in the same host too
- they automagically register themselves to the (default/given) main container
- one single JVM executed per host/platform (2 JADEs on the same host are 2 JVMs)
Agent Management System (AMS)

- **JADE white pages service**
- **one AMS service (agent) for each JADE platform**
- always runs in the main container
- is contacted (automagically) by every JADE agent upon start...
  - AMS register() method called prior to agent setup() abstract method being called by the container
- ...and death
  - deregister() called after takedown()
Recap on JADE Architecture IV

Agent Communication Channel (ACC)
- JADE distributed, location-transparent messaging service
- asynchronous by default (agents autonomy)...
- ...but, can also provide for synchronicity (if required)
- compliant to FIPA ACL message format

Directory Facilitator (DF)
- JADE yellow pages service
- similar to the AMS agent...
  - one DF service (agent) for each JADE platform
  - always runs in the main container
- ...but, should be explicitly contacted by advertising and client agents upon need—public/subscribe pattern
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
Brief Recap

**JADE agents**

- instances of `jade.core.Agent`-derived classes
- single-threaded, *multitasking* computational model based on concurrent behaviours
- *asynchronous* communication model based on FIPA ACL messages
- FSM-like lifecycle with public methods to perform state transitions
- `jade.core.AID` class implements the globally unique naming service
  - agent name of the kind `<localname>@<platformname>`
  - pool of platform addresses, only used for *inter-platform* communications
Agents Lifecycle

Lifecycle methods

- `doActivate()` from SUSPENDED to where it was when `doSuspend()` was called
- `doDelete()` from either state to UNKNOWN
  - `doWait()` from ACTIVE to WAITING
- `doSuspend()` from ACTIVE or WAITING to SUSPENDED
  - `doWake()` from WAITING to ACTIVE
- `doMove()` from either state to TRANSIT
- `doClone()` same as `doMove()`
Agents Execution I

Starting agents

Agents are launched with command

```bash
$> java -cp ... jade.Boot ... -agents <name>:<class>
```

(or, from the RMA GUI)

1. the agent constructor is executed
2. the proper AID is given by the platform
3. registration to the AMS is done calling `register()` method
4. the agent is put in the ACTIVE state
5. `setup()` is executed...
6. ...then, behaviours `scheduling` begins
Agents Execution II

Stopping agents

Agents can be stopped by any of their behaviours calling the `doDelete()` method

1. prior to go into UNKNOWN state, the abstract method `takeDown()` is called by the platform to allow application specific clean-up
2. upon its completion, the agent is deregistered from the AMS calling `deregister()` method
3. the agent is put into the UNKNOWN state
4. the thread executing the agent is destroyed
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
**Brief Recap**

**JADE behaviours**

- instances of `jade.core.behaviours.Behaviour`-derived classes
- executed concurrently according to a *round-robin, non-preemptive* scheduler internal to agents—thus, hidden to programmers
- everything is still *single-threaded*...
  - method `action()` should be overridden to carry out the application-specific task
  - method `done()` should be overridden too to check such task termination condition
(Simplified) Behaviours Hierarchy

- **Behaviour**
  - **action()**
  - **done()**
  - **onStart()**
  - **onEnd()**
  - **block()**
  - **restart()**

- **CompositeBehaviour**
  - Models a complex task i.e. a task that is made up by composing a number of other tasks.

- **SimpleBehaviour**
  - Models a simple task i.e. a task that is not composed of sub-tasks.

- **OneShotBehaviour**
  - Models an atomic task (its done() method returns true)

- **CyclicBehaviour**
  - Models a cyclic task (its done() method returns false)

- **SequentialBehaviour**
  - Models a complex task whose sub-tasks are executed sequentially

- **ParallelBehaviour**
  - Models a complex task whose sub-tasks are executed concurrently

- ** FSMBehaviour**
  - Models a complex task whose sub-tasks corresponds to the activities performed in the states of a Finite State Machine
All behaviours are in package `jade.core.behaviours`

**SimpleBehaviour**

- **OneShotBehaviour**
  - method `action()` is executed only once...
  - ...hence, method `done()` always returns `true`

- **CyclicBehaviour**
  - method `done()` always returns `false`...
  - ...hence, method `action()` is executed forever—until agent death
Behaviour APIs II

CompositeBehaviour I

- **SequentialBehaviour**
  - method `addSubBehaviour()` to add *child* behaviours...
  - ...to be scheduled *sequentially*—method `done()` drives progress
  - the whole behaviour ends when the last child ends

- **ParallelBehaviour**
  - method `addSubBehaviour()` to add *child* behaviours...
  - ...to be scheduled *concurrently*
  - two termination conditions provided by default—through constants
    - WHEN_ALL children are done
    - WHEN_ANY child is done

Other conditions may be implemented by the programmer exploiting JADE APIs—see `checkTermination()` method

...
CompositeBehaviour II

- **FSM Behaviour**
  - **method registerState()** to add a child behaviour to the FSM
  - each child represents the activity to be performed within a state of the FSM
  - **method registerTransition()** to add a transition
    - the value returned by the **onEnd()** callback method is used to select the transition to fire
  - some of the children can be registered as **final states**...
  - ...hence, the whole behaviour terminates after the completion of any of them
Other behaviours

Many other very useful abstract behaviours exist, such as:

- **WakerBehaviour**
  - methods action() and done() are already implemented, so to execute abstract method onWake() when specified, then terminate

- **TickerBehaviour**
  - methods action() and done() are again already implemented, so to execute abstract method onTick() periodically as specified, then terminate when abstract method stop() is called

- ...

...refer to JADE APIs for the others.
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
Behaviours Scheduling Recap

```
setup()

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

YES

Get the next behaviour from the pool of active behaviours

b.action()

NO

b.done()?

YES

Remove currentBehaviour from the pool of active behaviours

takeDown()

NO

- Initializations
- Addition of initial behaviours

- Agent “life” (execution of behaviours)

- Clean-up operations
```

Highlighted in red the methods that programmers have to/can implement
Round-Robin, Non-Preemptive Scheduling I

**The setup() method**

By overriding the `setup()` method, JADE programmers ensure their agents have an initial pool of *ready-to-schedule* behaviours

- method `addBehaviour()` to add a behaviour (also usable elsewhere)
- method `removeBehaviour()` to remove one (better use it elsewhere...)

`setup()` serves to create instances of these behaviours and *link* them to the owner agent.

**Round-robin**

After initialisation, first behaviour from the *active behaviours* pool (*ready queue*) is scheduled for execution.
Some remarks

! Behaviours switch occurs only when the `action()` method of the currently scheduled behaviour returns
  → hence, when it is running no other behaviour can execute

! Behaviour removal from the scheduler pool occurs only when `done()` returns `true`
  → thus, if it returns `false` the behaviour is re-scheduled for next `round`

! `action()` is run from the beginning every time: there is no way to “stop-then-resume” a behaviour
  → therefore, the computational state must be explicitly managed by the programmer in instance variables
Round-Robin, Non-Preemptive Scheduling III

One more remark

- Programmers may need their agents to wait for something to happen—typically, a message to arrive
- Programmers may be lured to use method doWait() for the purpose...
  ! ...don’t do it!
    ! doWait() moves the agent to the WAITING state, where none of its behaviours can be executed!
→ Use method block() provided by any behaviour class instead, which allows to suspend only the calling behaviour
  → as soon as action() returns, the behaviour is moved to a special queue of blocked behaviours...
  → ...from which can be restored in the ready queue whenever any message arrives or by explicitly calling restart method
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
More on $\textit{ACL}$ Messages I

FIPA performatives

Performatives identify the type of $\textit{communicative act}$ carried out by the message—thus its semantics and expected response

- **CFP** (Call For Proposal) to obtain proposals about something
- **INFORM** to let someone know something
- **PROPOSE** to propose something
- **REQUEST** to ask for a service
- **SUBSCRIBE** to subscribe for notification about something
- **AGREE** to express consensus about something
- **REFUSE** to refuse a request

They are constants to be set for any $\textit{ACL}$ message exchanged by agents.
More on \textit{ACL} Messages II

\textbf{FIPA message syntax}

The syntax of an \textit{ACL} message is defined by FIPA to enable interoperability.

- \texttt{addReceiver()} to add a value to the \texttt{:receiver} slot
- \texttt{setContent()} to fill in the \texttt{:content} slot
- \texttt{setConversationId()} to fill in the \texttt{:conversation-id} slot
- \texttt{setEncoding()} to fill in the \texttt{:encoding} slot
- \texttt{setInReplyTo()} to fill in the \texttt{:in-reply-to} slot
- \texttt{setLanguage()} to fill in the \texttt{:language} slot
- \texttt{setOntology()} to fill in the \texttt{:ontology} slot
- \texttt{setSender()} to fill in the \texttt{:sender} slot

\ldots \ldots
Outline

1. Back to JADE Architecture

2. JADE Agents
   - Behaviours
   - Scheduling

3. JADE Messaging
   - ACL
   - JADE Communication APIs
Agents Communication Basics I

Sending messages

To send a message, an agent should:

1. create an \texttt{ACL} message
   
   \begin{verbatim}
   ACLMessage msg = new ACLMessage(ACLMessage.<performative>);
   \end{verbatim}

2. fill its (mandatory) fields

   \begin{itemize}
   \item msg.addReceiver(new AID(receiver));
   \item msg.setContent("<content>");
   \item ...
   \end{itemize}

3. call the \texttt{send()} method

   \begin{verbatim}
   send(msg);
   \end{verbatim}
Agents Communication Basics II

Replies to messages

To simplify answering, the ACLMessage class provides method createReply() to automagically set a number of ACL fields:

- :receiver
- :language, :ontology
- :conversation-id, :protocol
- :in-reply-to, :reply-with

Anyway, the programmer is free to overwrite such slots.
Who to talk with?

? How to find agents to talk to? When sending messages we must know the receiver AID

→ should we necessarily know it at compile-time?

! **JADE** provides several ways to get an agent ID:

- by using the agent **local name** (whenever known)
- from the RMA GUI
- by asking to the AMS
- by asking to the DF (we’ll see how to next lesson)
The simplest way to identify an agent is by its local name:

```java
... 
msg.addReceiver(new AID("myAgent", AID.ISLOCALNAME));
... 
```

JADE ACC will automagically associate to the given agent name its AID.

By simply launching the RMA with `java -cp ... jade.Boot -gui` you have a gui which displays all agents in the monitored JADE platform along with their AIDs.
Using the AMS

A much more comprehensive and flexible way to query JADE about existing agents is by interacting with the AMS service:

1. prepare a placeholder for agents with `AMSAgentDescription []`
   
   ```java
   agents = null;
   ```

2. configure some kind of “template” on agents with
   
   ```java
   AMSAgentDescription template = new AMSAgentDescription (...);
   ```

3. configure search parameters with `SearchConstraints c = new
   
   ```java
   SearchConstraints(...);
   ```

4. launch the search process with `agents = AMSService.search(this,
   
   ```java
   template, c);
   ```

5. collect AIDs with AID `aid = agents[i].getName();`
More on Agents Communication I

**JADE communication primitives**

- `send()` to asynchronously send a message—recipient is implicit
- `receive()` to asynchronously retrieve the first message from the mailbox (if any)
- `receive(MessageTemplate)` to perform a *selective receive*
- `blockingReceive()` to perform a *synchronous receive*
- `blockingReceive(long)` to perform a *timed synchronous receive*
- `blockingReceive(MessageTemplate)` to perform a selective, synchronous receive
- `blockingReceive(MessageTemplate, long)` to perform a timed, selective, synchronous receive
More on Agents Communication II

Receiving messages

Be careful when receiving messages:

- Method `blockingReceive()` **suspends all agent behaviours**, not only the calling one—due to synchronicity
  - call `receive()` then `block()` instead, so to resume the behaviour whenever any message arrives
  - call `blockingReceive()` only when you actually need to suspend all behaviours—e.g. during `setup()`

- Method `receive()` **removes** the first message from the mailbox, therefore it may “steal” someone else’s
  - use `jade.lang.acl.MessageTemplate` within a `receive()` to get only messages **matching a given pattern**
More on Agents Communication III

**Selective receive**

`jade.lang.acl.MessageTemplate` allows JADE agents to perform receive operations only *on a subset of their mailbox*, which is the subset with only those messages *matching* the given template.

**Hint**

When your agent should have *parallel negotiations* with several other agents, you should:

- create a `:conversation-id` string to *uniquely* identify messages
- by using the proper `MessageTemplate`, set-up a behaviour which only responds to messages with that particular `:conversation-id`
A set of static, *factory methods* are provided to build different kinds of template objects...

- `matchAll()` matches any *ACL* message
- `matchContent()` matches checked on :content slot
- `matchCustom(ACLMessage)` template built so to match the given *ACL* message
- `matchConversationId()` matches checked on :conversation-id slot
- `matchOntology()` matches checked on :ontology slot
- `matchSender()` matches checked on :sender slot

... ...
More on Agents Communication V

MessageTemplate APIs II

... along with elementary boolean operators to combine them into more complex patterns...

and() to build a template which is the intersection of two given templates

not() to build a template which is the union of two given templates

or() to build a template which is the negation of a given template

... and a non-static method to actually check matching:

match(ACLMessage) returns true if the given message matches the template upon which it is called
JADE: Java Agent DEvelopment Framework Basics

Stefano Mariani
s.mariani@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM – Università di Bologna a Cesena

Academic Year 2014/2015