Agents & Multi-Agent Systems with Jade

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Stefano Mariani Andrea Omicini

s.mariani@unibo.it andrea.omicini@unibo.it

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

Academic Year 2017/2018

1. JADE Overview
2. Getting Started with JADE
3. JADE Basics
4. JADE Advanced

Disclaimer

- all the material presented in these slides is rearranged by the authors starting 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 authors of these slides*

Next in Line. . .

1. JADE Overview
2. Getting Started with JADE
3. JADE Basics
4. JADE Advanced

What is Jade? I

- Jade [Bellifemine et al., 2007] stands for *Java Agent DEvelopment Framework*

  http://jade.tilab.com/

- Jade is a Java-based framework to develop agent-based applications in compliance with the FIPA specifications for interoperable, intelligent, multi-agent systems
- FIPA stands for *Foundation for Intelligent Physical Agents*

  http://www.fipa.org/

- FIPA is the IEEE Computer Society standards organisation that promotes agent-based technology and the interoperability of its standards with other technologies

What is JADE? II

JADE goals

As an agent-oriented middleware, JADE pursues the twofold goal of being

- a full-fledged FIPA-compliant *agent platform*
    - hence, it takes charge of all those application-independent aspects – such as agent lifecycle management, communication, distribution transparency, etc. – necessary to develop a MAS
- a simple yet comprehensive *agent development framework*
    - therefore, it provides Java developers a set of APIs to build their own customisations

Jade Main Ingredients

Java

- being fully implemented in Java, Jade is a notable example of a distributed, object-based, agent-oriented infrastructure

→ hence an interesting example about how to face a design/programming paradigm shift

FIPA

- being compliant to FIPA standards, Jade is a *complete* and *coherent* agent platform providing all the necessary facilities to deploy MAS

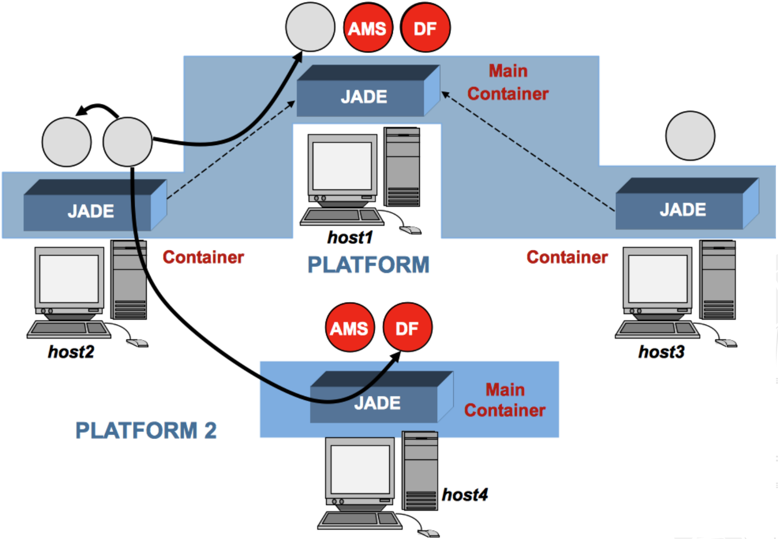
→ promoting interoperability

JADE Main Features

JADE offers. . .

- a distributed agent platform, where *distributed* means that a single (logical) JADE system can be split among different networked hosts
- transparent, distributed message passing service
- transparent, distributed naming service
- white pages & yellow pages discovering facilities
- intra-platform agent mobility (code & context, to some extent)
- debugging & monitoring graphical tools
- . . . and much more

Jade Architecture Overview



Focus on. . .

1. Jade Overview
   - Jade & FIPA
   - Jade Agents
   - Jade ACC
   - Jade Tools
2. Getting Started with Jade
3. Jade Basics
   - Jade Architecture
   - Jade Agents
   - Agent Behaviours
   - Jade Messaging
   - Jade Communication API
4. Jade Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in Jade
   - Jade Agents & Java Swing
   - Agent Mobility in Jade

FIPA Architecture I

Platforms & containers

- a FIPA agent platform can be split onto several hosts, provided that
  - each host acts as a container of agents—that is, provides a complete *runtime environment* for JADE agents execution(lifecycle management, message passing facilities, etc.)
  - there is (at least) one of these containers acting as the main container (actually, the first started)
  - the main container is responsible to maintain a *registry* of all other containers in the same JADE platform—through which agents can discover each other

→ hence, JADE promotes a peer-to-peer interpretation of a MAS

FIPA Architecture II

Agent Management System (AMS)

- for a given JADE platform, a single Agent Management System (AMS) exists, which
  - keeps track of all other agents in the same JADE platform—even those living in remote containers
  - should be contacted by JADE agents prior to any other action (they do not even exist until registered by the AMS)

→ hence, the AMS provides the white pages service—that is, a *location-transparent* naming service

FIPA Architecture III

Directory Facilitator (DF)

- a single Directory Facilitator (DF) exists for each JADE platform that
  - keeps track of all advertised services provided by all the agents in the same JADE platform
  - should be contacted by JADE agents who wish to publish their capabilities

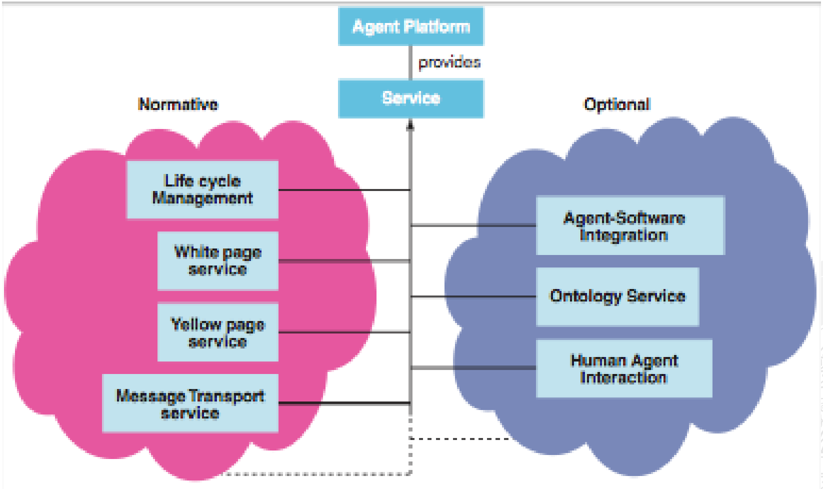
→ hence, the DF provides the default yellow pages service—according to the *publish/subscribe* paradigm

FIPA Architecture IV

Agent Communication Channel (ACC)

- for a given JADE platform, a *distributed message passing system* exists, called Agent Communication Channel (ACC), which
    - controls the exchange of messages within the JADE platform, be them local or remote
    - implements all the required facilities to provide for *asynchronous* communication
    - manages all aspects regarding FIPA $\mathcal{ACL}$ ($\mathcal{A}$gent $\mathcal{C}$ommunication $\mathcal{L}$anguage, [FIPA ACL, 2002]) message format, such as serialisation and deserialisation

FIPA Architecture V



FIPA required services

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

Agents in JADE I

An agent is a Java object executed by a Java thread

Since JADE is an *object-based middleware*, JADE agents are first of all Java objects

- user-defined agents must extend `jade.core.Agent` class, inheriting some ready-to-use methods

Agents in JADE II

An agent is more than a Java object

Despite being Java objects, JADE agents have a wide range of features promoting their autonomy

- each JADE agent is executed by a single Java thread (with an exception, though)
- all JADE agents have a globally unique name (agent ID, `AID`), which is (by default) the concatenation – by symbol '`@`' – of their *local name* and of the JADE platform name
- agents *business logic* must be expressed in terms of behaviours
- JADE agents communicate by exchanging FIPA $\mathcal{ACL}$ messages

FIPA Agent's Lifecycle I

FIPA states in a JADE agent lifecycle

Initiated the agent object has been built, but cannot do anything since it is not registered to the AMS yet—it has no `AID` even

Active the agent is registered to the AMS and can access all JADE features—in particular, it is executing its behaviour(s)

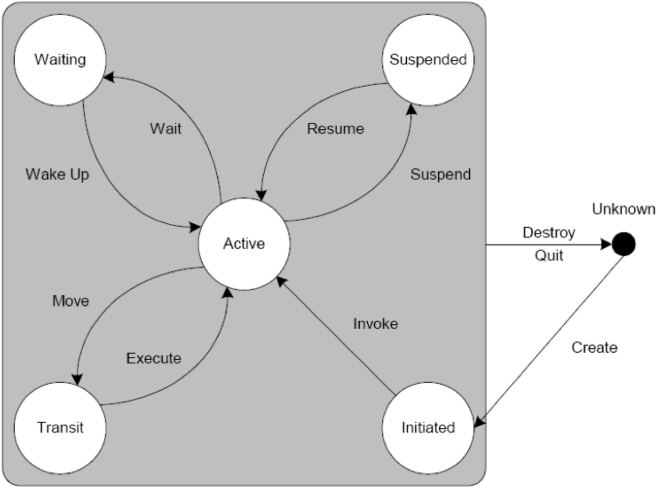
Waiting the agent is blocked, waiting for something to happen (and to react to)—typically, an $\mathcal{ACL}$ message

Suspended the agent is stopped, therefore none of its behaviours are being executed

Transit the agent has started a *migration* process—it will stay in this state until migration ends

Unknown the agent is dead—it has been deregistered to the AMS

FIPA Agent's Lifecycle II



FIPA agent lifecycle

Agent Behaviours I

Why behaviours?

- by definition, agents are autonomous entities, therefore they should act independently and concurrently w.r.t. one another
- the need for *efficiency* drives toward the execution of Jade agents as a single Java thread each

! however, agents need to perform complex activities, possibly composed by multiple tasks—even concurrently

? how can such contrasting requirements be satisfied altogether?

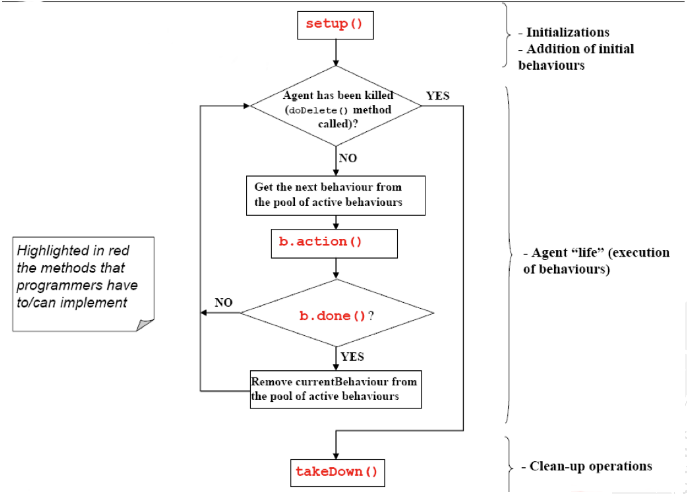
Agent Behaviours II

Concurrent agent activities with behaviours

- a behaviour can be seen as *an activity to perform with the goal of completing a task*
- a behavior can represent a *proactive* activity – started by the agent on its own – as well as a *reactive* activity—performed in response to some events (timeouts, messages, etc.)

! JADE implements behaviours as Java objects, which are executed concurrently (still by a single Java thread) using a non-preemptive, round-robin scheduler (internal to the agent class but hidden to the programmer)

Agent Behaviours III


setup()
Agent has been killed (doDelete() method called)?
YES
NO
Get the next behaviour from the pool of active behaviours
b.action()
b.done()?
NO
YES
Remove currentBehaviour from the pool of active behaviours
takeDown()
- Initializations
- Addition of initial behaviours
- Agent "life" (execution of behaviours)
- Clean-up operations
Highlighted in red the methods that programmers have to/can implement

Jade non-preemptive scheduling policy

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

The Agent Communication Channel (ACC) I

JADE messaging runtime

Following the FIPA specification, JADE agents communicate via asynchronous message passing

- each agent has a *message queue* (a sort of mailbox) where the JADE ACC delivers $\mathcal{ACL}$ messages sent by other agents
- whenever a new entry is added to the mailbox, the receiving agent is *notified*—it does not need to block nor to continuously ask either

! *if* and *when* the agent actually processes a message is up to the agent itself (or the programmer)—for the sake of agents autonomy

The Agent Communication Channel (ACC) II

$\mathcal{ACL}$-compliant messages

- to *understand* each other, it is crucial that agents agree on the format and semantics of the messages they exchange
- hence, an $\mathcal{ACL}$ message contains

`:sender` who sends the message—automatically set

`:receiver` who the message targets—may be many

`:performative` the name of the communication act the agents want to carry out—constrained by a FIPA ontology

`:content` the actual information conveyed by the message

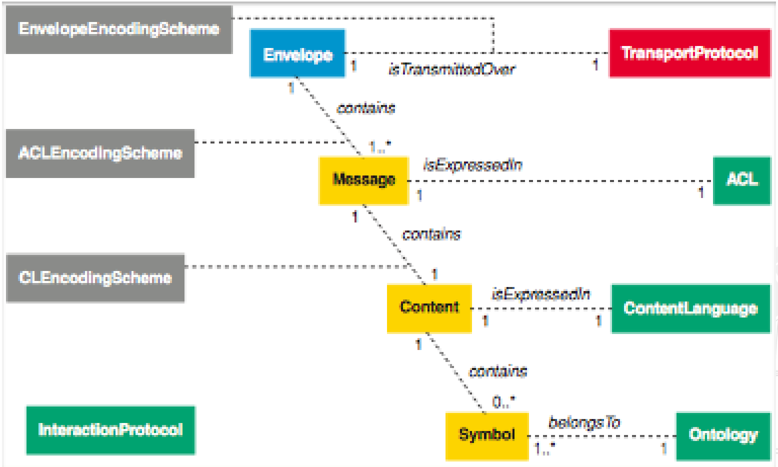
`:language` the syntax used to encode the `:content`

`:ontology` the semantics upon which the `:content` relies

⋮ ⋮

others fields. . .

The Agent Communication Channel (ACC) III



FIPA communication model abstractions

The Agent Communication Channel (ACC) IV

JADE communication primitives

- to interact, JADE agents have a number of ready-to-use methods

  send to send a message to a recipient agent

  receive to asynchronously retrieve the first message in the mailbox (if any)

  timed receive to perform a *timed*, synchronous receive on the mailbox—timeout causes agent to resume execution

  selective receive to retrieve a message from the mailbox which *matches* a given *message template*—message queue order is bypassed

! all the above methods are distribution-transparent, that is, they choose the proper address and transport mechanism based upon sender and receiver locations

Focus on. . .

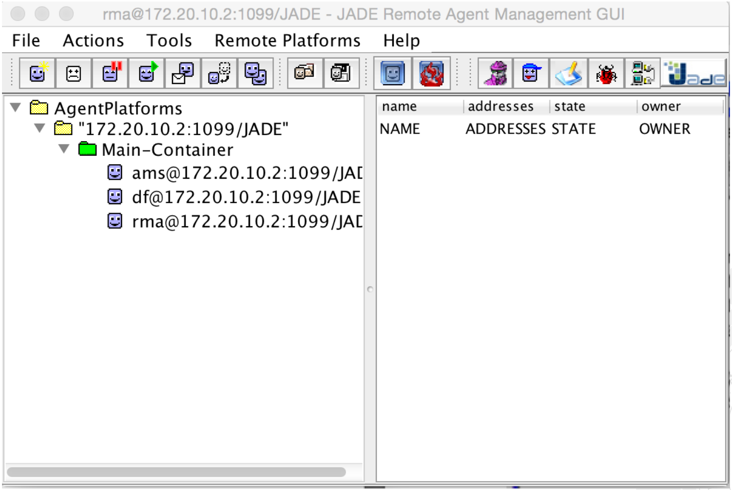
1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

JADE Management Tools I

Remote Monitoring Agent (RMA)

- the Remote Monitoring Agent (RMA) enables the control of the life cycle of the agent platform and of all the registered (possibly, remote) agents
- in particular, the RMA makes it possible to
  - start, stop, kill agents
  - send them messages
  - clone and/or migrate agents
  - add, remove, shutdown (remote) platforms
  - . . . and much more

Jade Management Tools II



rma@172.20.10.2:1099/JADE - JADE Remote Agent Management GUI

- AgentPlatforms
  - "172.20.10.2:1099/JADE"
    - Main-Container
      - ams@172.20.10.2:1099/JAD
      - df@172.20.10.2:1099/JADE
      - rma@172.20.10.2:1099/JAD

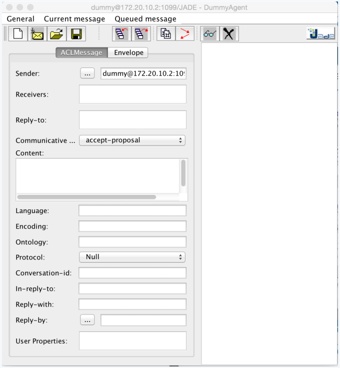
name	addresses	state	owner
NAME	ADDRESSES	STATE	OWNER

Jade RMA GUI

JADE Management Tools III

Dummy Agent

The Dummy Agent allows a human user to interact with JADE agents by sending, inspecting, recording custom $\mathcal{ACL}$ messages



dummy@172.20.10.2:1099/JADE - DummyAgent

Sender: dummy@172.20.10.2:10!

Receivers:

Reply-to:

Communicative ... accept-proposal

Content:

Language:

Encoding:

Ontology:

Protocol: Null

Conversation-id:

In-reply-to:

Reply-with:

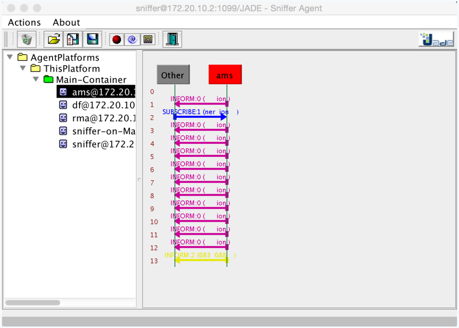
Reply-by:

User Properties:

Jade Management Tools IV

Sniffer Agent

The Sniffer Agent allows a user to *sniff* an agent or a group of agents, which means that every message directed to/from that agent / agent group is tracked and displayed

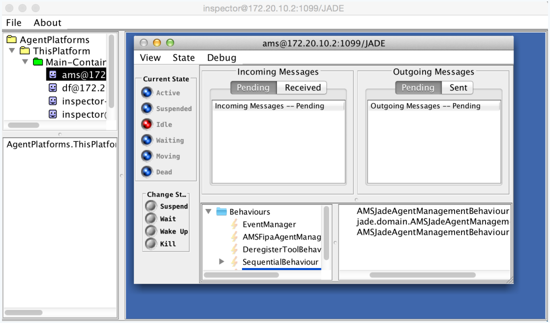


```
0
1  INFORM:0 (     ion )
2  SUBSCRIBE:1 (ner  ion    )
3  INFORM:0 (     ion )
4  INFORM:0 (     ion )
5  INFORM:0 (     ion )
6  INFORM:0 (     ion )
7  INFORM:0 (     ion )
8  INFORM:0 (     ion )
9  INFORM:0 (     ion )
10 INFORM:0 (     ion )
11 INFORM:0 (     ion )
12 INFORM:0 (     ion )
13 INFORM:2 (683  688    )
```

JADE Management Tools V

Introspector Agent

The Introspector Agent allows to monitor and control both the queue of sent and received messages as well as the queue of behaviours—including executing them step-by-step



Next in Line. . .

1 JADE Overview

2 Getting Started with JADE

3 JADE Basics

4 JADE Advanced

JADE: Where & What

JADE web page

Go to `http://jade.tilab.com`, then

1. hover over “Download” in the upper navigation bar, then click “Jade”
2. click “Continue”
3. scroll down—and yes, you agree
4. download (at least) `jadeBin` (or, `jadeAll` if you like more)

Running JADE I

Requirements

The only software requirement to execute JADE is JRE version 6 or later

Classpath yes

- add the `jade.jar` archive in `jade/lib/` to your JVM classpath
  - you are supposed to know how to do that
- open up your command prompt (wherever you can run `java`), and type
  - `java jade.Boot -gui &` to launch JADE main container with RMA attached
  - `java jade.Boot -container [-container-name` *`name`*`] &` to launch a *peripheral* (non-main) container (possibly with a given *name*) connected to the same JADE platform (previous main container)

Running JADE II

Classpath no

- open up your command prompt and navigate to `jade/` folder, then type
  - `java -cp lib/jade.jar jade.Boot -gui &` to launch JADE main container with RMA attached
  - `java -cp lib/jade.jar jade.Boot -container [-container-name name] &` to launch a *peripheral* (non-main) container (possibly with a given *name*) connected to the same JADE platform (previous main container)

Running JADE III

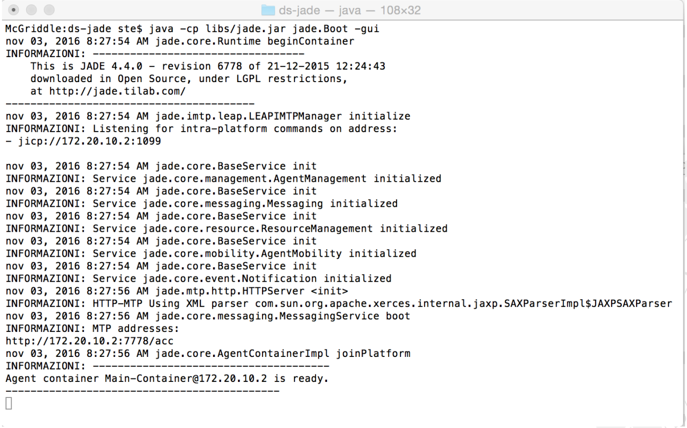
Default ports

port 1099 is the default main container listening TCP port for intra-platform (remote) communications

port 7778 is the default main container listening port for inter-platform communications (HTTP is the default MTP)

What to Expect I

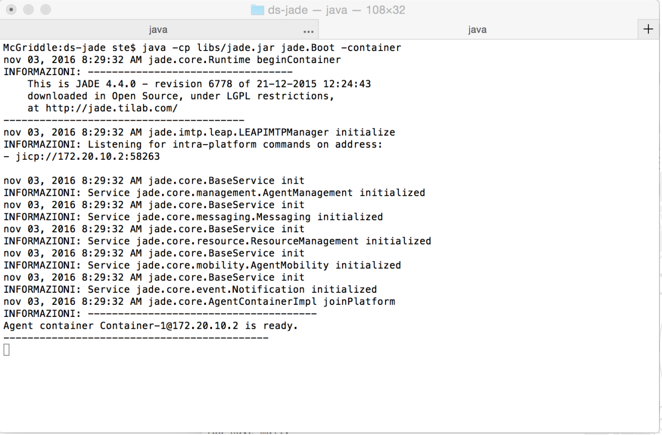
- if you launched the main container, the RMA GUI should show up and something like this should appear on the command prompt



```
McGriddle:ds-jade ste$ java -cp libs/jade.jar jade.Boot -gui
nov 03, 2016 8:27:54 AM jade.core.Runtime beginContainer
INFORMAZIONI: ----------------------------------
    This is JADE 4.4.0 - revision 6778 of 21-12-2015 12:24:43
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
nov 03, 2016 8:27:54 AM jade.imtp.leap.LEAPIMTPManager initialize
INFORMAZIONI: Listening for intra-platform commands on address:
- jicp://172.20.10.2:1099

nov 03, 2016 8:27:54 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.management.AgentManagement initialized
nov 03, 2016 8:27:54 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.messaging.Messaging initialized
nov 03, 2016 8:27:54 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.resource.ResourceManagement initialized
nov 03, 2016 8:27:54 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.mobility.AgentMobility initialized
nov 03, 2016 8:27:54 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.event.Notification initialized
nov 03, 2016 8:27:56 AM jade.mtp.http.HTTPServer <init>
INFORMAZIONI: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
nov 03, 2016 8:27:56 AM jade.core.messaging.MessagingService boot
INFORMAZIONI: MTP addresses:
http://172.20.10.2:7778/acc
nov 03, 2016 8:27:56 AM jade.core.AgentContainerImpl joinPlatform
INFORMAZIONI: ----------------------------------------
Agent container Main-Container@172.20.10.2 is ready.
--------------------------------------------
```

What to Expect II

- if you launched a peripheral container, the RMA GUI should self-update and something like this should appear on the command prompt



```
McGriddle:ds-jade ste$ java -cp libs/jade.jar jade.Boot -container
nov 03, 2016 8:29:32 AM jade.core.Runtime beginContainer
INFORMAZIONI: ----------------------------------
    This is JADE 4.4.0 - revision 6778 of 21-12-2015 12:24:43
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
nov 03, 2016 8:29:32 AM jade.imtp.leap.LEAPIMTPManager initialize
INFORMAZIONI: Listening for intra-platform commands on address:
- jicp://172.20.10.2:58263

nov 03, 2016 8:29:32 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.management.AgentManagement initialized
nov 03, 2016 8:29:32 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.messaging.Messaging initialized
nov 03, 2016 8:29:32 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.resource.ResourceManagement initialized
nov 03, 2016 8:29:32 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.mobility.AgentMobility initialized
nov 03, 2016 8:29:32 AM jade.core.BaseService init
INFORMAZIONI: Service jade.core.event.Notification initialized
nov 03, 2016 8:29:32 AM jade.core.AgentContainerImpl joinPlatform
INFORMAZIONI: --------------------------------------
Agent container Container-1@172.20.10.2 is ready.
--------------------------------------------
```

Some Notes

- option `-name` when launching the main container lets you give a name to the JADE platform
- option `-container-name` when launching a peripheral container lets you choose a name for that container
- options `-local-host` / `-local-port` when launching the main container let you choose a custom host / listening port for the JADE platform
- options `-host` / `-port` when launching a peripheral container let you specify where to find the remote main container to register to
- option `-agents` *`name`*`:`*`full-class-name`* in conjunction with `-container` launches the agent implemented in class *`full-class-name`* on the newly-created peripheral container
- for other options, please refer to the JADE documentation [Bellifemine et al., 2010b]

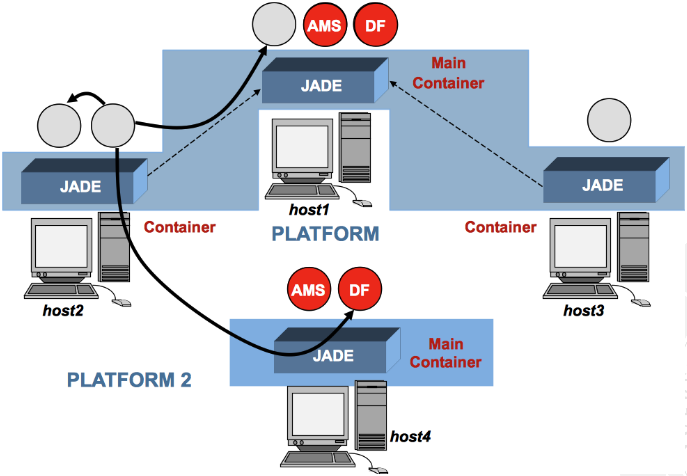
Next in Line...

1. Jade Overview
2. Getting Started with Jade
3. **Jade Basics**
4. Jade Advanced

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

JADE Architecture: Recap I



JADE Architecture: Recap 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 automatically register themselves to the (default/given) main container
- one single JVM executed per host/platform (2 JADE on the same host are 2 JVM)

JADE Architecture: Recap III

Agent Management System (AMS)

- JADE white pages service
- *one* AMS service (agent) for *each* JADE platform
- always runs in the main container
- is contacted (automatically) 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()`

Jade Architecture: Recap IV

Agent Communication Channel (ACC)

- Jade *distributed*, *location-transparent* messaging service
- asynchronous by default (uncoupling for agents autonomy)
- also supporting *synchronous communication*, if required
- compliant to FIPA $\mathcal{ACL}$ 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
- except that it should be explicitly contacted by *advertising* and *client* agents upon need—*public/subscribe* pattern

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - **JADE Agents**
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

JADE Agents: 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 $\mathcal{ACL}$ 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

```
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

Focus on. . .

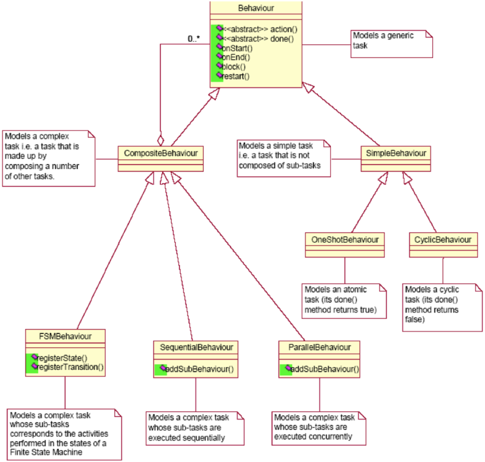
1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

Jade Behaviours: 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



Models a generic task

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

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

Models an atomic task (its done() method returns true)

Models a cyclic task (its done() method returns false)

Models a complex task whose sub-tasks corresponds to the activities performed in the states of a Finite State Machine

Models a complex task whose sub-tasks are executed sequentially

Models a complex task whose sub-tasks are executed concurrently

Behaviour API I

jade.core.behaviours

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 API II

CompositeBehaviour: sequential vs. parallel

- 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 API—see checkTermination() method

Behaviour API III

CompositeBehaviour: FSM

- FSMBehaviour
  - 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

Behaviour API IV

Other behaviours

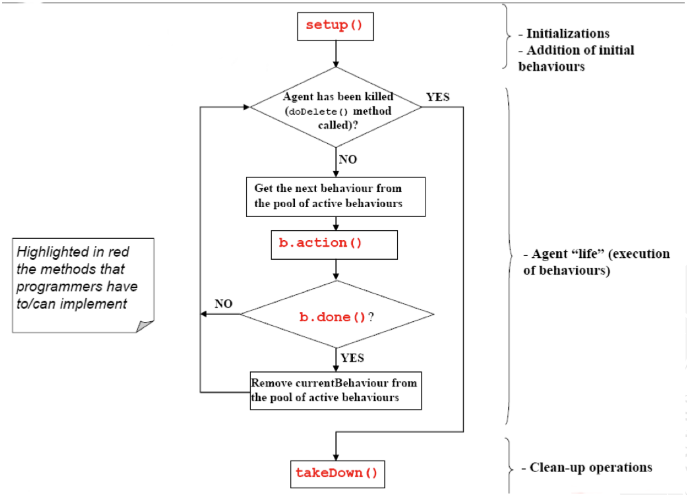
Many other very useful abstract behaviours exist, such as

- `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
- ...

Please refer to the JADE Programmer's Guide for the others

[Bellifemine et al., 2010a]

Behaviours Scheduling: Recap


setup()
- Initializations
- Addition of initial behaviours
Agent has been killed (doDelete() method called)?
YES
NO
Get the next behaviour from the pool of active behaviours
b.action()
b.done()?
NO
YES
Remove currentBehaviour from the pool of active behaviours
- Agent "life" (execution of behaviours)
takeDown()
- 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

Round-Robin, Non-Preemptive Scheduling II

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

Examples in `ds.lab.jade.behaviours.*`

Open a command prompt and position yourself into folder `ds-jade/`

- `java -cp libs/jade.jar:bin/ jade.Boot -gui -agents ste:ds.lab.jade.behaviours.SimpleBehavioursAgent`
- `java -cp libs/jade.jar:bin/ jade.Boot -gui -agents ste:ds.lab.jade.behaviours.CompositeBehavioursAgent`
- `java -cp libs/jade.jar:bin/ jade.Boot -gui -agents ste:ds.lab.jade.behaviours.FSMLikeAgent`

On Windows, substitute ":" with ";", and "/" with "\"

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

More on $\mathcal{ACL}$ Messages I

FIPA performatives

Performatives identify the type of *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 $\mathcal{ACL}$ message exchanged by agents

More on $\mathcal{ACL}$ Messages II

FIPA message syntax

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

`addReceiver()` to add a value to the `:receiver` slot

`setContent()` to fill in the `:content` slot

`setConversationId()` to fill in the `:conversation-id` slot

`setEncoding()` to fill in the `:encoding` slot

`setInReplyTo()` to fill in the `:in-reply-to` slot

`setLanguage()` to fill in the `:language` slot

`setOntology()` to fill in the `:ontology` slot

`setSender()` to fill in the `:sender` slot

... ...

Focus on. . .

1. Jade Overview
   - Jade & FIPA
   - Jade Agents
   - Jade ACC
   - Jade Tools
2. Getting Started with Jade
3. Jade Basics
   - Jade Architecture
   - Jade Agents
   - Agent Behaviours
   - Jade Messaging
   - Jade Communication API
4. Jade Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in Jade
   - Jade Agents & Java Swing
   - Agent Mobility in Jade

Agents Communication Basics I

Sending messages

In order to send a message, an agent should

1. create an $\mathcal{ACL}$ message
   - `ACLMessage msg = new ACLMessage(ACLMessage.<performative>);`
2. fill its (mandatory) fields
   - `msg.addReceiver(new AID(receiver));`
   - `msg.setContent("<content>");`
   - ...
3. call the `send()` method
   - `send(msg);`

Agents Communication Basics II

Replying to messages

In order to simplify answering, the ACLMessage class provides method createReply() to automatically set a number of $\mathcal{ACL}$ fields

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

Anyway, the programmer is free to overwrite such slots

Agents Communication Basics III

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)

Agents Communication Basics IV

JADE local names

The simplest way to identify an agent is by its local name

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

JADE ACC will automatically associate to the given agent name its `AID`

JADE RMA

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 `AID`s

Agents Communication Basics V

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 [] agents = null;
   ```

2. configure some kind of "template" on agents with

   ```
   AMSAgentDescription template = new AMSAgentDescription (...);
   ```

3. configure search parameters with

   ```
   SearchConstraints c = new SearchConstraints(...);
   ```

4. launch the search process with

   ```
   agents = AMSService.search(this, 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

One need to 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 a JADE agent is required to have *parallel negotiations* with several other agents, one 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`

More on Agents Communication IV

MessageTemplate API I

A set of static, *factory methods* are provided to build different kinds of template objects

`matchAll()` matches any $\mathcal{ACL}$ message

`matchContent()` match checked on `:content` slot

`matchCustom(ACLMessage)` template built so to match the given $\mathcal{ACL}$ message

`matchConversationId()` match checked on `:conversation-id` slot

`matchOntology()` match checked on `:ontology` slot

`matchSender()` match checked on `:sender` slot

... ...

More on Agents Communication V

`MessageTemplate` API 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
- `or()` to build a template which is the *union* of two given templates
- `not()` 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

Examples in `ds.lab.jade.messaging.*` I

Open a command prompt and position yourself into folder `ds-jade/`

- `java -cp libs/jade.jar:bin/ jade.Boot -gui -agents ste:ds.lab.jade.messaging.PingPongAgent`
  - from RMA GUI launch the “DummyAgent”
  - right-click on blank text field next to “Receivers” and left-click “Add”
  - check “NAME” checkbox and digit “ste” (or whichever name you gave to the ping pong agent) then “Ok”
  - select “propose” as the “Communicative act” and fill “Content” with either “ping” or “pong”
  - fill “Ontology” with “ping-pong”
  - click “Send” (envelope icon)
  - select messages from the box on the right (most recent at the top) and click “View” (glasses icon) to inspect message content

On Windows, substitute “:” with “;” an “/” with “\”

Examples in ds.lab.jade.messaging.* II

- `java -cp libs/jade.jar:bin/ jade.Boot -gui -agents calculator:ds.lab.jade.messaging.calculator.CalculatorAgent &` (agent name MUST BE “calculator”)
- `java -cp libs/jade.jar:bin/ jade.Boot -container -agents ste:ds.lab.jade.messaging.calculator.ClientAgent &`

On Windows, substitute “&” with “start /B” (placed as first command)

Next in Line. . .

1 Jade Overview

2 Getting Started with Jade

3 Jade Basics

4 Jade Advanced

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

Directory Facilitator (DF): Recap

What we already know

By default, a singleton Directory Facilitator (DF) exists for each JADE platform, which

- provides the yellow pages service by keeping track of published services provided by advertising agents—be them local or remote
- should be *explicitly* contacted by JADE agents who wish to advertise their capabilities—both to submit an advertisement and to remove it
- supports the publish/subscribe pattern by offering a *notification service*
- can be *federated* with other DFs to implement a *truly distributed* yellow pages service

DF API I

What's new

The DF service is implemented as a JADE agent – as the AMS – in class `jade.domain.DFService`

- ! being JADE DF FIPA-compliant, *all interactions with the DF must follow* FIPA*'s standards*:
  - → interaction protocols taken from package `jade.proto`
  - → $\mathcal{ACL}$ messages must adhere to the `FIPAManagementVocabulary` (ontology) in package `jade.domain.FIPAAgentManagement`
  - → $\mathcal{ACL}$ messages content must adhere to the `SL0Vocabulary` in package `jade.content.lang.sl`

...

DF API II

Jade helps us

...

- static methods are provided to automatically build *semantically-correct* $\mathcal{ACL}$ messages:
  - `createRequestMessage()` to request the execution of a `fipa-agent-management` ontology action by the DF
  - `createSubscriptionMessage()` to request subscription for a given `DFAgentDescription` template
  - `decodeResult()` to process the content of the final message received as a result of `search()` operation
  - `decodeNotification()` to process the content of a notification message received as a consequence of a previous subscription

...

DF API III

JADE helps us even more

...

- to ease developer's work, a set of static methods embedding such interaction protocols are provided by class `DFService`
  - `register()` called by an agent wishing to advertise a service
  - `deregister()` called by an agent who no longer offers a previously advertised service
  - `search()` called by *client* agents looking for a service to exploit

! be careful 'cause all these methods are blocking calls, therefore *every activity of the agent is suspended* until success or failure of the call

→ if you need *asynchronous* interactions, go for the FIPA protocols approach

DF Entries Syntax I

The DFAgentDescription class (DFD)

The DFD is an entry in the DF, thus must contain (at least):

- the agent ID
- the set of services the agent wishes to advertise, in the form of `ServiceDescription`
- the set of *ontologies*, *protocols* and *languages* the agent is able to support/understand

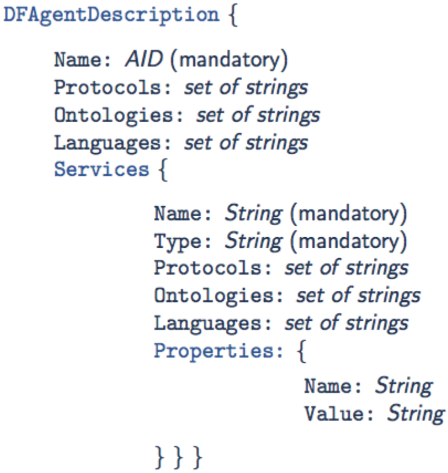
DF Entries Syntax II

The ServiceDescription class (SD)

The SD is a descriptor of the service the agent wishes to publish to the DF, thus must contain (at least):

- the service *name*
- the service *type*
- the set of *ontologies* and *languages* whose knowledge is required to exploit the service
- a number of *service-specific* properties

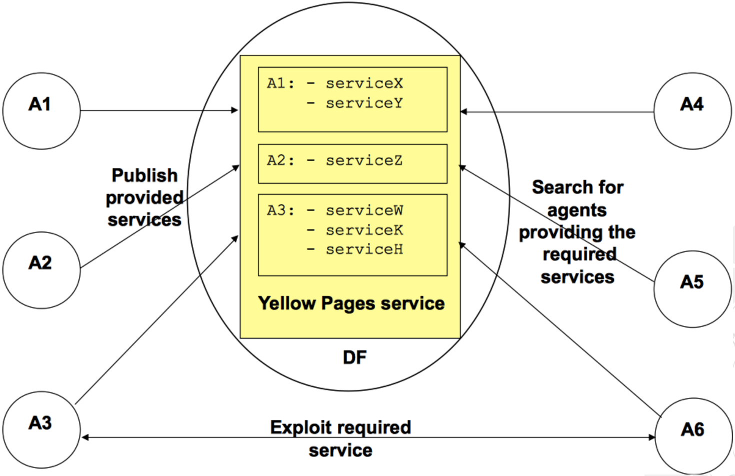
DF Entries Syntax III



```
DFAgentDescription {

    Name: AID (mandatory)
    Protocols: set of strings
    Ontologies: set of strings
    Languages: set of strings
    Services {

            Name: String (mandatory)
            Type: String (mandatory)
            Protocols: set of strings
            Ontologies: set of strings
            Languages: set of strings
            Properties: {

                        Name: String
                        Value: String

            } } }
```

Pseudo-code view of a DF entry

Using DF API I



Using DF API II

Registering to the DF

1. instantiate a DFAgentDescription object

   →
   ```
   DFAgentDescription dfd = new DFAgentDescription();
   ```

2. fill in (at least) its Name field with the advertising agent AID

   →
   ```
   dfd.setName(getAID());
   ```

3. instantiate a ServiceDescription object

   →
   ```
   ServiceDescription sd = new ServiceDescription();
   ```

4. fill in (at least) its Name and Type fields with meaningful strings

   →
   ```
   sd.setType("buyer");
   sd.setName("online trad");
   ```

5. add the ServiceDescription to the DFAgentDescription

   →
   ```
   dfd.addServices(sd);
   ```

6. call `DFService.register(this, dfd);`

Using DF API III

Deregistering from the DF

Since dead agent's AIDs are automatically removed *solely from the AMS*, it is a good practice to deregister agents upon death

- a good place where to do so is in `takeDown()` callback method
  - → `DFService.deregister(this);`
- ! keep in mind that each agent is allowed to have *only one entry* in the DF
  - → each attempt to register an already registered agent throws an exception

Using DF API IV

Browsing the DF I

Client agents may query the DF to know if any agent offers the services they are looking for and then acquire their AIDs:

1. create a DFD (with no AID, obviously. . .) filling its fields with the properties you look for

   →
   ```
   DFAgentDescription dfd = new DFAgentDescription();
   ServiceDescription sd = new ServiceDescription();
   sd.setType("buyer");
   dfd.addServices(sd);
   ```

2. specify as `SearchConstraints` that you want to get *all* the agents offering the service (skip this if you need only one)

   →
   ```
   SearchConstraints all = new SearchConstraints();
   all.setMaxResults(new Long(-1));
   ```

. . .

Using DF API V

Browising the DF II

...

1. launch the search process (skip last parameter if skipped previous point)
   → `DFAgentDescription[] result = DFService.search(this, dfd, all);`
2. extract the AID(s) from the results set
   →
```
AID[] providers = new AID[result.length];
for (int i = 0; i < result.length; i++) {
  providers[i] = results[i].getName();
}
```

Using DF API VI

! check the `ds.lab.jade.bookTrading` example for the whole code

Launching `ds.lab.jade.bookTrading`

Open a command prompt and position yourself into folder `ds-jade/`

-
  ```
  java -cp libs/jade.jar:bin/ jade.Boot -gui -agents
  seller:ds.lab.jade.bookTrading.BookSellerAgent &
  ```

-
  ```
  java -cp libs/jade.jar:bin/ jade.Boot -container -agents
  buyer:ds.lab.jade.bookTrading.BookBuyerAgent &
  ```

On Windows, substitute "&" with "start /B" (placed as first command)

The example should work anyway regardless of the order in which agents are launched and regardless of how many buyers and sellers you launch (at least one), provided they have different names

Using DF API VII

Subscribing to the DF I

JADE agents can ask the DF to notify them *as soon as* a given service is advertised

1. as usual, create a DFD suited for the service you wish to be notified about. . .

   →
   ```
   DFAgentDescription dfd = new DFAgentDescription();
   ServiceDescription sd = new ServiceDescription();
   sd.setType(...);
   dfd.addServices(sd);
   ```

2. . . .configure your chosen `SearchConstraints` (if you please). . .

   →
   ```
   SearchConstraints sc = new SearchConstraints();
   sc.setMaxResults(new Long(1));
   ```

. . .

Using DF API VIII

Subscribing to the DF II

. . .

1. . . . then, perform your subscription

   →
   ```
   send(
     DFService.createSubscriptionMessage(this, getDefaultDF(), dfd, sc)
   );
   ```

Now the DF will send an `ACLMessage.INFORM` to the subscribed agent *whenever* an agent matching the supplied description registers

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - **FIPA Interaction Protocols in JADE**
   - JADE Agents & Java Swing
   - Agent Mobility in JADE

Interaction Protocols I

Protocol according to FIPA

> *Predefined sequences of messages that can be reused in different domains to implement a given interaction*

! some kind of "design pattern" for communications

The jade.proto package

jade.proto contains behaviours implementing both the initiator and responder roles in most common interaction protocols

- managing the flow of messages and checking that it is *consistent to the protocol*
- providing *callback methods* that can be overridden to take the necessary actions when a message is received

Interaction Protocols II

(Some) Protocol classes I

`AchieveRE[Initiator/Responder]` factorization of all the FIPA *Request-like* interaction protocols[a], that is, those in which the initiator aims to achieve a RE (Rational Effect) and needs to verify if it has been achieved or not

`ContractNet[Initiator/Responder]` allows the initiator to send a *Call for Proposal* to a set of responders, evaluate their proposals and then accept the preferred one (or even reject all of them)

... ...

[a] such as `FIPA-Request`, `FIPA-query`, `FIPA-Request-When`, `FIPA-recruiting`, `FIPA-brokering`.

Interaction Protocols III

(Some) Protocol classes II

... ...

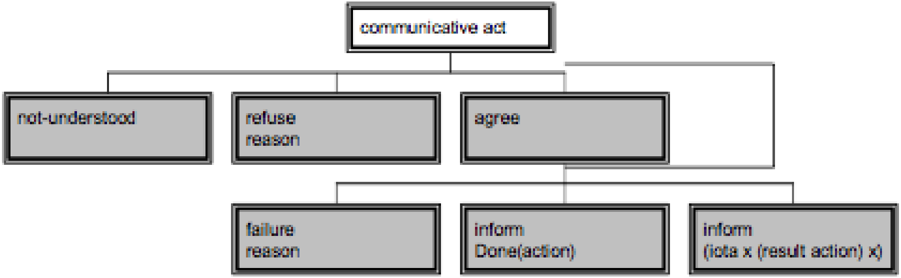
`Propose[Initiator/Responder]` allows the initiator to send a `PROPOSE` message to the participants indicating *its will to perform some action if they agrees*. The participants responds by either accepting or rejecting such proposal, then the initiator either carries out the action or not accordingly

`Subscription[Initiator/Responder]` allows the initiator to *subscribe* to a target agent for certain kind of events. If the participant agrees, it communicates all content *matching the subscription condition* using an `INFORM-RESULT`

... ...

. . . refer to JADE API for the others

Achieve Rational Effect (AchieveRE) I



FIPA AchieveRE protocol message flow

Achieve Rational Effect (AchieveRE) II

AchieveREInitiator

Initiator role for FIPA request-like protocols

- constructed by passing the protocol-starting $\mathcal{ACL}$ message
  - ! be sure to set the protocol field of the ACLMessage with the proper constant taken from FIPANames.InteractionProtocols in package jade.domain
- to be extended by overriding its handle[...] *callback* methods, which provide hooks to handle all the states of the protocol
  - e.g. handleAgree(), handleInform(), ...
  - ! be aware of the functioning of callbacks such as handleOutOfSequence(), handleAllResponses(), handleAllResultNotifications—refer to JADE *programmer's guide*
- manages an *expiration timeout* expressed by the value of the reply-by slot in ACLMessage
  - ! as defined by FIPA, such timeout refers to the *first response*: second response timeouts can be managed "by hand"

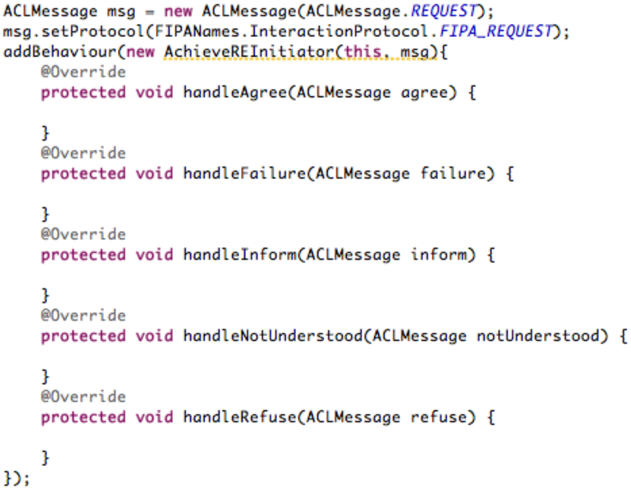
Achieve Rational Effect (AchieveRE) III

AchieveREResponder

Responder role for FIPA request-like protocols

- constructed by passing the MessageTemplate describing $\mathcal{ACL}$ messages we'd like to manage
    - ! method createMessageTemplate is provided to create templates for each interaction protocol
- to be extended by overriding its handle/prepare[...] *callback* methods, which provide hooks to handle all the states of the protocol
    - handleRequest() to reply to first initiator message
    - prepareResultNotification() to send the final response about the RE achieved
    - ...

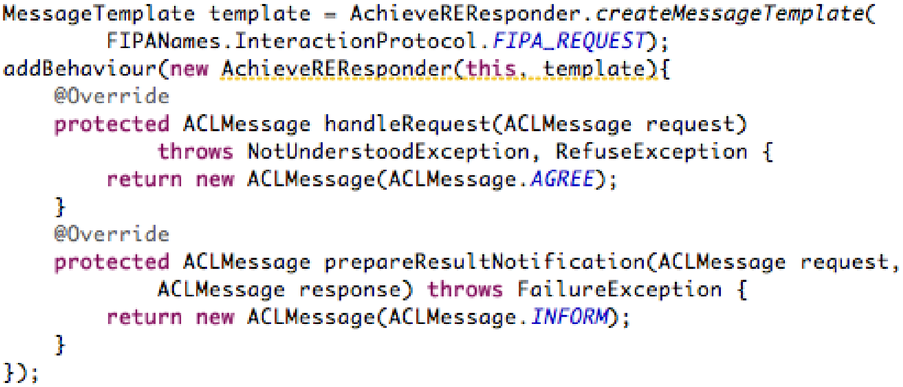
Achieve Rational Effect (AchieveRE) IV



```
ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
msg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
addBehaviour(new AchieveREInitiator(this, msg){
    @Override
    protected void handleAgree(ACLMessage agree) {

    }
    @Override
    protected void handleFailure(ACLMessage failure) {

    }
    @Override
    protected void handleInform(ACLMessage inform) {

    }
    @Override
    protected void handleNotUnderstood(ACLMessage notUnderstood) {

    }
    @Override
    protected void handleRefuse(ACLMessage refuse) {

    }
});
```

JADE `AchieveREInitiator`

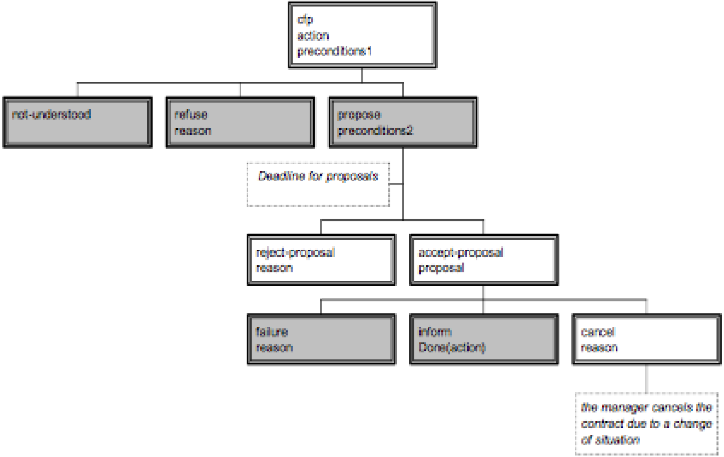
Achieve Rational Effect (AchieveRE) V



```
MessageTemplate template = AchieveREResponder.createMessageTemplate(
        FIPANames.InteractionProtocol.FIPA_REQUEST);
addBehaviour(new AchieveREResponder(this, template){
    @Override
    protected ACLMessage handleRequest(ACLMessage request)
            throws NotUnderstoodException, RefuseException {
        return new ACLMessage(ACLMessage.AGREE);
    }
    @Override
    protected ACLMessage prepareResultNotification(ACLMessage request,
            ACLMessage response) throws FailureException {
        return new ACLMessage(ACLMessage.INFORM);
    }
});
```

JADE AchieveREResponder

Contract Nets I



cfp
action
preconditions1

not-understood

refuse
reason

propose
preconditions2

Deadline for proposals

reject-proposal
reason

accept-proposal
proposal

failure
reason

inform
Done(action)

cancel
reason

the manager cancels the contract due to a change of situation

FIPA Contract Net protocol

Contract Nets II

ContractNetInitiator

Initiator role for FIPA contract-net protocol

- constructed by passing the protocol-starting $\mathcal{ACL}$ message
  - ! again, use `FIPANames.InteractionProtocols` to set the `protocol` field of the `ACLMessage`
- to be extended by overriding its `handle[...]` *callback* methods
  - e.g. `handlePropose()`, `handleInform()`, ...
  - ! be sure to implement `handleAllResponses()` by adding to the `acceptances` `Vector` all the `ACLMessage.ACCEPT_PROPOSAL` $\mathcal{ACL}$ messages to send
- manages the *expiration timeout*
  - ! again, `reply-by` timeout refers to the *first response*
  - ! late answers *are not consumed*, thus remain in the agent message box

Contract Nets III

`ContractNetResponder`

Responder role for FIPA contract-net protocol

- constructed by passing the proper `MessageTemplate`
  - ! again, use method `createMessageTemplate`
- to be extended by overriding its `handle[...]` *callback* methods
  - `handleCfp()` the initial `CFP` message
  - `handleAcceptProposal())` when an `ACCEPT_PROPOSAL` message is received from the initiator
  - . . .

Contract Nets IV

! check the `ds.lab.jade.bookTrading.contractNet` example for the code

Launching `ds.lab.jade.bookTrading.contractNet`

Open a command prompt and position yourself into folder `ds-jade/`

-
  ```
  java -cp libs/jade.jar:bin/ jade.Boot -gui -agents
  seller:ds.lab.jade.bookTrading.contractNet.BookSellerAgent &
  ```

-
  ```
  java -cp libs/jade.jar:bin/ jade.Boot -container -agents
  buyer:ds.lab.jade.bookTrading.contractNet.BookBuyerAgent &
  ```

On Windows, substitute “&” with “start /B” (placed as first command)
The example should work anyway regardless of the order in which agents are launched and regardless of how many buyers and sellers you launch (at least one), provided they have different names

Responder Behaviours I

Cyclic vs. single-session responders

Responder behaviours may have two forms

Cyclic Serve interactions initiated by different agents sequentially

1. wait for the protocol initiation message
2. serve the protocol
3. go back waiting for a new protocol initiation message

Single-Session Serve interactions initiated by different agents in parallel

1. get the protocol initiation message in the constructor
   → requires an external behaviour to be used
2. serve the protocol
3. terminate

! check the `jade.proto` package to learn more

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - **JADE Agents & Java Swing**
   - Agent Mobility in JADE

Java Swing Troubles I

What is the problem?

Whenever developing JADE agents which need to interact with a Java GUI, the *thread-per-agent* concurrency model of JADE agents must work together with the Swing Event Dispatcher Thread (EDT) concurrency model

Java Swing Troubles II

More in detail

- as you should know, the Swing framework *is not thread-safe*, so any code that updates the GUI elements must be executed within the EDT
  - → since modifying a *model object* triggers an update of the GUI, model objects too have to be manipulated just by the EDT
- the `SwingUtilities` class exposes two *static methods* to delegate execution of `Runnable` objects to the EDT

  `invokeLater()` puts the `Runnable` into the System Event Queue (SEQ) (accessed by the EDT only) and returns immediately—*asynchronous* call

  `invokeAndWait()` puts the `Runnable` into the SEQ and blocks waiting its completion—*synchronous* call

JADE Solution I

GuiAgent class

To develop JADE agents interacting with a GUI, simply extend `GuiAgent` class in package `jade.gui`

`onGuiEvent(GuiEvent e)` may be viewed as the *equivalent* of the `actionPerformed()` method in Java Swing, that is, a callback invoked by JADE platform as soon as a `GuiEvent` is generated

`postGuiEvent(GuiEvent e)` used by the agent's GUI to *queue GUI events* for later processing—similar to queueing $\mathcal{ACL}$ messages in its mailbox

Jade Solution II

GuiEvent class

A GuiEvent object has

- two *mandatory* attributes

  source the Object source of the event

  type an integer identifying the kind of event generated

- an optional list of parameters eventually used for events processing

  addParameter() takes the Object to add as a GuiEvent parameter
  getParameter() gets the i-*th* parameter
  getAllParameter() returns an Iterator to browse all parameters

JADE Solution III

One advice

From JADE programmer's guide

> *"In general, it is not a good thing that an external software component maintains a direct object reference to an agent, because this component could directly call any public method of the agent, skipping the asynchronous message passing layer and turning an autonomous agent into a server object, slave to its caller. The correct approach is that to gather all the external methods into an interface, implemented by the agent class, then an object reference of that interface will be passed to the external software component (e.g., a GUI) so that only the external methods will be available from event handlers."*

JADE Solution IV

On mixing paradigms

Both the GUI and the objects reference issues raise the following warnings

! beware of mixing programming paradigms

- JADE provides an agent-oriented development framework
- JADE is implemented in Object Oriented (OO, e.g. Java)

! when developing agent-oriented software, stay in the agent-oriented world as much as possible

- using OO GUIs is ok
- using OO external references is ok (but be careful)

! do not think in OO terms, think in agent-oriented terms

- e.g. no threads
- e.g. no method calls between agents

JADE Solution V

! check the `ds.lab.jade.bookTrading.gui` example carefully

Launching `ds.lab.jade.bookTrading.gui`

- open a command prompt and position yourself into folder `ds-jade/`

  -
    ```
    java -cp libs/jade.jar:bin/ jade.Boot -gui -agents
    seller:ds.lab.jade.bookTrading.gui.BookSellerAgent &
    ```

  -
    ```
    java -cp libs/jade.jar:bin/ jade.Boot -container -agents
    buyer:ds.lab.jade.bookTrading.gui.BookBuyerAgent &
    ```

! on Windows, substitute "&" with "start /B" (placed as first command)

- the example should work anyway regardless of the order in which agents are launched and regardless of how many buyers and sellers you launch (at least one), provided they have different names

Focus on. . .

1. JADE Overview
   - JADE & FIPA
   - JADE Agents
   - JADE ACC
   - JADE Tools
2. Getting Started with JADE
3. JADE Basics
   - JADE Architecture
   - JADE Agents
   - Agent Behaviours
   - JADE Messaging
   - JADE Communication API
4. JADE Advanced
   - Directory Facilitator
   - FIPA Interaction Protocols in JADE
   - JADE Agents & Java Swing
   - **Agent Mobility in JADE**

Jade Mobility I

What does *mobility* mean for Jade?

In Jade, *mobility* is the ability of an agent program to either migrate or clone (make a copy of) itself *across one or multiple network hosts*

Which kind of mobility?

As you may know, at least two different forms of mobility can be defined:

weak only the program (agent) *code* is moved/cloned

strong also the program (agent) state is moved/cloned along with its code—supposing to know what "state" means

Jade supports *some form* of strong mobility

JADE Mobility II

JADE strong mobility

A JADE agent can

- move/clone its state, which means:
  1. stop its execution on the local container
  2. move/clone to a remote container
  3. resume its execution there from the *exact point* where it was interrupted
- move/clone its code, which means that if its code is not available on the destination container, then it is automatically retrieved by JADE platform

Keep in mind that. . .

! in order to be able to move, an agent must be `Serializable`

API: `Location` I

Where to move/clone to?

The `jade.core.Location` *interface* represents a place where agents can move / be cloned to

`getID()` to obtain the `Location` unique ID

`getName()` to obtain the `Location` name

`getAddress()` to get its address

`getProtocol()` to know the exploited transport protocol

API: Location II

Intra- vs. inter- platform mobility

Two different classes implement the Location interface (both from its same package):

- ContainerID for intra-platform mobility
  - let cName be a container name, its ContainerID can be obtained with `new ContainerID(cName, null)`
- PlatformID for inter-platform mobility
  - requires the migration service add-on to be installed
  - it is developed and maintained by the Universitat Autònoma de Barcelona at tao.uab.cat/ipmp/

API: Action I

Finding destinations

To get a Location object, an agent must query the AMS by sending it an ACLMessage.REQUEST (thus, expecting an .INFORM back) storing either

- a new WhereIsAgentAction(AID aid) object
  → to get the Location where the given agent is
- a new QueryPlatformLocationsAction() object
  → to get all the Locations available within the JADE platform

In both cases, what you get is a Location object which hides a ContainerID

API: Action II

What *actions*?

- jade.content.onto.basic.Action is the class representing a FIPA *action*, that is "an act to be carried out by an agent"

? do you remember we defined $\mathcal{ACL}$ messages as *communicative acts*?

API: `Action` III

Action how-to

To create and request an `Action`

1. instantiate the `Action` object
   → `Action a = new Action()`
2. decide who should perform the action—the AMS in our case
   → `a.setActor(getAMS())`
3. choose the action to be performed
   → `a.setAction(new QueryPlatformLocationsAction())`
4. *embed* the action into the request $\mathcal{ACL}$ message
   →
   ```
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST)
   Agent.getContentManager().fillContent(msg, a))
   ```
5. send the message to the receiver—again, the AMS for us
   →
   ```
   msg.addReceiver(getAMS())
   send(msg)
   ```

API: Action IV

Collecting AMS replies I

To collect AMS replies you can do something like

1. create a suitable data store for locations

   → `Map locations = new HashMap()`

2. receive replies according to your preferred policy but using the correct `MessageTemplate`

   →
   ```
   MessageTemplate mt = MessageTemplate.and(
     MessageTemplate.matchSender(getAMS()),
     MessageTemplate.matchPerformative(ACLMessage.INFORM))
   ACLMessage reply = blockingReceive(mt)
   ```

3. ...

API: Action V

Collecting AMS replies II

1. . . .
2. *decode* the content of AMS reply—method dual to `Agent.getContentManager().fillContent(msg, a))`

   → `Result res = (Result) getContentManager().extractContent(reply)`
3. (in our case) collect all the `Locations`

```
→ Iterator it = res.getItems().iterator()
  while(it.hasNext()){
     Location l = (Location)it.next()
     locations.put(loc.getName(), l)
  }
```

doMove/doClone How-To I

Self-motion

In the case an agent autonomously decides to move itself to another (remote) container in the same Jade platform, it simply calls the method `doMove()` passing the destination `Location` as a parameter—either discovered thanks to the AMS or a-priori known

Self-cloning

The case of cloning is similar, except that method to call is obviously `doClone()` and that a second parameter other than the target `Location` should be passed to the call: the new name of the cloned agent (a `String`)

doMove/doClone How-To II

Request for movement/cloning I

Instead, if any JADE agent wishes to make another agent move, it can only perform a [Move/Clone]Action *request*, hoping the destination agent will do it—nothing more should be expected as usual

1. create a MobileAgentDescription
   → `MobileAgentDescription mad = new MobileAgentDescription`
2. fill its mandatory fields
   →
   ```
   mad.setName(aid)
   mad.setDestination(location)
   ```
3. embed it in a [Move/Clone]Action object
   →
   ```
   MoveAction ma = new MoveAction()
   ma.setMobileAgentDescription(mad)
   ```
4. ...

doMove/doClone How-To III

Request for movement/cloning II

1. ...
2. pack the $\mathcal{ACL}$ request message encoding the [Move/Clone]Action object

   →
   ```
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST)
   Agent.getContentManager().fillContent(msg, ma))
   ```
3. send the move/clone request message

   →
   ```
   msg.addReceiver(aid)
   send(msg)
   ```

doMove/doClone How-To IV

Response for movement/cloning

The receiver agent, if agrees with the request

1. decodes the content of the $\mathcal{ACL}$ message conveying the action request
   - → `ContentElement content = getContentManager().extractContent(msg)`
2. gets the `[Move/Clone]Action`
   - → `MoveAction ma = (MoveAction)(((Action)content).getAction())`
3. gets the destination `Location`
   - → `Location loc = ma.getMobileAgentDescription().getDestination()`
4. eventually, moves/clones itself
   - → `if(loc != null) doMove(loc)`

doMove/doClone How-To V

Last note

To be able to call the above-used methods from the ContentManager object, the jade.content.lang.sl.SLCodec and the jade.domain.mobility.MobilityOntology must be *registered* with it.

→ to do so, write in agents setup() method

```
getContentManager().registerLanguage(new SLCodec())
getContentManager().registerOntology(
   MobilityOntology.getInstance()
)
```

doMove/doClone How-To VI

Not a FIPA standard

Please notice that such ontology is not yet a FIPA standard, hence may adapted in the future[a]

[a] unsure whether a standard is currently available, nor if JADE 4.4 complies to it, for we were not able to find references in documentation.

doMove/doClone How-To VII

! check the ds.lab.jade.mobility example

Launching ds.lab.jade.mobility: single host scenario

Open a command prompt and position yourself into folder ds-jade/

- `java -cp libs/jade.jar:bin/ jade.Boot -gui &`
- `java -cp libs/jade.jar:bin/ jade.Boot -container -agents ste:ds.lab.jade.mobility.MobileAgent &`

On Windows, substitute “&” with “start /B” (placed as first command)

doMove/doClone How-To VIII

Launching ds.lab.jade.mobility: multi host scenario

On the first host, open a command prompt and position yourself into folder `ds-jade/`

- `java -cp libs/jade.jar:bin/ jade.Boot -gui &`

On the second host, open a command prompt and position yourself into folder `ds-jade/`

- `java -cp libs/jade.jar:bin/ jade.Boot -container -host` *`ip.of.first.host`* `-agents ste:ds.lab.jade.mobility.MobileAgent &`

On Windows, substitute "&" with "start /B" (placed as first command)

doMove/doClone How-To IX

If for some reason (e.g. you are in lab.) you need to bind JADE to a specific IP address and TCP port on your hosts, issue the following commands:

- on he first host: `java -cp libs/jade.jar:bin/ jade.Boot -gui -local-host` *`ip.of.first.host`* `-local-port` *`firstport`*`&`
- on the second host: `java -cp libs/jade.jar:bin/ jade.Boot -container -local-host` *`ip.of.second.host`* `-local-port` *`secondport`* `-host` *`ip.of.first.host`* `-port` *`firstport`* `-agents ste:ds.lab.jade.mobility.MobileAgent &`

Are We Done with JADE?

In this course and lab, yes we are

The general answer, instead, is no. JADE offers many other things in addition to what we've seen during lab. lessons:

- topic-based communication
- fault tolerance service
- persistent message delivery service
- user-defined ontologies support
- ...

... feel free to experiment by yourselves, and to ask questions as well!

References

Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2010a).
JADE Programmer's Guide.
JADE Board, JADE v. 4.0 edition.

Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., and Mungenast, R. (2010b).
JADE Administrator's Guide.
JADE Board, JADE v. 4.0 edition.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007).
Developing Multi-Agent Systems with JADE.
Wiley.

FIPA ACL (2002).
Agent Communication Language Specifications.
Foundation for Intelligent Physical Agents (FIPA).

Agents & Multi-Agent Systems with Jade

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Stefano Mariani   Andrea Omicini

s.mariani@unibo.it   andrea.omicini@unibo.it

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

Academic Year 2017/2018