Lecture 11: Introduction to JADE

Design of Agent-Based Systems
(TIES433), Autumn 2008

Artem Katasonov
University of Jyväskylä

Previously on…

- Mandatory services to be provided by an agent platform are life-cycle management, message transport service, white-pages service and yellow-pages service.
- There exist many agent middleware platforms. Some comply to FIPA standards, some do not but still use a similar architecture.
- FIPA does not standardize agent's inner architecture. So, this is the main differentiating factor for the agent platforms. Some use an APL (Agent Factory, 3APL, Jason, UBIWARE), some do not use but imply an architecture otherwise (Cougaar), some leave it is up to the developers (JADE).
- Our recommendation: the UBIWARE Platform (of course :)

JADE

- http://jade.tilab.com/
- Java Agent Development Framework (JADE) is free software developed and distributed by Telecom Italia Lab.
- Currently the JADE Board has 5 members: Telecom Italia, Motorola, Whitestein Technologies AG., Profactor GmbH, and France Telecom R&D.
- The latest version of JADE 3.6 is released on 5 May 2008.
- Basically, a Java implementation of FIPA specifications.
- Comes as just a JAR library of size 2.3 MB (+docs, examples, etc).
- The only software requirement to execute the system is the Java Run Time Environment version 1.4.

Components of JADE

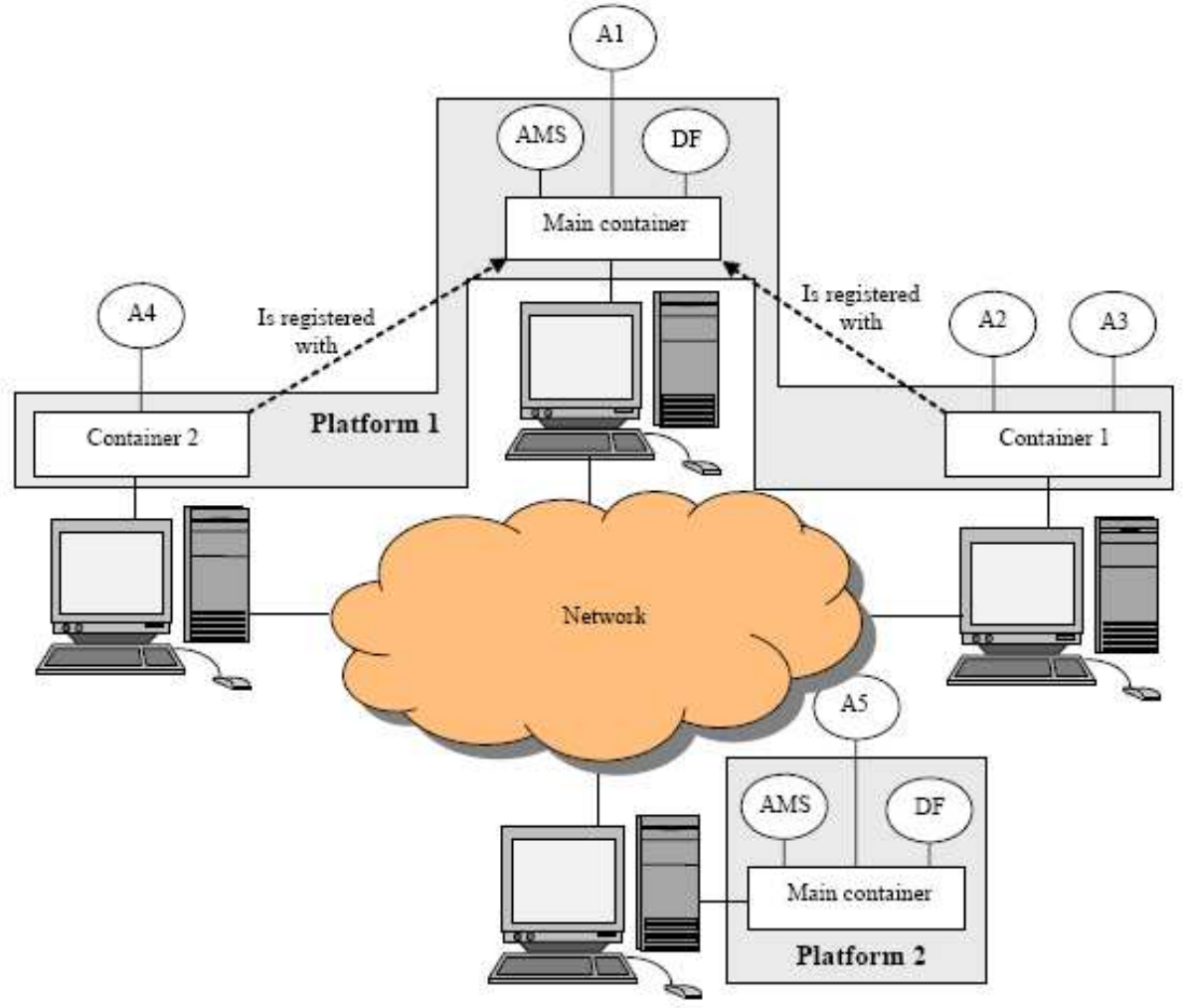
- JADE is a framework that facilitates the development of multi-agent systems. It includes:
  - A *runtime environment* where JADE agents can “live” and that must be active on a given host before one or more agents can be executed on that host.
  - A *suite of graphical tools* that allows administrating and monitoring the activity of running agents.
  - A *library of classes* that programmers use (directly or by extending them) to develop their agents.

- Has a number of add-ons including 3rd party.

JADE run-time environment

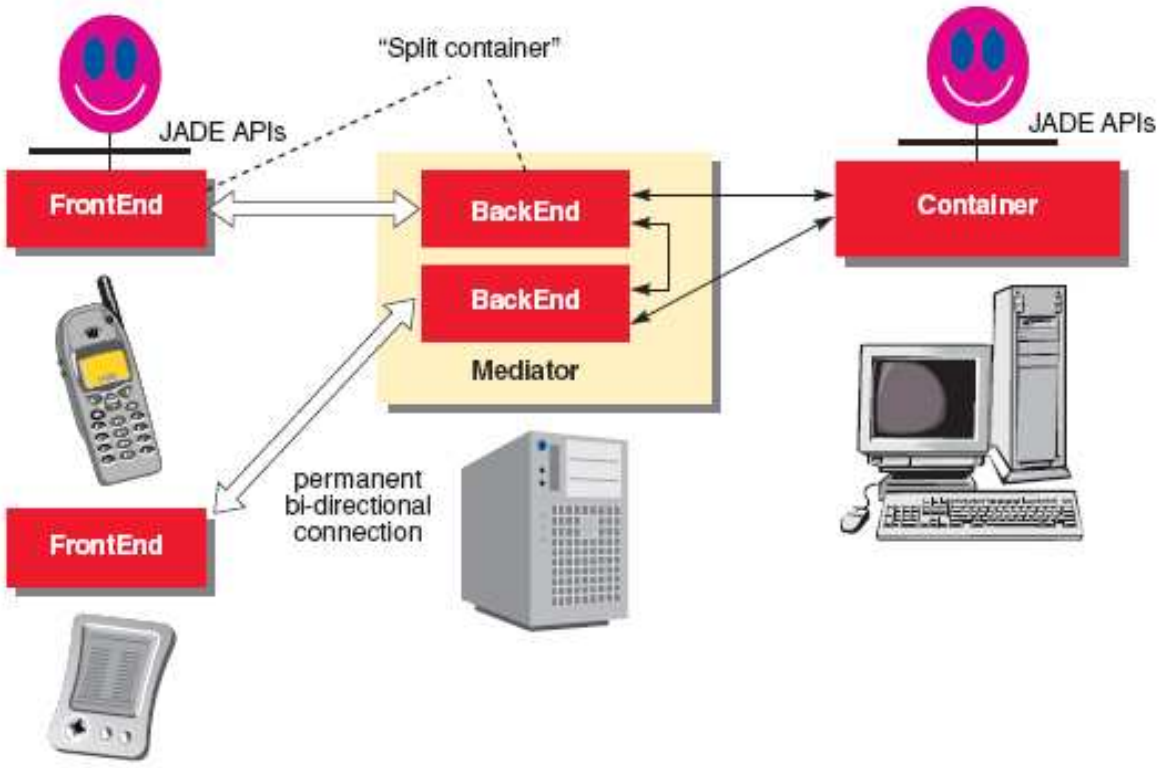
- Each running instance of the JADE runtime environment is called a *Container* as it can contain several agents.
- The set of active containers is called a *Platform*.
- A single special *Main container* must always be active in a platform and all other containers register with it as soon as they start.
- When starting secondary containers, they must know the address of the main container
- Normally, one container is run on host. However, can also run several containers on the same host.

JADE run-time environment (2)



Running on mobile terminals

- Requires a mediator computer in fixed network.
- Container is split into back-end and front-end



Starting JADE

java -cp .;JadeLeap.jar jade.Boot –port 80
john:org.example.JohnAgent(par1,par2);mary:org.example.MaryAgent

"john" and "mary" are names of the agents, while org.example.JohnAgent and org.example.MaryAgent are Java classes implementing those agents

Starting additional container on a remote host:

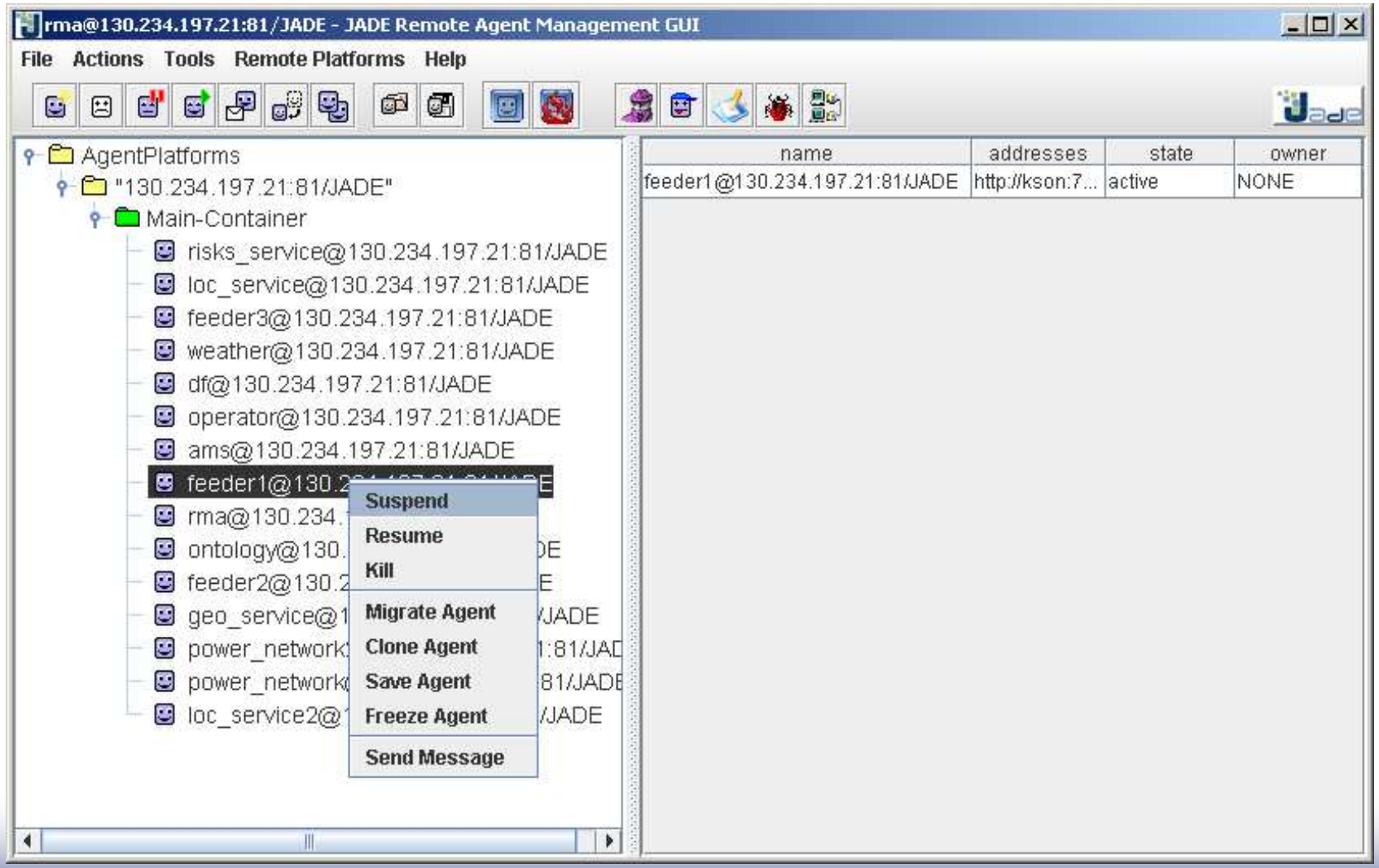
java jade.Boot –*container* –host arman –port 80 –local-port 80
bill:org.example.BillAgent

Starting JADE with launching Remote Monitoring Agent (RMA) GUI:

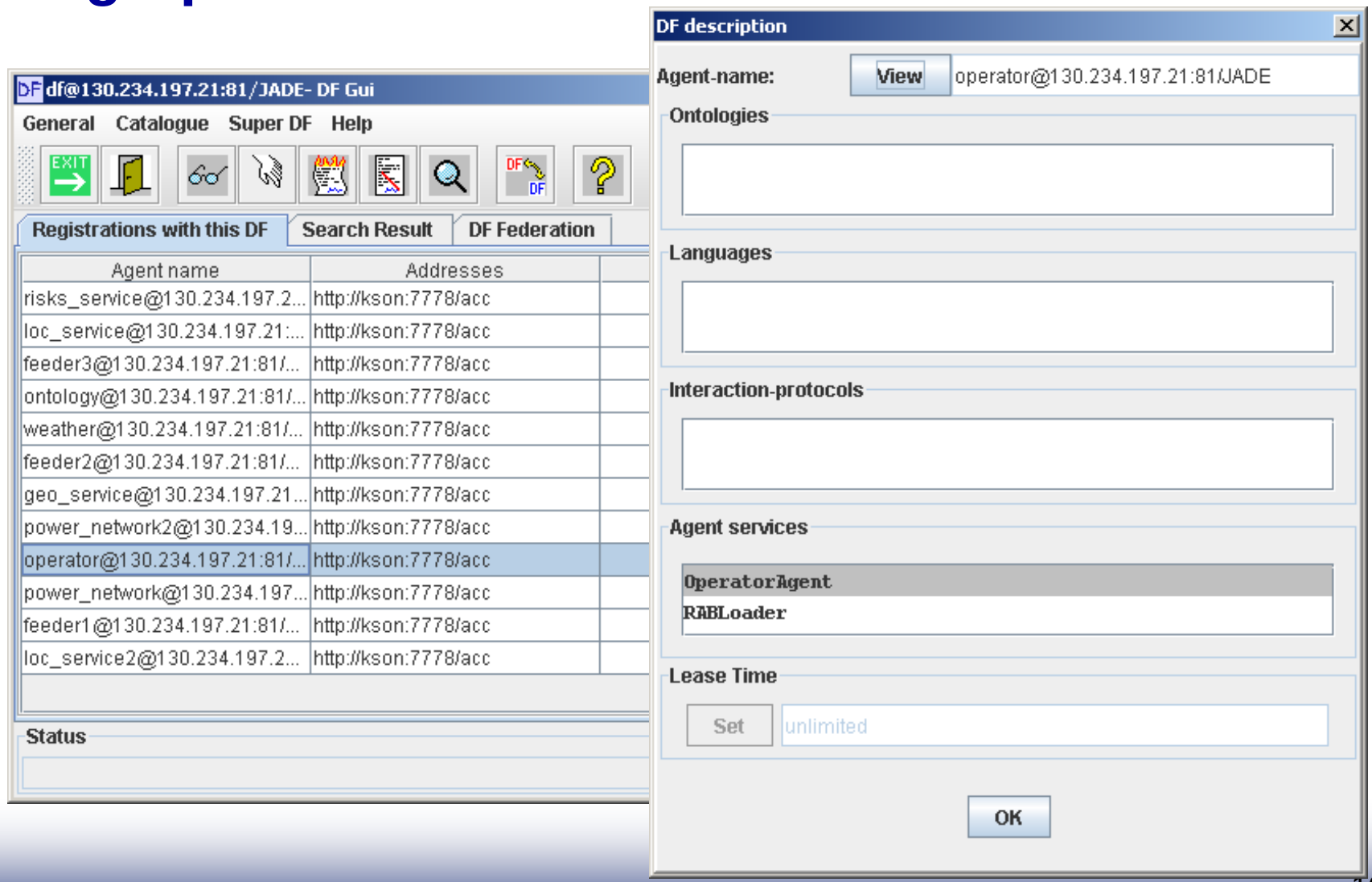
java jade.Boot –gui …

See http://jade.tilab.com/doc/administratorsguide.pdf for other command line parameters.

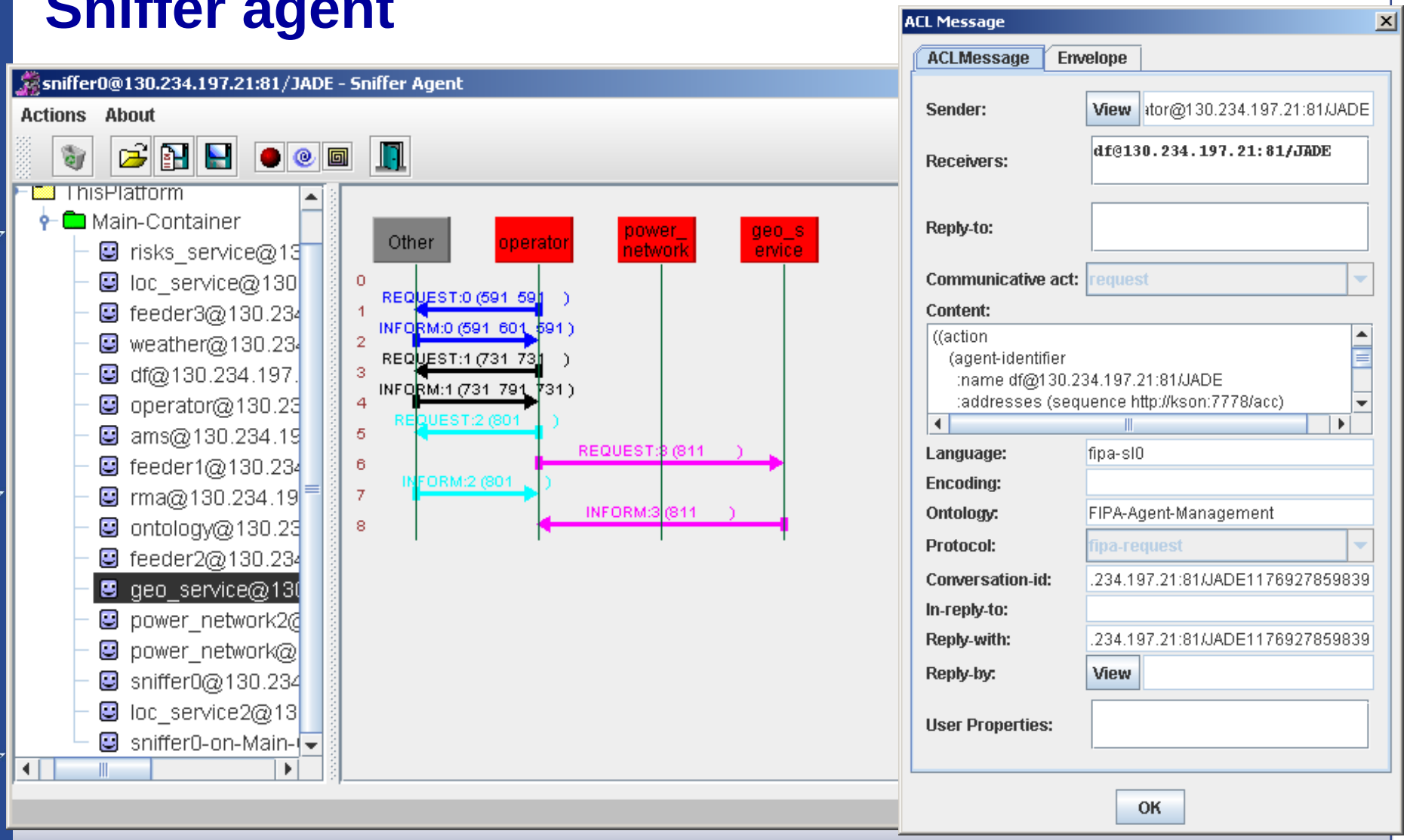
RMA graphic Interface



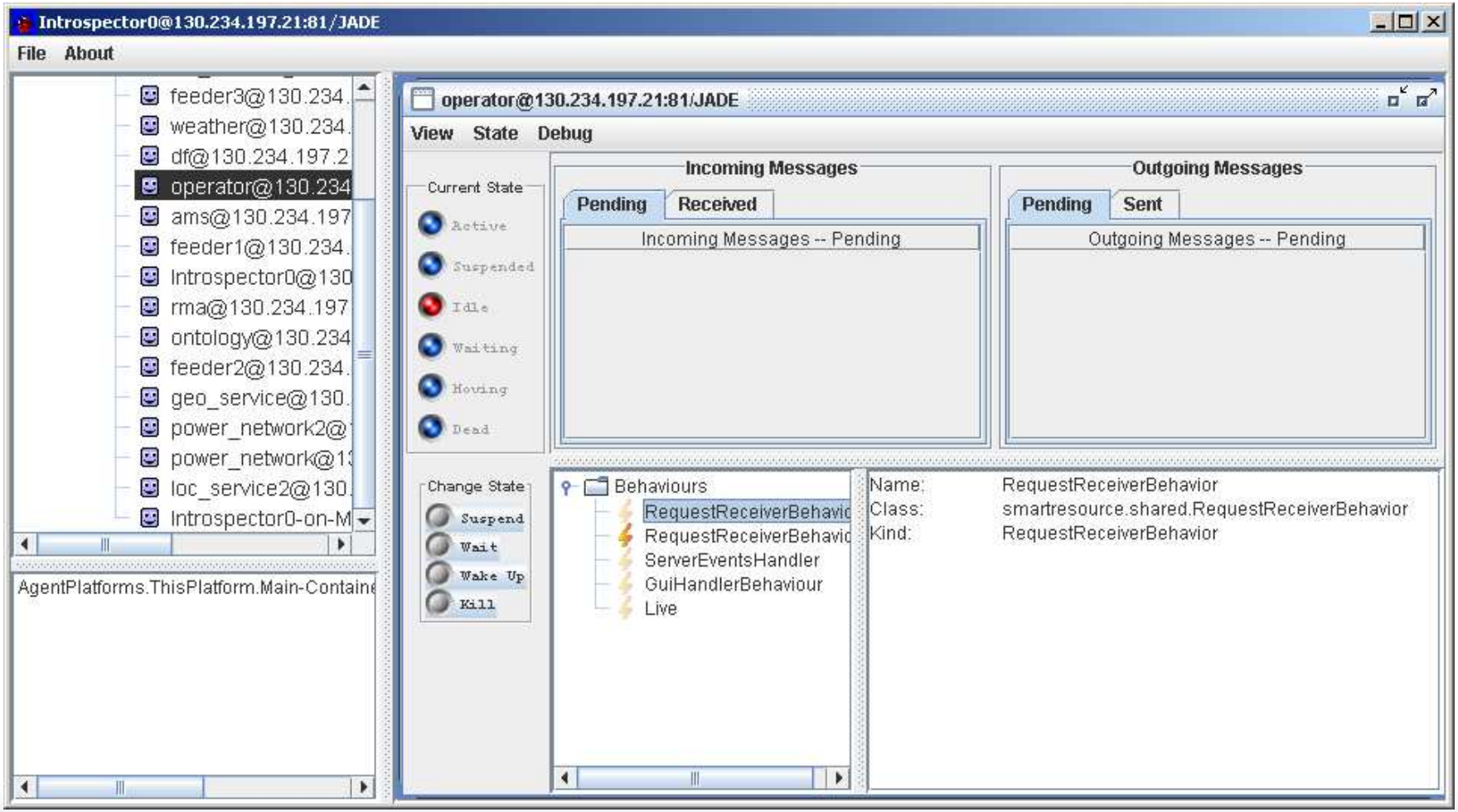
DF graphic interface



Sniffer agent



Introspector agent



Programming JADE agents

- Programmer's tutorial: http://jade.tilab.com/doc/programmersguide.pdf
- Online JavaDocs: http://jade.tilab.com/doc/api/index.html

- Creating a JADE agent requires writing a subclass of the class *jade.core.Agent*

- Minimal agent program (HelloWorldAgent.java):

```
import jade.core.Agent;
public class HelloWorldAgent extends Agent {
        protected void setup() {
                System.out.println("Hello World! My name is "+getLocalName());
        }
}
```

To compile: javac -classpath JadeLeap.jar HelloWorldAgent.java

To start: java -cp .;JadeLeap.jar jade.Boot -port 80 John:HelloWorldAgent

Agent with a Behavior

- Instead of putting the actions to be performed inside *setup()*, they may be put inside an *agent behavior*.
- JADE provides class jade.core.behaviours.Behaviour and several subclasses of it, e.g. OneShotBehaviour, CyclicBehaviour.

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

public class HelloWorldAgent extends Agent {
    protected void setup() {
      addBehaviour(new OneShotBehaviour() {
          public void action(){
                  System.out.println("Hello World! My name is "+getLocalName());
          }
      });
    }
}
```

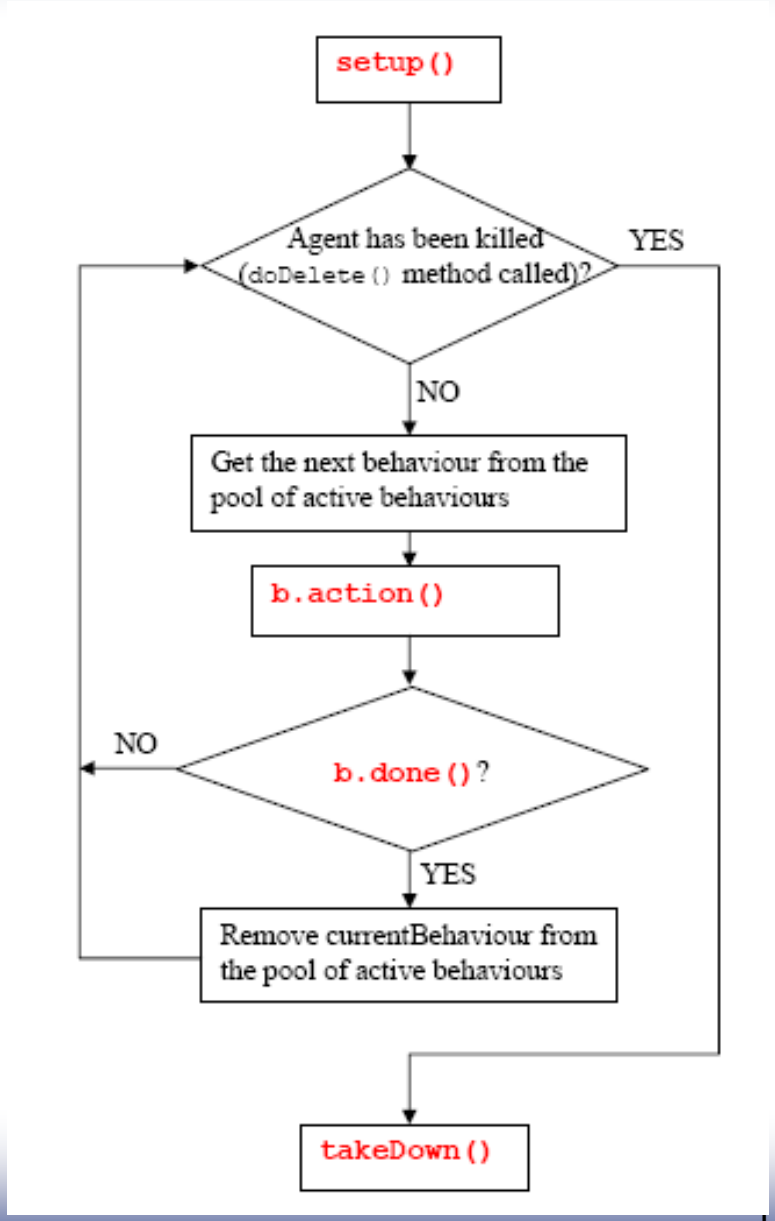
Agent with a Behavior (2)

- Other way is to create an explicit subclass of OneShotBehaviour. Then, that behavior can also be reused by other agents.

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
public class HelloWorldAgent extends Agent {
    protected void setup() {
                addBehaviour(new HelloWorldBehaviour());
    }
}
class HelloWorldBehaviour extends OneShotBehaviour{
    public void action(){
        System.out.println("Hello World! My name is
                                "+myAgent.getLocalName());
    }
}
```

Behaviors execution

- Each agent has its own thread (process in Java Virtual Machine).
- Agent's core uses simple queue for sequential execution of behaviors.
- This means that until *action()* of the current behavior returns, no other behavior can be executed.
- If behavior is supposed to have more iterations/steps, its *done()* method has to return *false.* Then, the behavior will be put in the end of the queue and its action will be called again when time comes.
- If behavior's *done()* returns *true*, the behavior is removed as finished.



Standard behavior classes

- **SimpleBehaviour** – basic behavior class, the programmer has to implements both *action()* and *done()*
- **OneShotBehaviour** – behavior with one iteration/step only, its *done()* always returns *true*
- **CyclicBehaviour** – never-ending behavior, its *done()* always returns *false*

- **TickerBehaviour** - periodically executes a user-defined piece of code, has *action()* already implemented, programmer writes *onTick()*
- **WakerBehaviour** - executed only once just after a given timeout is elapsed, has *action()* already implemented, programmer writes *onWake()*

- **ParallelBehaviour** – composite behavior whose sub-behaviors are executed in parallel, finishes when all, any, or specified number of children finishes.
- **SequentialBehaviour** - composite behavior whose sub-behaviors are executed sequentially.

Elements on the class Behaviour

- **onStart()** – can be implemented to do some initializing actions, called just before calling *action()* for the first time
- **onEnd()** – can be implemented to do some finalizing actions, called just before removing the behavior
- **block()** - blocks this behavior until a new message arrives. Blocked behavior is not executed until it returns to the active state.
- **block (long millis)** - blocks this behavior until a new message arrives or until specified time (in milliseconds elapses).
- **isRunnable()** – returns *false* if behavior is blocked and *true* if not.
- **myAgent** - reference to the agent owning this behavior. Enables behaviors to call public methods of jade.core.Agent such as send(), receive(), doMove(), getLocalName() and other.

Accessing agent's start parameters

```
protected void setup() {
    Object[] args = getArguments();
    if(args!=null && args.length>0){
        for (int i=0; i<args.length; i++){
            System.out.println(args[i].toString());
        }
    }
```

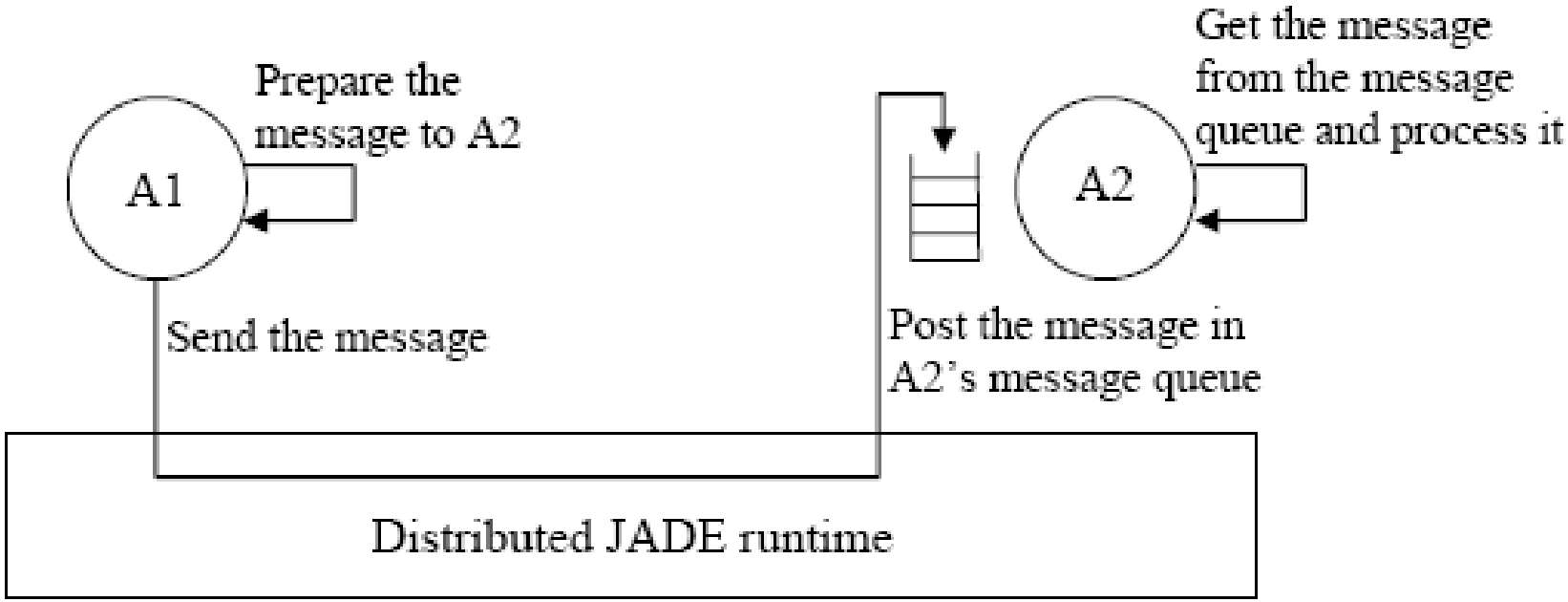
- Parameters are passed from the command line starting JADE, comma-separated, e.g. *john:HelloWorldAgent(parameter1,parameter2)*;

Sending an ACL message

```
import jade.lang.acl.ACLMessage;
import jade.core.AID;
…
ACLMessage message = new ACLMessage(ACLMessage.INFORM);
message.addReceiver (new AID (“mary”, AID.ISLOCALNAME) );
message.setContent (“Hi Mary!”);
myAgent.send (message);
```

- Class *AID* – enables working with Agent Identifiers according to FIPA standards (includes localname, platform name, ACC addresses, resolvers). For messaging inside an agent platform, local name is enough.
- Class *ACLMessage* – enables working with FIPA’s ACL Messages. Can set and get the whole set of ACL standard fields. Also provides *createReply()* method.

Incoming messages queue



- Jade platform posts incoming messages to the agent's queue
- It is responsibility of one of the agent's behaviors to get messages from the queue.

Receiving an ACL message

```
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
…
MessageTemplate mt = MessageTemplate.MatchAll();
mt=MessageTemplate.and (mt,MessageTemplate.MatchPerformative
                                        (ACLMessage.REQUEST) );
ACLMessage msg = myAgent.receive(mt);
if (msg != null){
…
}
else block();
```

- Class *MessageTemplate* – enables matching performative, content, encoding, ontology, conversation id, in-reply-to, sender and other fields using OR, AND, NOT logical operations.

Registering with Directory Facilitator

```
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
…
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(myAgent.getAID());

ServiceDescription sd = new ServiceDescription();
sd.setName(”Seller”);
sd.setType(”Book-Trading");
dfd.addServices(sd);

try{
      DFService.register( myAgent, dfd );
}catch (FIPAException fe) {…}
```

Searching with Directory Facilitator

```
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
…
DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setName(”Seller”);
sd.setType(”Book-Trading");
template.addServices(sd);
try {
      DFAgentDescription[] result = DFService.search(myAgent, template);
      for(int i=0; I < result.length; i++){
            String name=result[i].getName().getLocalName();
             …
      }
}catch (FIPAException fe) {…}
```

Controlling agent's state

- **doWait()** – causes the whole agent to block, stopping all its activities until a message arrives or the doWake() method is called.
- **doWait (long millis)** – agents blocks until a message arrives or timeout elapses.
- **doSuspend()** – suspends the agents until doActivate() method is called.
- **doWake()** – returns the agent to active state from waiting state. Obviously, should be called from the outside of the agent's thread.
- **doActivate()** – returns the agent to active state from suspended or waiting state. Obviously, should be called from the outside of the agent's thread.
- **doDelete()** – kills the agent.
- **takeDown()** - can be implemented to do some finalizing actions, called after *doDelete()* was called and just before terminating the agent.
- **blockingReceive(MessageTemplate pattern)** – receive a message. This method is blocking and suspends the whole agent until a message is available in the queue. There is also a version having **long millis** parameter to wait at most specified time.

Mobility

```
import jade.core.ContainerID;
import jade.core.Location;
…
Location loc = here();
if (!loc.getName().equals("Main-Container")){
      doMove( new ContainerID("Main-Container", null) );
}
…
protected void beforeMove() {}
protected void afterMove() {
      System.out.println( here().getAddress() );
      System.out.println( here().getName() );
}
```

- The actual transition happens after the behavior that called *doMove()* ends. *beforeMove()* is called before actual transition and can be implemented to do some preparing actions. *afterMove()* is called after the transition is complete and can be implemented to perform some actions.

Cloning

```
doClone( new ContainerID("Main-Container", null),  “copyOfJohn”);
…
protected void beforeClone() {
…
}
protected void afterClone() {
}
```

- The actual cloning happens after the behavior that called *doClone()* ends. *beforeClone()* is called for mother agent before actual cloning and can be implemented to do some preparing actions. *afterClone()* is called for daughter agent after the cloning is complete and can be implemented to perform some actions.

Requesting from AMS the list of containers

```
import java.util.Iterator;
import jade.core.ContainerID;
import jade.core.Location;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.domain.FIPANames;
import jade.domain.mobility.MobilityOntology;
import jade.content.onto.basic.Action;
import jade.content.onto.basic.Result;
import
      jade.domain.JADEAgentManagement.QueryPlatformLocati
      onsAction;
import jade.content.lang.sl.SLCodec;
…
getContentManager().registerLanguage(new SLCodec(),
      FIPANames.ContentLanguage.FIPA_SL0);
getContentManager().registerOntology(MobilityOntology.getInst
      ance());

ACLMessage message = new
      ACLMessage(ACLMessage.REQUEST);
message.addReceiver(getAMS());
message.setLanguage(FIPANames.ContentLanguage.FIPA_SL0);
message.setOntology(MobilityOntology.NAME);
message.setProtocol(FIPANames.InteractionProtocol.FIPA_REQ
      UEST);
```

```
Action action = new Action();
action.setActor(getAMS());
action.setAction(new QueryPlatformLocationsAction());
try {
      getContentManager().fillContent(message, action);
}catch (Exception e){e.printStackTrace();}

 myAgent.send(message);

MessageTemplate mt = MessageTemplate.MatchAll();
ACLMessage msg = myAgent.blockingReceive(mt);

Result results;
try {
     results =
       (Result)myAgent.getContentManager().extractContent(msg);
}catch (Exception e){e.printStackTrace();}

Iterator res_it= results.getItems().iterator();
while(res_it.hasNext()){
       Location loc = (Location)res_it.next();
       System.out.println("Exists: "+loc.getName());
}
```

Requesting from AMS the list of containers (2)

- Mainly, this is an example of using ContentManager class to work with messages in a content language, FIPA's SL0 in this case.
  - Need to register language
  - Need to register ontology
  - Need to set language and ontology for the message
  - Need use "actions", QueryPlatformLocationsAction in this case

- Then, the ContentManager can take care on encoding the messages with *fillContent()* and parsing them with *extractContent()*.

Creating an agent

```
import jade.wrapper.AgentContainer;
import jade.wrapper.AgentController;
…
AgentContainer container=(AgentContainer) myAgent.getContainerController();

Object[] params=new Object[1];
params[0]=”something”;
try{
    AgentController agnt=container.createNewAgent(“Mary”, “HelloWorldAgent”, params);
}
catch(Exception e){…}

agnt.start();
```

- *getContainerController()* method of Agent class provides access to the controller or the container in which the agent is situated.

Other nice things

- Killer

```
AgentContainer container = myAgent.getContainerController();
AgentController victim = container.getAgent(“John");
victim.kill();
```

- Kamikaze (destroy the container and all the agents in it, including oneself)

```
AgentContainer container = getContainerController();
container.kill();
```

Lecture 11: Main points

- JADE is a framework that facilitates the development of multi-agent systems. It includes a runtime environment, a suite of graphical tools for administrating and monitoring, and a library of classes for developing agents.
- Two main classes in JADE are Agent and Behaviour.
- When programming in JADE, some things take two lines of code, but some may take two pages of code.

Following next:

- The UBIWARE Platform
  - Motivation
  - Architecture
  - Semantic Agent Programming Language (S-APL)
  - Available Reusable Atomic Behaviors (RABs)
  - Developing new RABs