Java Agent DEvelopment Framework

Cos'è JADE

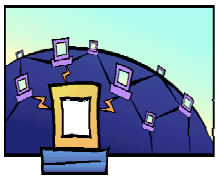
- Jade è una piattaforma per la realizzazione di sistemi distribuiti multi-agente
- E' interamente realizzato in Java da Telecom Italia Lab.
- Può essere scaricato da http://jade.tilab.com

Sistemi distribuiti

Una definizione

- Un sistema distribuito (SD) è un sistema i cui componenti sono dislocati nei diversi host di una rete;
- I componenti di un SD sono processi fortemente cooperanti eseguiti in parallelo su Unità di Elaborazione autonome;
- L'utente non dovrebbe percepire la distribuzione dei componenti del sistema (Location Transparency);

Alcune motivazioni



- E' possibile gestire la crescita di un sistema (nodi e numero utenti) in modo incrementale;
- E' possibile condividere dati e risorse;
- I SD Presentano una maggiore tolleranza ai guasti;

Sistemi distribuiti basati su agenti

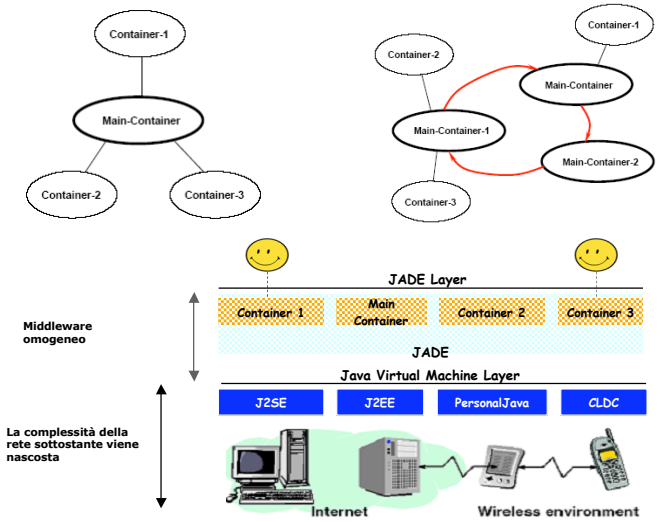
Caratteristiche

- MAS (Multi Agent System): Sistemi distribuiti i cui componenti sono Agenti;
- Gli agenti sono entità autonome capaci di svolgere compiti e comunicare per mezzo di messaggi;
- In un MAS gli agenti possono cooperare alla soluzione di compiti complessi che richiedono competenze trasversali a quelle dei singoli agenti;

FIPA

- Foundation for Intelligent Physical Agent: http://www.fipa.org
- E' un'organizzazione no-profit che ha lo scopo di produrre standard per l'interoperabilità di agenti software eterogenei.

La Piattaforma Jade



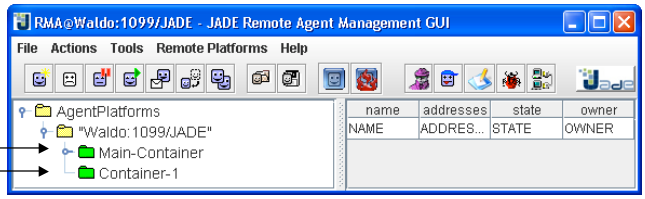
La Piattaforma Jade

Eseguire il MainContainer: `java jade.Boot -gui`

```
C:\>java jade.Boot -gui
    This is JADE 3.2 - 2004/07/26 13:41:05
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/

http://Waldo:7778/acc
Agent container Main-Container@JADE-IMTP://Waldo is ready.
```

Eseguire altri Container: `java jade.Boot -container`

```
C:\>java jade.Boot -container
    This is JADE 3.2 - 2004/07/26 13:41:05
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/

Agent container Container-1@JADE-IMTP://Waldo is ready.
```



Agenti Jade

Cosa fanno?

- Svolgono task
- Rendono pubblici i servizi che offrono;
- Comunicano con altri agenti;
- Cooperano con altri agenti per la risoluzione di compiti che richiedono competenze trasversali;
- Possono cambiare container effettuando una migrazione logica e, se necessario, fisica;

Dove sono?

- Gli agenti JADE sono ospitati dai diversi Container costituenti una Piattaforma JADE

Eseguire un agente

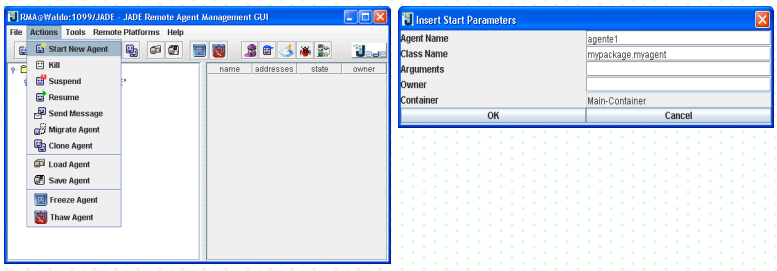
Da riga di comando

java jade.Boot *Option* AgentSpecifier**

AgentSpecifier = AgentName "**:**" ClassName ("(" Argument* ")")?
AgentName = nome dell'agente, definito dall'utente;
ClassName = la classe dell'agente;
Argument* = argomenti da passare all'agente;

```
Java jade.Boot –gui agente1:mypackage.agent agente2:mypackage.anotheragent
Java jade.Boot –container agente1:mypackage.agent agente2:mypackage.anotheragent
```

Oppure...



Creare Agenti

Come?

- Per creare un agente JADE è necessario estendere la classe `jade.core.Agent` e ridefinirne il metodo `setup();`
- Ogni agente JADE è univocamente definito attraverso un AID (`jade.core.AID`) che gli viene assegnato direttamente dalla piattaforma;
- Un agente può ottenere il proprio AID mediante il metodo `getAID();`

Esempio

```
import jade.core.Agent;

public class HelloWorldAgent extends Agent {

  protected void setup() {
    System.out.println("Hello World! my name is "+
                        getAID().getName());
  }
}
```

Terminare Agenti

Come?

- Un agente viene terminato quando ne viene invocato il metodo `doDelete();`
- Il metodo `doDelete()` richiama al suo interno il metodo `takeDown()` che può essere ridefinito in modo da personalizzare le operazioni da compiere prima che l'agente venga tolto dalla piattaforma;

Esempio

```
import jade.core.Agent;

public class HelloWorldAgent extends Agent {

  protected void setup() {
    System.out.println("Hello World! my name is "+
                        getAID().getName());
    doDelete();
  }

  protected void takeDown() {
    System.out.println("Addio...");
  }
}
```

Behaviours

Le operazioni che gli agenti JADE sono in grado di compiere vengono definite attraverso oggetti di tipo `Behaviour`;

I `Behaviour` vengono creati estendendo la classe astratta **`jade.core.behaviours.Behaviour`**;

Per fare in modo che un agente sia in grado di eseguire un particolare compito è sufficiente creare un'istanza della sottoclasse di `Behaviour` che implementa il *task* desiderato ed utilizzare il metodo `addBehaviour()` dell'agente

Ogni sotto-classe di `Behaviour` **deve** implementare i metodi

- `action()`: cosa il Behaviour deve fare;
- `done()`: condizione per cui il Behaviour può considerarsi concluso.

Tipi di Behaviour

One shot behaviours
Il metodo `action()` viene eseguito un'unica volta.
Il metodo `done()` restituisce sempre `true`.
Classe: `jade.core.behaviours.OneShotBehaviour`

Cyclic behaviours
Non terminano mai completamente.
Il metodo `action()` esegue la stessa operazione ogni volta che viene invocato.
Il metodo `done()` restituisce sempre false;
Classe: `jade.core.behaviours.CyclicBehaviour`

Complex behaviours
Hanno uno stato.
Il comportamento del metodo `action()` varia in funzione dello stato.
Il metodo `done()` restituisce `true` solo quando è verificata una certa condizione.

Esistono diversi tipi di behaviour predefiniti (*composite behaviour, sequential behaviour, parallel behaviour, sender behaviour*…). Tuttavia è possibile, creando estendendo la classe `Behaviour`, definire behaviour completamente personalizzati.

Behaviours, un esempio

Un complex Behaviour

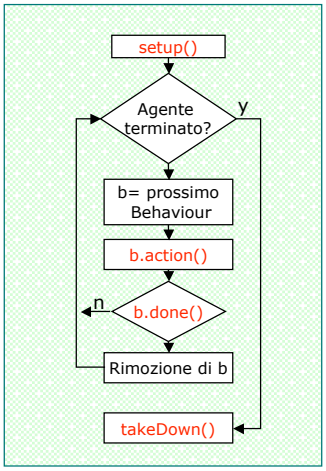
```
public class my3StepBehaviour extends Behaviour {

    private int state = 1;
    private boolean finished = false;

    public void action() {
          switch (state) {
                case 1: { op1(); state++; break; }
                case 2: { op2(); state++; break; }
                case 3: { op3(); state=1; finished = true; break; }
          }
    }
    … op1(){…} … op2(){…}… op3(){…}…
    public boolean done() {
          return finished;
    }
}
```

All'interno dell'agente…

```
protected void setup() {...

addBehaviour(new my3StepBehaviour());

…

}
```

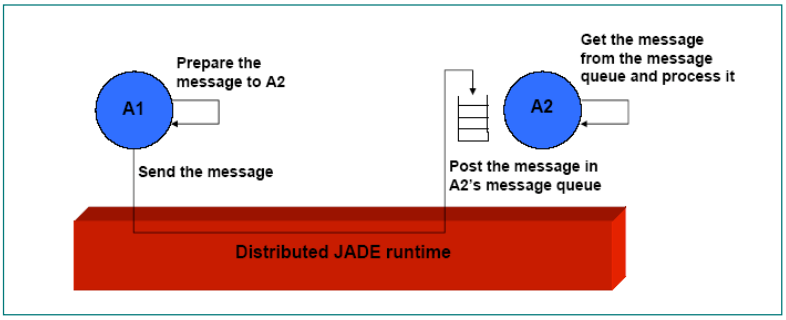
Modello esecutivo di un agente

- Un agente può eseguire diversi behaviours. Lo scheduling dei Behaviours avviene in modo cooperativo e non preemptive;
- Il cambiamento tra Behaviour avviene quando il metodo `action()` del Behaviour corrente è terminato;
- Il metodo `setup` deve contenere le operazioni di inizializzazione e l'aggiunta dei Behaviours;
- Il metodo `takeDown` deve contenere le operazione di *Clean up*



Messaggi

Il modello di comunicazione tra gli agenti è basato su scambio di messaggi asincrono



Messaggi

Quali messaggi?

Gli agenti JADE comunicano utilizzando ACL (Agent communication language), uno standard di comunicazione tra agenti definito da FIPA.

I messaggi scambiati tra gli agenti sono istanze della classe `jade.lang.acl.ACLMessage;`

ACLMessage:

La classe `ACLMessage` fornisce i metodi per impostare e ottenere i valori dei campi definiti da FIPA per l' ACL

addReceiver();
get/setPerformative();
get/setSender();
add/getAllReceiver();
get/setLanguage();
get/setOntology();
get/setContent();
....

Messaggi

Inoltrare un messaggio:

Per inoltrare un messaggio è necessario creare un oggetto `ACLMessage`, aggiungere uno o più riceventi per mezzo del metodo `addReceiver` (utilizzando gli AID); aggiungere il contenuto attraverso il metodo `setContent`; inoltrare il messaggio attraverso il metodo `send` dell'agente mittente.

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-Forecast-Ontology");
msg.setContent("Today it's raining");
send(msg);
```

Ricevere un messaggio:

Per Ricevere un messaggio un agente può utilizzare il metodo `receive()`, non bloccante!

```
ACLMessage msg = receive();
if (msg != null) {
// Process the message
}
```

Aspettare l'arrivo di un messaggio

Cosa non fare:

Una grossolana soluzione per mettere un **behaviour** in attesa di un messaggio potrebbe essere richiamare continuamente il metodo `receive()`. Questa soluzione comporta uno spreco del tempo di CPU.

Cosa fare:

Il metodo `block()` di un behaviour, lo rimuove dal *pool* di behaviour attivi dell'agente e lo mette in un insieme di behaviour bloccati.
Ogni volta che un messaggio viene ricevuto dall'agente, i behaviour bloccati vengono riattivati in modo da avere una *chance* di leggere e processare il messaggio

Esempio

```
public void action() {
        ACLMessage msg = myAgent.receive();
        if (msg != null) {
                // Process the message
        }
        else {
                block();
        }
}
```

Ricezione selettiva dei messaggi

Problema:

Il metodo `receive()` restituisce il primo messaggio ricevuto e lo toglie dalla coda. Se più behaviours stanno aspettando dei messaggi è possibile, utilizzando `receive()`, che sorgano problemi (un behaviour potrebbe prendere un messaggio non diretto a lui);

Soluzione:

è possibile richiedere di leggere solo i messaggi che rispettano determinate caratteristiche. Per fare questo si deve utilizzare un'istanza della classe `jade.lang.acl.MessageTemplate` come parametro del metodo `receive()`

Esempio

```
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");
public void action() {
        ACLMessage msg = myAgent.receive(tpl);
        if (msg != null) {
                // Process the message
        }
        else {
                block();
        }
}
```

Ricezione bloccante di messaggi

Ricezione bloccante:

La classe Agent fornisce il metodo `blockingreceive()` per la ricezione bloccante di messaggi;

Esistono implementazioni di questo metodo che accettano `MessageTemplate` e *timeout*;

Utilizzare `blockingreceive()` all'interno di un behaviour è pericoloso perché **viene bloccato l'intero agente** e quindi tutti i suoi behaviours.

E' Consigliabile:

- Utilizzare `receive()` + `Behaviour.block()` all'interno dei Behaviour;
- `blockingreceive()` per ricevere messaggi all'interno dei metodi `setup()` e `takeDown()` di un agente.

Protocolli di comunicazione

FIPA specifica dei protocolli di interazione per la conversazione tra agenti;

Per ogni conversazione JADE prevede agenti con ruoli di *Initiator* e *Responder* fornendo behaviours già implementati per entrambi i ruoli.

Queste classi possono essere trovate nel package `jade.proto`

- `AchieveREInitiator;`
- `SimpleAchieveREInitiator;`
- `AchieveREResponder;`
- `SimpleAchieveREResponder;`
- `ContractNetInitiator;`
- `ContractNetResponder;`
- …

Importante:

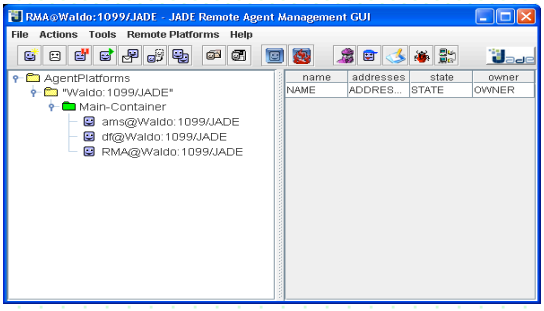
Non sono necessari ma sono una soluzione elegante e funzionale.

Agenti Fondamentali

In ogni piattaforma JADE esistono 2 agenti fondamentali residenti nel MainContainer:

- l' Agent Management System (AMS);
- Il Directory Facilitator (DF);

RMA è l'agente responsabile della GUI di JADE. Viene eseguito soltanto nel caso in cui venga richiesta l'interfaccia grafica



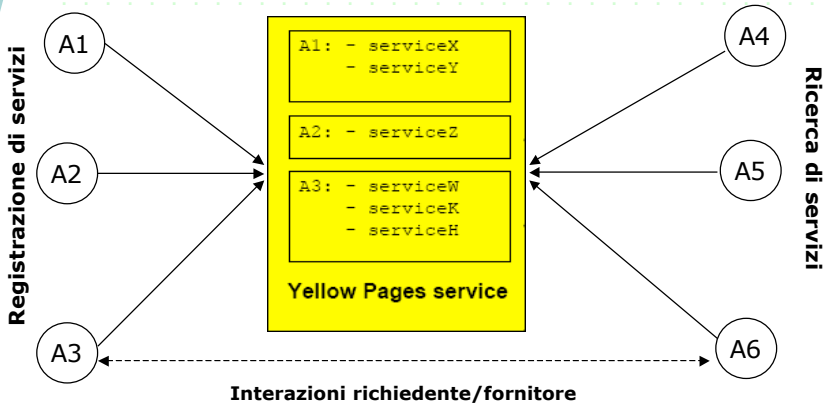
AMS – Agent Management System

Chi è e cosa fa?

- E' un elemento fondamentale di qualsiasi piattaforma basata su agenti che sia conforme alle specifiche FIPA;
- Svolge funzione di supervisore controllando l'accesso e l'uso della piattaforma da parte degli agenti;
- Ne esiste solo 1 in ogni piattaforma;
- Mantiene un registro degli identificatori degli agenti (AID) e del loro stato;
- Ogni agente deve registrarsi presso l'AMS per ottenere un AID valido. Tale registrazione avviene in modo automatico (trasparente al programmatore ed all'utente) appena un agente viene eseguito in un Container;

DF – Directory Facilitator

Il Directory Facilitator (DF) è l'agente che fornisce il servizio di pagine gialle di base della piattaforma.



Comunicare con il DF

Trovare il DF:

Il metodo `getDefaultDF()` della classe `Agent`, permette di determinare l'AID del DF di default.

Comunicare con il DF:

La classe `jade.domain.DFService` fornisce un insieme di metodi statici per la comunicazione con un Directory Facilitator

Questi permettono di compiere azioni di:

- Registrazione;
- Deregistrazione;
- Modifica;
- Ricerca;

di agenti per certi servizi presso il DF.

l'esecuzione di questi metodi blocca l'attività dell'agente finchè l'azione non è stata eseguita o un'eccezione di tipo `jade.domain.FIPAException` non è stata lanciata. In pratica l'agente rimane bloccato finchè non termina la sua "conversazione" con il DF.

DFService, un esempio

Registrazione

DFAgentDescription: descrive un agente nel catalogo del DF

ServiceDescription: descrive un servizio offerto da un agente

```
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setName(serviceName);
sd.setType("weather-forecast");
dfd.addServices(sd);
DFService.register(this, dfd);
```

Ricerca

Utilizzare DFAgentDescription e ServiceDescription al contrario.

DFService.search() restituisce un vettore di DFAgentDescription che soddisfano <template>

```
DFAgentDescription template = new DFAgentDescription();
                    ServiceDescription tsd = new ServiceDescription();
                    tsd.setType("weather-forecast");
                    template.addServices(tsd);
                    SearchConstraints sc = new SearchConstraints();
                    DFAgentDescription[] res = DFService.search(this,
                                                        template, sc);
```

Ontologie

Le informazioni contenute all'interno degli ACLMessage scambiati tra agenti sono stringhe.

Gli agenti sono entità Software ed è quindi poco "comodo" per loro gestire informazioni in formato unicamente testuale

La cosa migliore è convertire queste informazioni in oggetti Java.

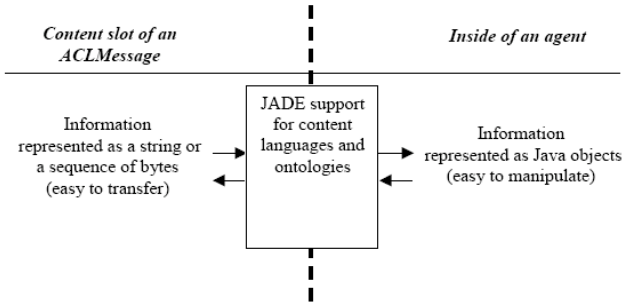
Ontologie

Ogni conversione è effettuata da un oggetto ContentManager istanza della classe `jade.content.ContentManager;`

Ogni Agente JADE possiede un ContentManager il cui riferimento gli è accessibile attraverso il metodo Agent.getContentManager()

Un ContentManager si avvale di:

- un'**ontologia** per verificare la correttezza dei contenuti di un messaggio ACL;
- un **content language codec** per effettuare le trasformazioni;



Ontologie

Per effettuare un adeguato controllo semantico dei messaggi nell'ambito di una conversazione, è stata fatta una classificazione degli elementi di un discorso

Questa classificazione deriva dal linguaggio ACL e si riferisce quindi al contenuto dei messaggi ACL scambiati tra agenti

- **Predicati:** Espressioni che dicono qualcosa del mondo, possono essere *true* o *false*. Esempio: (Works-for (Person: name John) (Company :name TILab))
- **Termini:** Espressioni che si riferiscono ad entità esistenti nel mondo di cui l'agente può parlare e su cui può "ragionare";
  - **Concetti:** Espressioni che indicano entità aventi strutture complesse. Esempio: (Person :name Andrea :age 24);
    - **Agent Action:** Speciali concetti che indicano possibili azioni di agenti. Esempio: (Sell (Book :title "The Hitchiker's Guide to the Galaxy") (Person :name Andrea))
  - **Aggregati:** Espressioni che indicano entità gruppi di altre entità;
  - **Variabili:** Espressioni che indicano elementi non noti a priori;
  - **Primitive:** Espressioni che indicano entità atomiche come stringhe e interi;

Problema

Ci interessa determinare gli agenti (se ce ne sono) residenti in un certo Container che offrono un certo servizio

Come fare?

Il DF può dirci quali sono gli agenti della piattaforma che offrono un particolare servizio ma non può sapere in quale Container risiedono…

L' AMS può dirci quali sono gli agenti residenti in un certo Container ma non può dirci quali servizi offrono…

Per ottenere quello che desideriamo dobbiamo utilizzare entrambi gli agenti

QueryAgentsOnLocation

- Il package `jade.domain.JADEAgentManagement` contiene concetti, azioni e predicati relativi alle azioni che possono essere richieste agli agenti AMS e DF.
- `Jade.domain.JADEAgentManagement.QueryAgentsOnLocation` rappresenta un'azione che può essere richiesta all'AMS per ottenere la lista di agenti eseguiti in un certo Container.

Linee Guida

- Ottenere un riferimento al ContentManager dell'agente;
- Istanziare un oggetto QueryAgentsOnLocation ed utilizzare il suo metodo setLocation per impostare il Container su cui si desidera effettuare la ricerca;
- Utilizzare il metodo fillContent del ContentManager per inizializzare il contenuto di un messaggio ACL con l'azione richiesta (indicare nel messaggio il fatto che utilizza l'ontologia JADEAgentManagement e il linguaggio SLCodec);
- Inviare il messaggio ACL all'AMS;
- Per gestire la risposta utilizzare il metodo extractContent del ContentManager ottenendo un oggetto ContentElement;
- Trasformare il ContentElement in un oggetto Result;

QueryPlatformLocationsAction

- `Jade.domain.JADEAgentManagement.QueryPlatformLocationsAction` rappresenta un'azione che può essere richiesta all'AMS per ottenere la lista di dei Container di una piattaforma.
- I Container sono oggetti di tipo `Location`;

Linee Guida

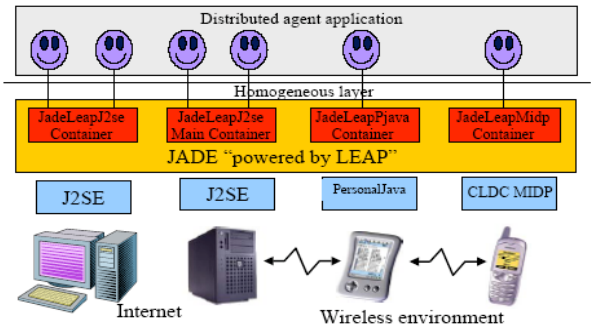
- Stesse operazioni del caso precedente ad eccezione del fatto che deve essere utilizzato un oggetto `QueryPlatformLocationsAction`. Quest'oggetto non richiede parametri aggiuntivi;
- Una volta ottenuto l'oggetto `Result` dalla risposta dell'AMS è possibile estrarre da questo un oggetto `List`, lista degli oggetti `Location` rappresentanti tutti i `Container` della piattaforma.

Esempio

```
ContentManager manager      = (ContentManager)getContentManager();
Codec codec = new SLCodec();
Ontology        ontology    = JADEManagementOntology.getInstance();
manager.registerLanguage(codec);
manager.registerOntology(ontology);
Action  qloc = new Action(getAMS(), new QueryPlatformLocationsAction());
ACLMessage req = new ACLMessage(ACLMessage.REQUEST);
req.setSender(getAID());
req.addReceiver(getAMS());
req.setLanguage(codec.getName());
req.setOntology(ontology.getName());
manager.fillContent(req,qloc);
send(req);
ACLMessage resp = blockingReceive();
ContentElement ce = getContentManager().extractContent(resp);
Result myresult = (Result) ce;
List loclist = myresult.getItems();
int indiceContainer =1;
Location aContainer = (Location)loclist.get(indiceContainer);
QueryAgentsOnLocation qal = new QueryAgentsOnLocation();
qal.setLocation(aContainer);
Action action2 = new Action(getAMS(), qal);
manager.fillContent(req,action2);
send(req);
ACLMessage resp2 = blockingReceive();
ContentElement ce2 = getContentManager().extractContent(resp2);
Result myresult2 = (Result) ce2;
jade.util.leap.Iterator agentsinlocation = myresult2.getItems().iterator();
```

JADE LEAP

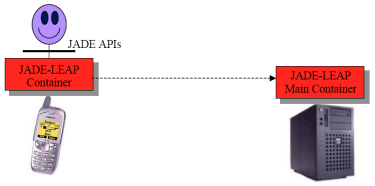
- Permette di eseguire agenti JADE su dispositivi mobili con ridotte capacità di calcolo (cellulari e palmari);
- Ha una implementazione interna differente ma fornisce lo stesso insieme di API;



JADE LEAP

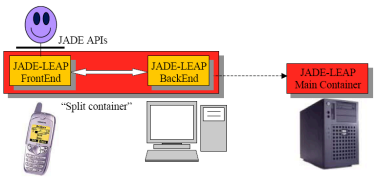
Stand-alone:

Nel dispositivo mobile viene eseguito un intero Container



Split:

Il Container è diviso in un FrontEnd (eseguito sul dispositivo mobile) ed in un BackEnd eseguito in un altro host

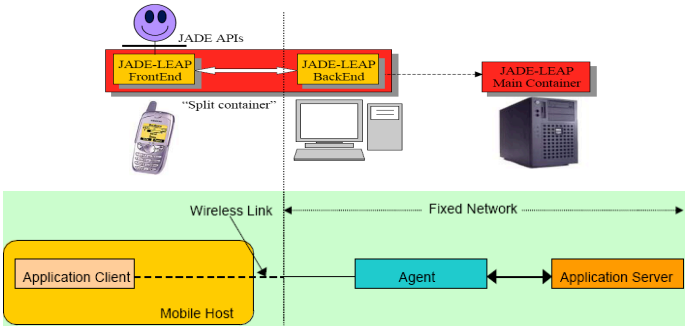


JADE Leap

Il BackEnd LEAP svolge la funzione di surrogato del client mobile nella rete fissa;

Svolge funzione di accodamento dei messaggi supportando operazioni caratterizzate da connessioni a tratti mancanti;

È un modello appropriato per client leggeri;



Consigli

Consultare:

- JADE Administrator's Guide: http://jade.tilab.com/doc/administratorsguide.pdf
- JADE Programmer's Guide: http://jade.tilab.com/doc/programmersguide.pdf
- Complete Api Documentation: http://jade.tilab.com/doc/api/index.html
- Basic Aspect of JADE Programming: http://www.cs.bath.ac.uk/~occ/agents_ecommerce/jade/
- LEAP User guide: http://jade.tilab.com/doc/LEAPUserGuide.pdf

Altre risorse:

Mailing list di Jade: jade-develop@sharon.cselt.it

Per iscriversi occorre riempire, durante la procedura di download di Jade, il form che permette l'scrizione alla mailing list.