Introduzione a JADE

Francesco Di Giunta

SOMMARIO

- Introduzione
- Architettura generale di un sistema multiagente (MAS) in JADE
- Architettura e implementazione di un agente JADE
- Comunicazione e interazione in JADE

Introduzione

- JADE (Java Agent DEvelopment framework)
- È un toolkit per lo sviluppo di MAS
- Sviluppato all'ex TILab (Telecom Italia Lab) di Torino
- http://jade.tilab.com
- Open source, licenza LGPL
- Implementato interamente in Java
- Prevede che gli agenti vengano sviluppati in Java

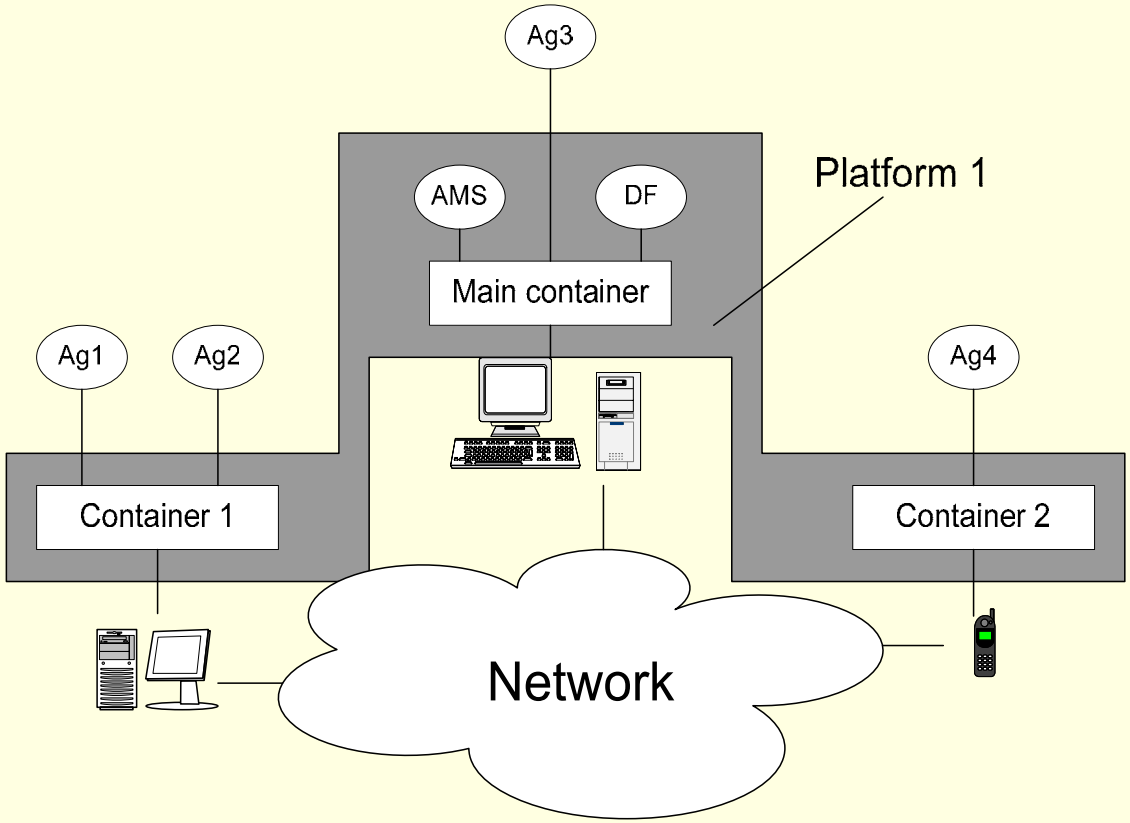
Introduzione
Principali caratteristiche di JADE

- Cosa include JADE:
  - Ambiente run time distribuito per MAS
  - Libreria di classi per la creazione di un MAS
  - Strumenti per debugging, gestione, monitoraggio di un MAS
- JADE prevede un'architettura "debole" di agente: come viene strutturato un agente lo decide quasi del tutto il progettista
- Interoperabilità: conformità alle specifiche FIPA per la comunicazione e l'interazione fra agenti
- Supporto alla mobilità di agenti
- Supporto per ontologie e content language definiti dal progettista
- Sistema "leggero": gira anche su dispositivi poco potenti

Architettura di un MAS in JADE

- L’ambiente run time di un MAS in JADE è detto *platform*
- Una platform è distribuita in un certo numero di unità dette *container*, collegate in rete
- In ogni container si trova un certo numero di *agenti*
- In una platform è sempre presente un (unico) *main container* con funzioni speciali (corrispondenza 1:1 platform-main_container)

Architettura di un MAS in JADE



Funzioni del main container

- E’ il primo container ad essere lanciato in una platform
- Gli altri container devono registrarsi con esso quando vengono lanciati
- Contiene due agenti speciali (creati da JADE e non dal progettista):
  - AMS (*Agent Management System*)
    - E’ l’“autorità” della piattaforma (es. eliminazione di altri agenti)
    - Gestisce l’unicità dei nomi degli agenti nella piattaforma
  - DF (*Directory Facilitator*)
    - Fornisce le pagine gialle del MAS

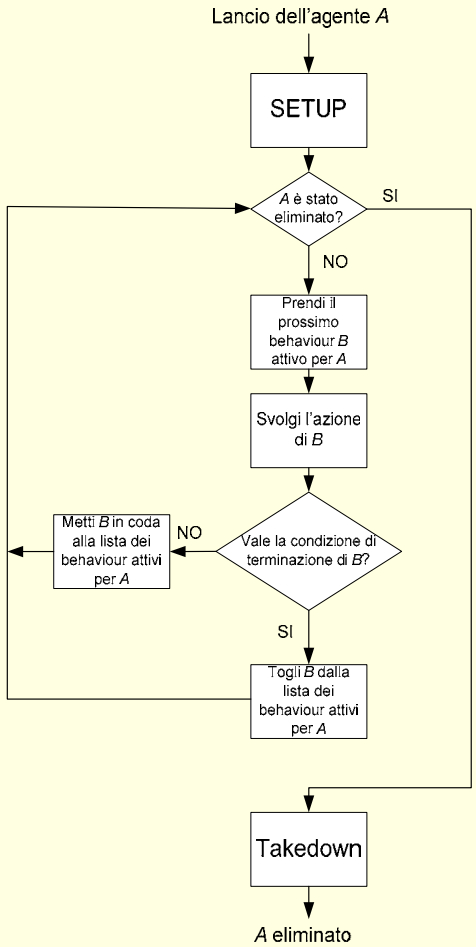
Architettura di un agente

- Un agente in JADE ha:
  - Un nome <nick_agente>@<nome_platform>
    - Il nome è unico
    - <nick_agente> è assegnato da chi lancia l'agente
    - Il nome completo è assegnato automaticamente dall'AMS (con controlli di unicità) all'atto del lancio dell'agente
  - Una procedura di inizializzazione (*setup*)
    - Definita dal progettista dell'agente
    - Eseguita dal sistema all'atto del lancio dell'agente
  - Una lista di *behaviour* attivi associati: il "cuore" di ciò che fa un agente (vedi slide successiva)
  - Una procedura di terminazione (*takedown*)
    - Definita dal progettista
    - Eseguita dal sistema all'atto dell'eliminazione dell'agente dalla piattaforma

Architettura di un agente behaviour

- Un behaviour è il “cuore” di ciò che fa un agente: ciascun behaviour è pensato per eseguire un particolare compito
- I behaviour sono definiti dal progettista
- Un behaviour è costituito da:
  - Un’azione
  - Una condizione di terminazione
- I behaviour sono definiti indipendentemente dagli agenti
- Vengono associati agli agenti dinamicamente:
  - Nel setup
  - Dall’interno di altri behaviour

Architettura di un agente



Come si costruisce un agente

- JADE mette a disposizione alcune classi Java
- Costruire un agente consiste nell'estendere opportunamente tali classi
- Classi principali:
  - jade.core.Agent
    - Un agente generico, le sue azioni di inizializzazione e di terminazione
  - jade.core.behaviours.behaviour
    - Un behaviour associabile ad un agente, l'azione del behaviour, la sua condizione di terminazione

Come si costruisce un agente
Classe Agent

- Si trova nel package jade.core
- Per costruire un agente si estende la classe **Agent**

```
import jade.core.Agent;
public class MioAgente extends Agent {…
```

- Si implementa il metodo **setup()**, inteso contenere le inizializzazioni dell'agente (es. l'attivazione di un primo behaviour)

```
import jade.core.Agent;
public class MioAgente extends Agent {
    protected void setup() {
    …
    }
}
```

  setup() è invocato dall'ambiente run time quando si lancia un agente

- Si compila la classe creata:

```
javac –classpath <JADE-classes> MioAgente.java
```

- Si ottiene una classe le cui istanze sono agenti del tipo MioAgente definito dal progettista

Come si costruisce un agente
Classe Agent

- Altri metodi di Agent:
  - getArguments()
    - Un agente può essere lanciato con una lista di parametri. getArguments() restituisce tale lista
  - getAID()
    - Un agente ha un Agent IDentifier contenente il suo nome (e gli indirizzi della piattaforma). getAID() restituisce un oggetto di classe jade.core.AID contenente l'agent identifier dell'agente. getAID().getName() restituisce il nome dell'agente
  - addbehaviour()
    - Riceve come argomento un behaviour (istanza di una classe che estende la classe behaviour; vedi più avanti) e lo aggiunge ai behaviour attivi per l'agente
  - doDelete()
    - Elimina l'agente
  - takeDown()
    - E' implementato dal progettista. E' inteso contenere procedure da eseguire subito prima della terminazione dell'agente. E' invocato dall'ambiente run time dopo il doDelete()

Come si costruisce un agente
Un esempio

```
import jade.core.Agent;
public class MioAgente extends Agent {
    protected void setup(){
        System.out.println("L'agente " + getAID().getName()+" è stato lanciato");
        Object[] args=getArguments();
        if (args != null) {
            System.out.println("con " + args.length + " argomenti");
        }
        doDelete();
    }
    protected void takeDown() {
        System.out.println("L'agente " + getAID().getName() + " è stato
    eliminato");

    }
}
```

Come si costruisce un agente
Classe behaviour

- Si trova nel package jade.core.behaviours
- Per costruire un behaviour si estende la classe **behaviour**

```
import jade.core.behaviours.behaviour;
public class Miobehaviour extends behaviour {…
```

- Si implementa il metodo **action()**, che contiene le istruzioni che definiscono l'operazione da svolgere nell'esecuzione del behaviour

```
import jade.core.behaviours.behaviour;
public class Miobehaviour extends behaviour {
    protected void action() {
        System.out.println("Sto compiendo un'azione");
    }
…
}
```

action() è invocato dall'ambiente run time ogni volta che il behaviour viene schedulato per l'esecuzione

- Si implementa il metodo **done()**, che definisce la condizione di terminazione del behaviour

```
…
public boolean done() {
  return true;
}
```

Come si costruisce un agente
Associare un behaviour a un agente

- Si usa il metodo addbehaviour della classe Agent
- Lo si può fare nel setup o all'interno di un altro behaviour

```
protected void setup() {
    addbehaviour(new Miobehaviour());
    …
}
```

- In un behaviour la variabile myAgent punta all'agente che sta eseguendo il behaviour

```
public void action() {
    System.out.println("Questo behaviour viene eseguito adesso
    dall'agente" + myAgent.getAID().getName());
    …
}
```

Come si costruisce un agente
Esempio: i behaviours

```
import jade.core.behaviours.behaviour;
public class behaviour1 extends behaviour{
  public void action(){
    System.out.println("L'agente “ + myAgent.getAID().getName() + " sta eseguendo il behaviour 1");
    myAgent.addbehaviour(new behaviour2());
  }
  public boolean done(){
    return true;
  }
}

import jade.core.behaviours.behaviour;
public class behaviour2 extends behaviour{
  private int iterazione=0;
  public void action(){
    System.out.println("L'agente " + myAgent.getAID().getName() + “ sta eseguendo il behaviour 2");
    iterazione++;
  }
  public boolean done(){
    return iterazione==2;
  }
}
```

Come si costruisce un agente

Esempio: l'agente

```
import jade.core.Agent;
public class AgenteDiEsempio extends Agent{
  private String tipoDiBehaviour;
  protected void setup() {
    Object[] args=getArguments();
    System.out.println("L'agente " + getAID().getName() + " inizia");
    if (args!=null && args.length>0)
      if (args[0].equals("inizia_da_1"))
        addbehaviour(new behaviour1());
      else
        addbehaviour(new behaviour2());
    else{
      System.out.println("Non hai inserito parametri di ingresso!");
      doDelete();
    }
  }
  protected void takeDown() {
    System.out.println("L'agente " + getAID().getName() + " termina");
  }
}
```

Come si costruisce un agente
Esempio: alcune esecuzioni

Lancio un agente di classe AgenteDiEsempio nella piattaforma pc-digiunta:1099/JADE chiamandolo col nickname “AgenteInutile”

- Se non gli passo parametri di ingresso ottengo
  L'agente AgenteInutile@pc-digiunta:1099/JADE inizia
  Non hai inserito parametri di ingresso!
  L'agente AgenteInutile@pc-digiunta:1099/JADE termina
- Se gli passo parametri il cui primo è “inizia_da_1” ottengo
  L'agente AgenteInutile@pc-digiunta:1099/JADE inizia
  L'agente AgenteInutile@pc-digiunta:1099/JADE sta eseguendo il behaviour 1
  L'agente AgenteInutile@pc-digiunta:1099/JADEsta eseguendo il behaviour 2
  L'agente AgenteInutile@pc-digiunta:1099/JADEsta eseguendo il behaviour 2
- Se gli passo parametri il cui primo non è “inizia_da_1” ottengo
  L'agente AgenteInutile@pc-digiunta:1099/JADE inizia
  L'agente AgenteInutile@pc-digiunta:1099/JADEsta eseguendo il behaviour 2
  L'agente AgenteInutile@pc-digiunta:1099/JADEsta eseguendo il behaviour 2

Come si costruisce un agente
Ancora sui behaviour

- Un agente può avere molti behaviour attivi
- I behaviour attivi sono eseguiti concorrentemente con politica FIFO:
  - eseguo l'azione del primo behaviour
  - metto il primo behaviour in coda (solo se done()==false)
  - eseguo l'azione del secondo behaviour
  - …
- La concorrenza è di tipo non-preemptive:
  - L'azione del behaviour in esecuzione non viene interrotta da altri behaviour
  - In tal modo un agente può essere eseguito come un singolo java thread
  - Se l'azione di un behaviour è un ciclo infinito, da tale behaviour non si esce mai
  - È compito del progettista definire behaviour opportunamente "collaborativi"
  - Questa limitazione è stata rimossa nella versione 3.2 di JADE e successive!
- Un behaviour può essere "bloccato" (metodo block() di behaviour) per essere riesumato quando:
  - l'agente riceve un messaggio
  - scade un timeout
  - viene invocato il metodo restart()

Come si costruisce un agente
Ancora sui behaviour

Il comportamento di un agente è inglobato nei behaviour

+

La scrittura dei behaviour è a carico del progettista

=

La definizione del comportamento di un agente è del tutto a carico del progettista

- In particolare, l’”intelligenza” dell’agente è a carico del progettista
- Due facilitazioni:
  - integrazione con JESS
  - estensioni predefinite della classe behaviour

Come si costruisce un agente Integrazione con JESS (cenni)

- **JESS** (Java Expert System Shell): è un sistema per la **programmazione basata su regole di produzione**
- Nella programmazione basata su regole:
  - il programmatore asserisce:
    - alcuni **fatti**
    - **regole** del tipo:“se vale il fatto *F* allora vale il fatto *G*”
  - un **motore inferenziale** asserisce nuovi fatti deducendoli dai fatti già asseriti e dalle regole
- Utile per definire il comportamento di un agente in modo dichiarativo in termini di “conoscenza” dell’agente
- JADE permette l’integrazione con JESS. Per esempio un agente può usare un sistema di regole JESS per rispondere a messaggio

Come si costruisce un agente behaviour predefiniti

- JADE mette a disposizione alcune classi che estendono behaviour implementandone action() o done()
- Spesso tali classi introducono metodi più specifici che il progettista deve implementare
- Le principali:
  - OneShotbehaviour
    - done() restituisce sempre true
    - l'azione viene eseguita solo una volta
  - Cyclicbehaviour
    - done() restituisce sempre false
    - Il behaviour è sempre attivo
  - Compositebehaviour
    - Compone behaviour "figli"
    - L'azione consiste nella chiamata dell'azione di un behaviour figlio
    - Il behaviour figlio prescelto dipende dalla politica di scheduling:
      - Sequentiabehaviour: figli eseguiti in serie
      - Parallelbehaviour: figli eseguiti parallelamente
      - FSMbehaviour: figli eseguiti secondo una macchina a stati finiti

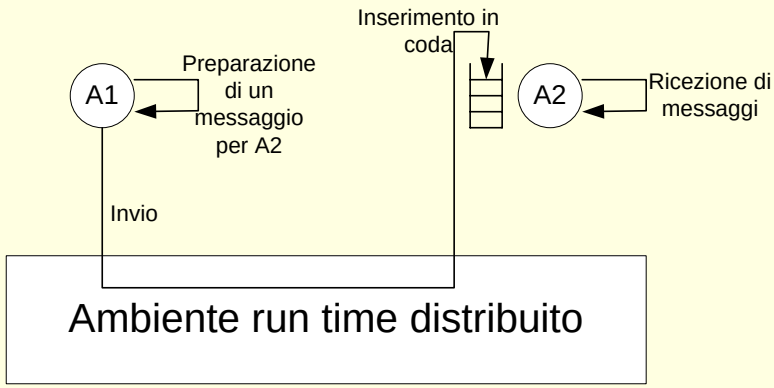
Comunicazione e interazione

Vedremo:

- Comunicazione
  - Paradigma di comunicazione
  - Implementazione dello scambio di messaggi
- Interazione
  - Protocolli di interazione FIPA
  - Implementazione dei protocolli

Paradigma di comunicazione

- I messaggi scambiati fra agenti sono messaggi FIPA ACL
- Meccanismo: passaggio asincrono di messaggi
  - Ogni agente ha una coda di messaggi entranti
  - La effettiva lettura dei messaggi è ad arbitrio dell'agente
  - Un agente può:
    - leggere il primo messaggio in coda
    - leggere il primo messaggio che soddisfa certi requisiti
  - Un behaviour può essere bloccato in attesa della ricezione di un messaggio: sincronizzazione
- L'invio di messaggi gode della trasparenza della collocazione fisica di mittente e destinatario (garantita dall'ambiente run time distribuito)



Messaggi in FIPA ACL

- Campi di un messaggio FIPA:
  - sender
  - receiver
  - performative
    - request
    - inform
    - query_if
    - cfp
    - propose
    - accept_proposal
    - reject_proposal
    - …
  - content
  - language
  - ontology
  - conversation_id
  - protocol
  - …

Implementazione di messaggi Classe ACLMessage

- Un messaggio in JADE è un'istanza della classe jade.lang.acl.ACLMessage
- Metodi di ACLMessage
  - **addReceiver()**: riceve in ingresso un AID ed aggiunge il corrispondente agente fra i destinatari
  - **setPerformative()**: riceve in ingresso una costante rappresentante un tipo di atto performativo e lo pone come *performative* del messaggio
  - **setContent()**: riceve in ingresso una stringa e la pone come *content* del messaggo
  - **setLanguage()**
  - …
  - **getSender()**
  - **getAllReceiver()**
  - **getContent()**
  - **getLanguage()**
  - …
  - **createReply()**: crea un messaggio di risposta a quello cui si applica, ponendo agli opportuni valori campi come *receiver*, *conversation_id* ecc.
- Taluni campi possono essere riempiti direttamente passandone il valore ai costruttori di ACLMessage

Implementazione di messaggi
Invio e ricezione

- Si usano i metodi **send()** e **receive()** della classe Agent
- **send()** riceve in ingresso un ACLMessage, aggiunge l'opportuno valore al campo *sender* ed invia il messaggio ai destinatari
- **receive()** preleva dalla coda il primo messaggio e lo restituisce (dà **null** se la coda è vuota)

Implementazione di messaggi
Esempio: un behaviour per mandare messaggi

```
import jade.core.behaviours.OneShotbehaviour;
import jade.lang.acl.ACLMessage;
import jade.core.AID;
public class EmergencySendbehaviour extends OneShotbehaviour{
  public void action() {
    ACLMessage msg=new ACLMessage(ACLMessage.INFORM);
    msg.addReceiver(new AID("Pompiere",AID.ISLOCALNAME));
    msg.setLanguage("Italiano");
    msg.setOntology("Emergenza");
    msg.setContent("Al fuoco");
    myAgent.send(msg);
  }
}
```

Implementazione di messaggi
Esempio: un behaviour per ricevere messaggi

```
import jade.core.behaviours.Cyclicbehaviour;
import jade.lang.acl.ACLMessage;
import jade.core.AID;

public class EmergencyReceivebehaviour extends Cyclicbehaviour{
  public void action() {
    ACLMessage msg=myAgent.receive();
    if (msg!=null) {
      String contenuto=msg.getContent();
      if (contenuto.equalsIgnoreCase("Al fuoco")) {
        System.out.println("L'agente " + msg.getSender().getName() +
                           " ha segnalato un incendio");
        System.out.println("L'agente " + myAgent.getAID().getName() +
                           " attiva la procedura antincendio");
      }
    }
  }
}
```

Implementazione di messaggi Esempio: due agenti

```
import jade.core.Agent;
public class AgenteAllarmato extends Agent {
  protected void setup() {
    addbehaviour(new EmergencySendbehaviour());
  }
}
```

```
import jade.core.Agent;
public class AgentePompiere extends Agent {
  protected void setup() {
    addbehaviour(new EmergencyReceivebehaviour());
  }
}
```

Implementazione di messaggi Esempio: una esecuzione

- Nella piattaforma pc-digiunta:1099/JADE lancio
  - un agente di classe AgentePompiere chiamandolo col nickname “Pompiere”
  - Un agente di classe AgenteAllarmato chiamandolo “Custode”
- Ottengo il seguente output:

```
L'agente Custode@pc-digiunta:1099/JADE ha segnalato un incendio
L'agente Pompiere@pc-digiunta:1099/JADE attiva la procedura antincendio
```

- NB: se avessi lanciato prima custode non avrei ottenuto alcun output

Implementazione di messaggi
Un problema

- Problema: se non ci sono messaggi in arrivo EmergencyReceivebehaviour gira a vuoto consumando CPU
- Soluzioni possibili:
  - Uso il metodo **block()** di behaviour:

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

  - Uso il metodo **blockingReceive()** di Agent al posto del metodo Receive()
- Differenza fra le due soluzioni:
  - block() blocca solo un behaviour dell'agente
  - blockingReceive() blocca tutti i behaviour dell'agente

Implementazione di messaggi Message template

- Un ***message template*** è uno schema di messaggio con il quale messaggi concreti possono fare matching
- Utili per la ricezione selettiva di messaggi: un agente riceve il primo messaggio in coda che fa matching con un template
- Sono implementati come istanze di

  Jade.lang.acl.MessageTemplate
- Li si usa come parametri di receive()
- Metodi applicabili:
  - **MatchPerformative()**
  - **MatchSender()**
  - …
- Con **and**, **or** e **not** si possono creare template composti

Implementazione di messaggi Message template: un esempio

Nell'esempio dell'incendio, l'agente pompiere potrebbe essere interessato solo a ricevere messaggi di tipo INFORM e in italiano. Ciò è ottenibile modificando EmergencyReceivebehaviour così:

```
public void action() {
   MessageTemplate MT1=MessageTemplate.MatchPerformative(ACLMessage.INFORM);
   MessageTemplate MT2=MessageTemplate.MatchLanguage("Italiano");
   MessageTemplate MT3=MessageTemplate.and(MT1,MT2);
   ACLMessage msg=myAgent.receive(MT3);
   if (msg!=null) {
      …
   }
}
```

Protocolli di interazione

- Sono schemi prestabiliti di conversazione fra agenti
- Vantaggio: costituiscono soluzioni di provata affidabilità per molte esigenze
- Sono standardizzati dalla FIPA
- JADE ne supporta alcuni
- Ciascun protocollo prevede
  - Due ruoli per agenti:
    - *Initiator*: l'agente che dà avvio alla conversazione
    - *Responder*: ogni altro agente
  - I tipi di messaggi da scambiare
  - Le fasi in cui tali messaggi vanno scambiati

Protocolli di interazione
Esempio: FIPA_Request

- L'Initiator richiede ai Responder di compiere un'azione (REQUEST)
- Ciascun Responder accetta o meno (AGREE / REFUSE)
- Chi accetta, non appena compie l'azione o incontra problemi, ne informa l'Initiator (INFORM(Done) / FAILURE)

Implementazione dei protocolli

- In JADE i protocolli sono implementati da behaviour predefiniti (nel package jade.proto)
- In particolare, il protocollo *X* (es. ContractNet) è implementato da
  - *X*Initiator (es. ContractNetInitiator)
  - *X*Responder (es. ContractNetResponder)
- Quindi, un agente per essere Initiator di un protocollo *X* deve avere *X*Initiator fra i suoi behaviour attivi. Analogamente per i Responder

Implementazione dei protocolli

- Le classi *X*Initiator/Responder hanno diversi metodi che
  - Vengono invocati automaticamente nella opportuna fase del protocollo
  - Devono essere implementati dal progettista del MAS, che decide cosa va fatto esattamente in ogni fase
- *X*Initiator ha metodi di tipo ***handle***: definiscono cosa l'Initiator deve fare quando riceve le varie risposte dei Responder
- *X*Responder ha metodi di tipo ***prepare***: definiscono quali messaggi un Responder deve inviare nelle varie fasi

Implementazione dei protocolli
Un esempio

- Voglio che un agente A1 operi una richiesta ad A2
- Voglio usare il protocollo FIPA_Request per ragioni di robustezza
- JADE supporta la FIPA_Request col nome AchieveRE (=Achieve Rational Effect)
- Doto A1 del behaviour AchieveREInitiator
- Doto A2 di AchieveREResponder

Implementazione dei protocolli
Un esempio

- Un’istanza di AchieveREInitiator deve essere costruita con la richiesta iniziale come parametro del costruttore
- Devo implementare opportunamente i metodi
  - handleRefuse()
  - handleAgree()
  - handleFailure()
  - handleInform()
- In un behaviour di A1 inserisco codice del seguente tipo:

```
ACLMessage req=new ACLMessage(ACLMessage.REQUEST);
req.addReceiver(new AID(“A2”,AID.ISLOCALNAME));
…
myAgent.addbehaviour(new AchieveREInitiator(myAgent,req) {
                                                    protected void handleRefuse(ACLMessage refs) {
                                                                System.out.println(“Richiesta negata”);
                                                    }
                                                    …
                                        }
                                );
```

- Analogamente, per A2 devo:
  - implementare i metodi prepareResponse(), prepareResultNotification(), ecc. di AchieveREResponder
  - associare AchieveREResponder ad A2 usando un costruttore con un message template come parametro