

Plate-forme JADE

Agents - SMA

Claude Moulin

Université de Technologie de Compiègne

IA04

Sommaire

- 1 Introduction
 - Conteneur d'agents
 - Outils
- 2 Structure d'un agent
 - Cycle de vie
 - Comportement d'un agent
- 3 Communication entre agents
 - Messages
 - Principaux behaviours

Sommaire

- 1 Introduction
- 2 Structure d'un agent
- 3 Communication entre agents

Sommaire

- 1 Introduction
 - Conteneur d'agents
 - Outils
- 2 Structure d'un agent
 - Cycle de vie
 - Comportement d'un agent
- 3 Communication entre agents
 - Messages
 - Principaux behaviours

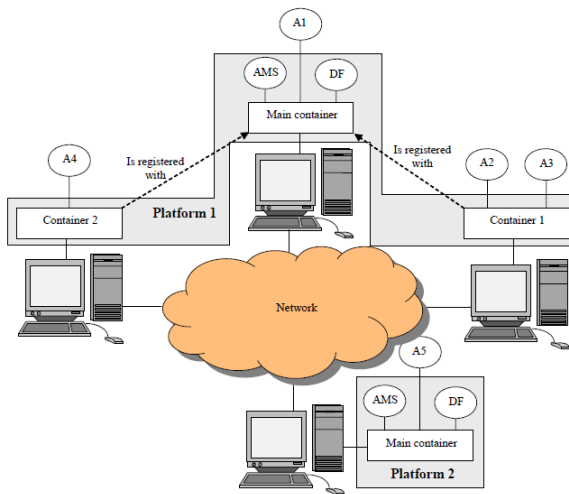
JADE

- JADE (Java Agent DEvelopment Framework) est une plateforme de développement de systèmes multi-agents implémentée en Java. <http://jade.tilab.com/>
- Elle contient une bibliothèque de classes utiles pour développer des agents.
- JADE est distribué par Telecom Italia.
- JADE est un ensemble d'outils qui permettent le débogage et le déploiement, l'administration et le monitoring de SMA.
- Il suffit d'attacher au classpath de l'application ou du projet le fichier jade.jar que l'on trouve dans la distribution de JADE.

Conteneur d'agents

- Un conteneur d'agents est un environnement qui contient plusieurs agents.
- Une plate-forme agent est l'ensemble des conteneurs actifs.
- Le conteneur principal :
 - Conteneur particulier lancé en premier.
 - Les autres conteneurs s'enregistrent auprès de lui (station hôte, port)

Plateformes



Agents système : AMS

- L'agent AMS (Agent Management System)
 - assure le service de nomage (chaque agent a un nom unique l'identifiant),
 - peut créer et supprimer des agents.

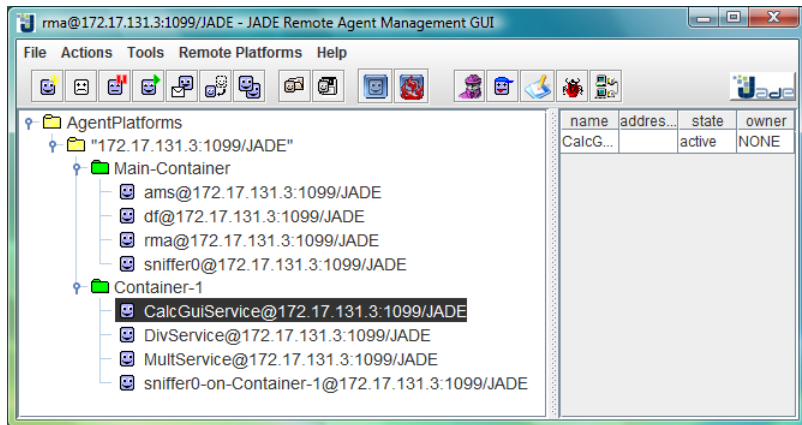
Agents système : DF

- L'agent DF (Directory Facilitator) :
 - assure le service de pages jaunes
 - permet à un agent de trouver d'autres agents avec lesquels communiquer.

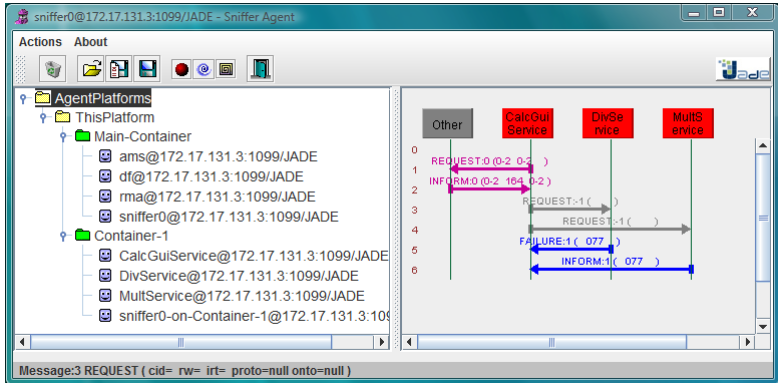
Sommaire

- 1 Introduction
 - Conteneur d'agents
 - Outils
- 2 Structure d'un agent
 - Cycle de vie
 - Comportement d'un agent
- 3 Communication entre agents
 - Messages
 - Principaux behaviours

Gestion des agents



Messages



Création du conteneur principal

Imports :

```
import jade.core.ProfileImpl;  
import jade.core.Runtime;  
import jade.wrapper.AgentContainer;  
import jade.wrapper.AgentController;
```

Code :

```
public static String MAIN_PROPERTIES_FILE = "...";  
Runtime rt = Runtime.instance();  
Profile p = null;  
try{  
    p = new ProfileImpl(MAIN_PROPERTIES_FILE);  
    AgentContainer mc = rt.createMainContainer(p);  
}  
catch(Exception ex) {...}
```

Fichier de propriétés

```
main=true  
gui=true  
platform-id=tdia04  
#local-port=1099
```

Création d'un conteneur secondaire

Un ou plusieurs conteneurs secondaire peuvent être lancés sur la même station ou une autre station.

```
Runtime rt = Runtime.instance();  
// host ; port (-1) ; platformId ; false (non main)  
Profile p = new ProfileImpl(<HOSTNAME>,-1,null,false);  
ContainerController cc = rt.createAgentContainer(p);  
try {  
    AgentController ac = cc.createNewAgent("Hello",  
        "tpl.agent.HelloAgent", null);  
    ac.start();  
    ...  
}  
catch(Exception ex) {...}
```

Création d'un conteneur secondaire (avec fichier)

```
Runtime rt = Runtime.instance();
ProfileImpl p = null;
try {
    p = new ProfileImpl(SECONDARY_PROPERTIES_FILE);
    ContainerController cc = rt.createAgentContainer(p);
    AgentController ac = cc.createNewAgent("Hello",
        "tpl.agent.HelloAgent", null);
    ac.start();
} catch (Exception ex) {
    ex.printStackTrace();
}
```


Fichier de propriétés

```
main=false  
gui=false  
container-name = tdtp  
host = 172.17...  
#local-port=1099
```

Sommaire

- 1 Introduction
- 2 Structure d'un agent**
- 3 Communication entre agents

Sommaire

- 1 Introduction
 - Conteneur d'agents
 - Outils
- 2 Structure d'un agent
 - Cycle de vie
 - Comportement d'un agent
- 3 Communication entre agents
 - Messages
 - Principaux behaviours

Identification

- Chaque agent est accessible uniquement grâce à son identifiant (instance de la classe `AID`).
- La méthode `getAID()` permet à un agent de retrouver son identifiant.
- Un nom d'agent est de la forme
`<nickname>@<platform-name>`
- Le surnom (nickname) est donné lors de la création de l'agent.

```
AgentController ac = cc.createNewAgent("Hello",  
    "tp1.agent.HelloAgent", null);
```

- Nom de plateforme par défaut : `<host>:<port>/JADE`.

Initialisation : `setup()`

- Une fois créé dans un conteneur un agent est initialisé.
- La méthode `setup()` est appelée après sa création. Elle sert à initialiser un agent.
- Exemple d'actions à exécuter :
 - Recherche des destinataires de messages ;
 - Récupération des arguments ;
 - Installation de certains behaviours.
 - Récupération des paramètres définis dans le fichier utilisé lors de la création du conteneur.

Exemple

- Agent capable de faire des produits donnés sous la forme $a \times b$;
- le signe de la multiplication donné en argument

```
AgentController ac = mc.createNewAgent(  
    "MultAgent", "cours.MultAgent", new Object[]{"x"});  
ac.start();
```

- Code principal de la classe agent :

```
public class MultAgent extends Agent {  
    protected void setup() {  
        System.out.println(getLocalName() + "--> Installed");  
        Object[] args = getArguments();  
        String op = (String) args[0];  
        addBehaviour(new MultBehaviour(this, op));  
    }  
}
```

Sommaire

- 1 Introduction
 - Conteneur d'agents
 - Outils
- 2 Structure d'un agent
 - Cycle de vie
 - Comportement d'un agent
- 3 Communication entre agents
 - Messages
 - Principaux behaviours

Tâche d'un agent

- Le travail d'un agent est exécuté au travers de "behaviours".
- Un behaviour est implementé comme un objet d'une classe qui étend la classe `Behaviour`.
- On peut écrire cette classe directement dans la classe `Agent`
- Pour qu'un agent exécute la tâche implémentée par un behaviour il suffit d'ajouter le behaviour à l'agent dans le `setup()` :

```
protected void setup() {  
    ...  
    addBehaviour(new MultBehaviour(this,op));  
}
```

- Un agent peut avoir plusieurs behaviours en parallèle.
- Un behaviour peut aussi installer d'autre behaviours.

Tâche d'un agent

- Chaque classe Behaviour doit implémenter la méthode `action()` qui définit la suite des instructions exécutées lorsque le behaviour est actif.
- La planification des behaviour déclenche cette méthode `action()` une fois que le behaviour est ajouté.
- La méthode booléenne `done()` spécifie si un behaviour est terminé et être retiré du pool des behaviours d'un agent.

Organisation du SMA

- Chaque agent JADE est exécuté dans un seul thread.
- L'exécution des threads Java dans la JVM est préemptive. Un certain temps est alloué par la JVM à chaque thread (chaque agent) qui exécute des instructions puis la JVM passe au thread suivant et ainsi de suite tant que les threads sont actifs.
- Cette organisation permet à plusieurs agents d'être actifs sur une même JVM donc dans un même SMA.

Organisation d'un agent

- Chaque agent JADE est exécuté dans un seul thread.
- Chaque agent peut posséder plusieurs behaviours en parallèle.
- Le scheduling des behaviours dans un agent n'est pas préemptif mais coopératif.
- Lorsqu'un behaviour entre en exécution, sa méthode `action()` est appelée et s'exécute jusqu'à son terme.
- C'est le programmeur qui définit quand un agent arrête l'exécution d'un behaviour pour passer à celle du suivant.

Structure d'un comportement

- Il faut prévoir des actions courtes qui puissent s'arrêter et qui puissent reprendre.
- Dans certains cas on peut prévoir un comportement en plusieurs pas.
 - A chaque prise en main, un pas est exécuté et rend la main.
 - La méthode `action()` apparaît comme un test simple ou un switch.

Exemple

```
public void action() {  
    switch(step) {  
        case 0 :  
            ...  
            step++;  
        break;  
        case 1 :  
            ...  
            if (...)  
                step = 0;  
            else  
                step++;  
            break;  
    }
```

Avantages

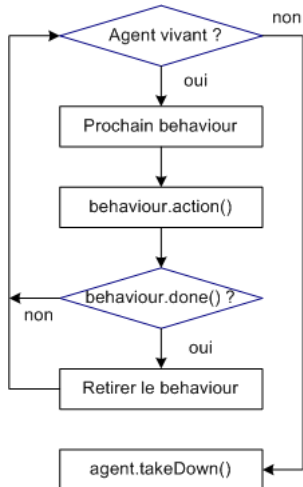
- Un seul thread Java par agent.
- Elimination des problèmes de synchronisation lors d'accès aux mêmes ressources.
- Meilleures performances car changer de behaviour est plus rapide que changer de thread Java.
- La persistance des agents et leur mobilité sont mieux assurées.

Scheduling

- Si l'on suppose qu'un agent possède 3 behaviours actifs :

```
b1.action() ;  
b2.action() ;  
b3.action() ;  
b1.action() ;  
b2.action() ;  
b3.action() ;  
b1.action() ;  
...
```

Scheduling



Sommaire

- 1 Introduction
- 2 Structure d'un agent
- 3 Communication entre agents**

Sommaire

1 Introduction

Conteneur d'agents

Outils

2 Structure d'un agent

Cycle de vie

Comportement d'un agent

3 Communication entre agents

Messages

Principaux behaviours

Attente d'un message

- Un agent peut exécuter plusieurs tâches mais est en général amené à attendre des messages pour entreprendre une action.
- Un agent peut récupérer un message (et donc le retirer) à partir de sa boîte à lettres :
 - Structure de file (first in first out).
- Pour éviter qu'un behaviour soit exécuté lorsqu'il n'y a pas de messages, il est possible et fortement conseillé de bloquer le behaviour.
- L'arrivée d'un nouveau message le débloque.

L'agent prévoit d'attendre des messages

```
public void action() {  
    ACLMessage message = receive();  
    if (message != null) {  
        // instructions  
    };  
    else  
        block();  
}
```

Plusieurs types de messages

```
public void action() {  
    ACLMessage message = receive(<template>);  
    if (message != null) {  
        // instructions  
    };  
    else  
        block();  
}
```

Un behaviour par type de message attendu.

L'agent attend une réponse à une requête

```
public void action() {  
    switch(step) {  
        case 0 :  
            // envoi d'un message  
            step++;  
            break;  
        case 1 :  
            ACLMessage resultmessage = receive(<template>);  
            if (resultmessage != null) {  
                // instructions  
            }  
            else  
                block();  
            break;  
    }  
}
```

Éléments d'un message

- Un message possède un certain nombre d'attributs parmi lesquels :
 - le ou les destinataires,
 - le type du message,
 - le contenu.

Envoi d'un d'un message

```
ACLMessage message =  
    new ACLMessage (ACLMessage.REQUEST) ;  
message.addReceiver (  
    new AID ("MULT", AID.ISLOCALNAME) ) ;  
message.setContent ("10 x 25") ;  
send (message) ;
```


Agent multiplicateur

Attend des requêtes de calcul :
analyse ; calcule ; répond.

```
public void action() {  
    ACLMessage message = receive();  
    if (message != null) {  
        answer(message);  
    }  
    else  
        block();  
}
```

Réponse

```
private void answer(ACLMessage message) {  
    String par = message.getContent();  
    ACLMessage reply = message.createReply();  
    if (par.contains("<op>")) {  
        String[] parameters = par.split("<op>");  
        int n = Integer.parseInt(parameters[0].trim())  
            * Integer.parseInt(parameters[1].trim());  
        reply.setPerformative(ACLMessage.INFORM);  
        reply.setContent(String.valueOf(n));  
    }  
    else {  
        reply.setPerformative(ACLMessage.FAILURE);  
        reply.setContent("unknown operator");  
    }  
    send(reply);  
}
```

Construction de la réponse

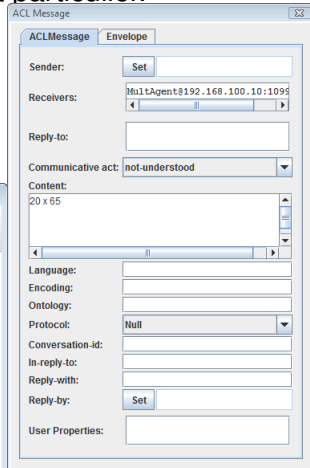
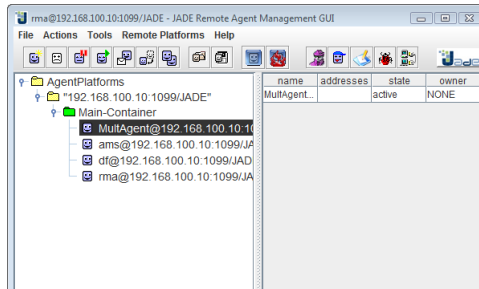
- La méthode `message.createReply()` permet de fixer automatiquement la majeure partie des éléments du message de réponse.
- le destinataire est l'expéditeur du premier message,
- le type du message et le contenu sont les seuls éléments à fournir.

Envoi d'un message

- Un agent peut envoyer un message à un ou plusieurs agents il doit fixer :
 - le ou les destinataires,
 - le type du message,
 - le contenu.

Envoi depuis le conteneur d'agents

- Il est possible à partir de la console du conteneur principal d'envoyer un message à un agent particulier.



Sommaire

- 1 Introduction
 - Conteneur d'agents
 - Outils
- 2 Structure d'un agent
 - Cycle de vie
 - Comportement d'un agent
- 3 Communication entre agents
 - Messages
 - Principaux behaviours

Behaviour générique

- La classe behaviour doit étendre `Behaviour`.
- Le constructeur contient toujours en paramètre une référence sur l'agent qui le possède.
- l'attribut protégé `myAgent` permet d'utiliser l'agent concerné dans le code du behaviour.
- Les behaviours génériques ont des états (repéré par exemple par la variable `step`) et exécutent différentes instructions en fonction de leur état. Ils se terminent lorsqu'une certaine condition est rencontrée.
- La méthode `done()` est à implémenter et retourne la condition booléenne d'arrêt.

One-shot behaviour

- La méthode `action()` est exécutée une seule fois et la tâche s'arrête
- La classe `behaviour` doit étendre `OneShotBehaviour`.
- La méthode `done()` est déjà implémentée et retourne `true`.

Cyclic behaviour

- La méthode `action()` exécute les mêmes instructions chaque fois qu'elle est appelée. La tâche n'est jamais terminée.
- La classe `behaviour` doit étendre `CyclicBehaviour`.
- La méthode `done()` est déjà implémentée et retourne `false`.
- Dans l'exemple proposé, l'agent "Mult" recevait un argument lui indiquant le nombre de sollicitations auxquelles répondre. On aurait pu étendre `CyclicBehaviour` pour simplifier le comportement de l'agent.