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

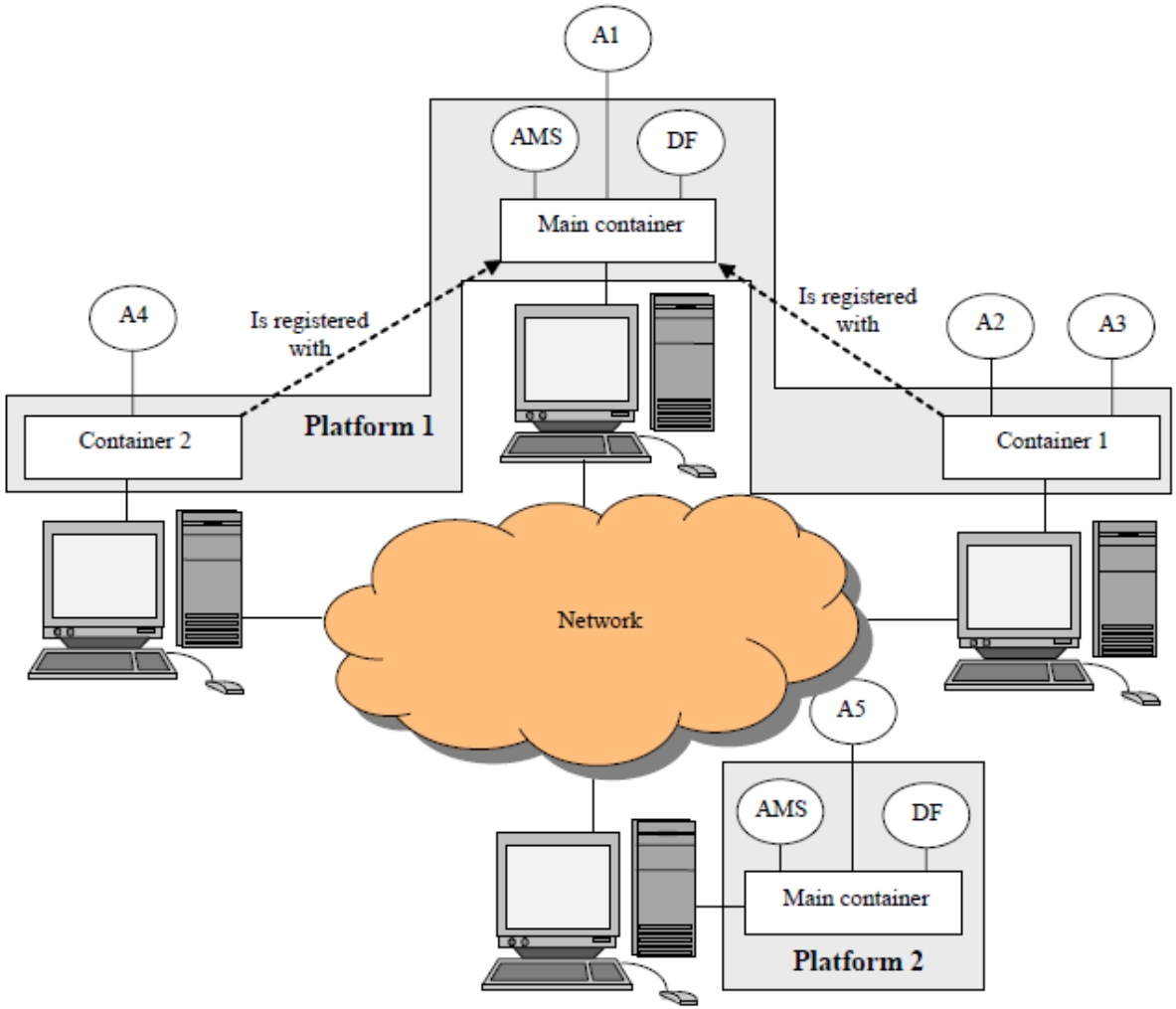
JADE

- JADE (Java Agent DEvelopment Framework) est une plate forme de développement de systèmes multi-agents implémentée en Java. `http://jade.tilab.com/`
- Elle contient une bibliothèques de classes utiles pour développer des agents.
- JADE est distribué par Telecom Italia.
- JADE est un ensemble d'outils qui permettent le déboggage et le déploiement, l'administration et le monitorage de SMA.
- Il suffit d'attacher au classpath de l'application ou du projet le fichier archive (jade4.5.0.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 conteneurs secondaires : ils s'enregistrent auprès du conteneur principal (sur la 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.

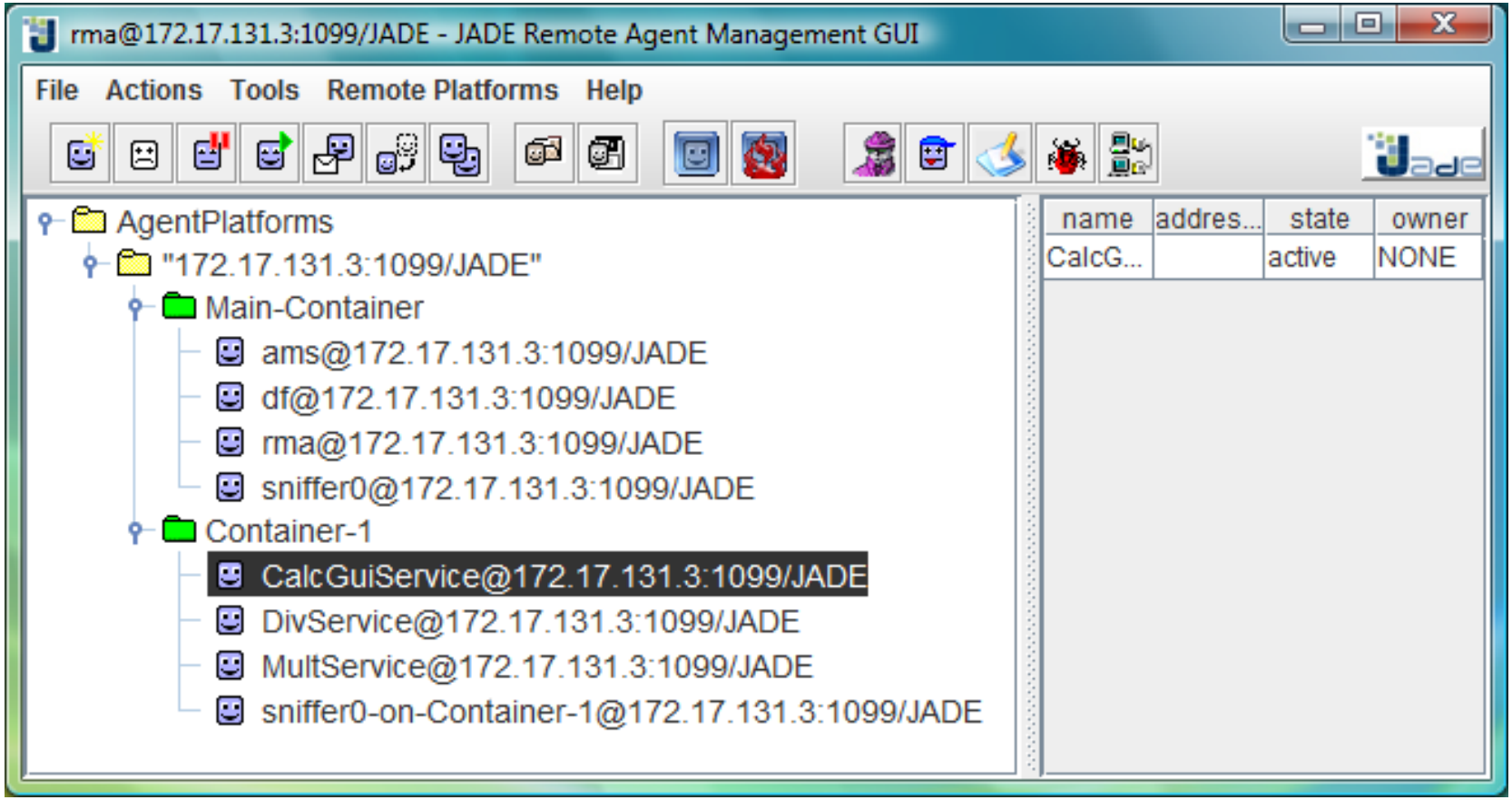
Sommaire

1 Introduction
- Conteneur d'agents
- Outils

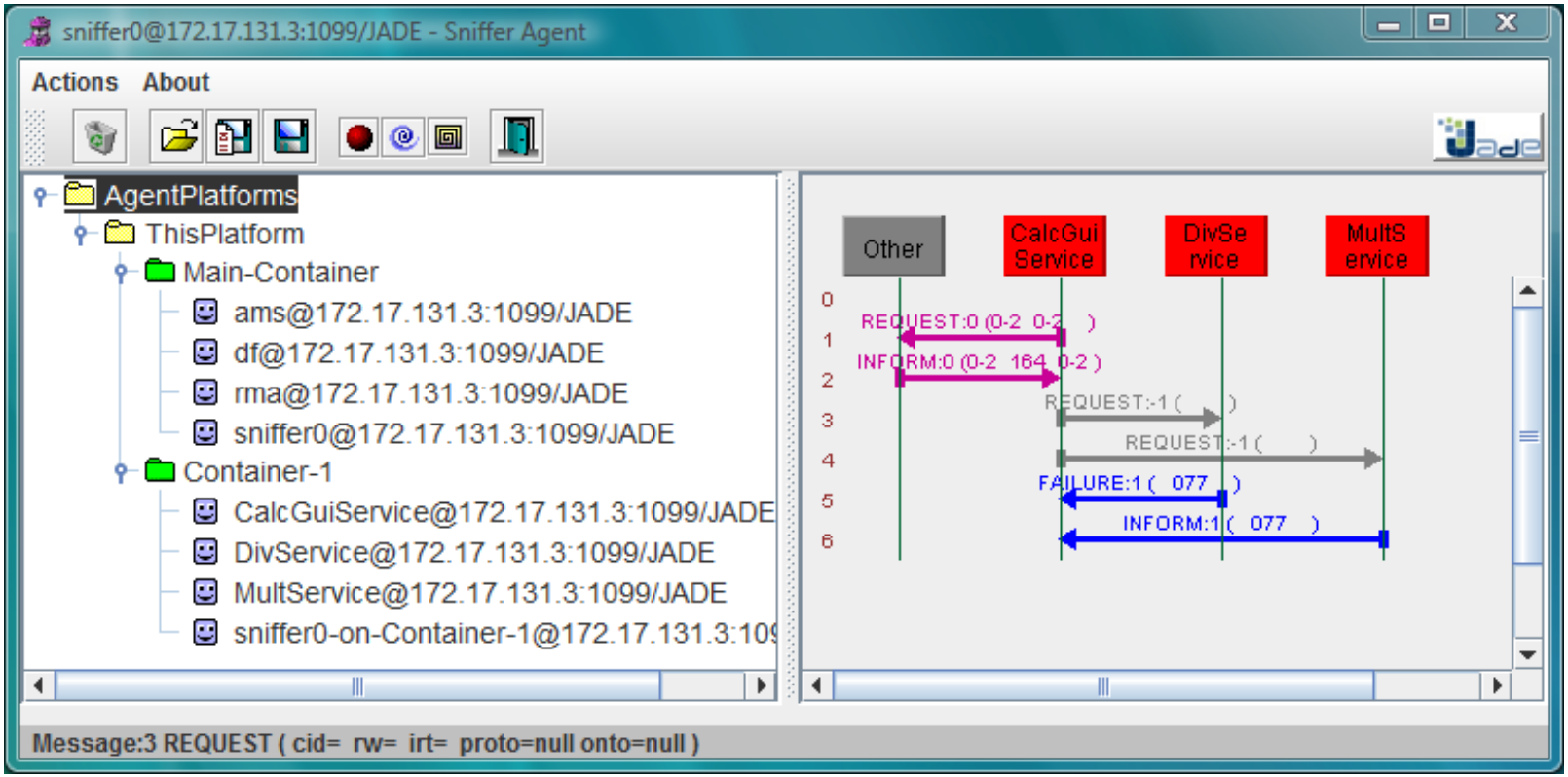
2 Structure d'un agent
- Cycle de vie
- Comportement d'un agent

3 Communication entre agents
- Messages
- Principaux behaviours

Gestion des agents



Messages


sniffer0@172.17.131.3:1099/JADE - Sniffer Agent
Actions
About
AgentPlatforms
ThisPlatform
Main-Container
ams@172.17.131.3:1099/JADE
df@172.17.131.3:1099/JADE
rma@172.17.131.3:1099/JADE
sniffer0@172.17.131.3:1099/JADE
Container-1
CalcGuiService@172.17.131.3:1099/JADE
DivService@172.17.131.3:1099/JADE
MultService@172.17.131.3:1099/JADE
sniffer0-on-Container-1@172.17.131.3:109
Other
CalcGui Service
DivSe rvice
MultS ervice
0
1
2
3
4
5
6
REQUEST:0 (0-2 0-2)
INFORM:0 (0-2 164 0-2)
REQUEST:-1 ( )
REQUEST:-1 ( )
FAILURE:1 (077)
INFORM:1 (077)
Message:3 REQUEST (cid= rw= irt= proto=null onto=null)

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
#port du conteneur principal
#local-port=1099
```

Création d'un conteneur secondaire

Un ou plusieurs conteneurs secondaires peuvent être lancés sur une même station ou sur différentes stations.

```
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 {
// Création des agents appartenant au container
  ...
}
catch(Exception ex) {...}
```

Le container est créé à partir d'un profil avec des valeurs par défaut.

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);
  // Création des agents
  ...
} catch (Exception ex) {
  ex.printStackTrace();
}
```

Container créé à partir d'un profil défini dans un fichier

Fichier de propriétés

```
main=false
gui=false
container-name = tdtp
host = 172.17...  #ip du container principal
#local-port=1099  #port de ce container
#port=1098        #port du container principal
```

Création des agents d'un conteneur

Un ou plusieurs conteneurs secondaires peuvent être lancés sur une même station ou sur différentes stations.

```
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",
    "tp1.agent.HelloAgent", null);
 ac.start();
 ...
}
catch(Exception ex) {...}
```

Création des agents d'un conteneur

```
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",
    "tp1.agent.HelloAgent", null);
 ac.start();
 ...
}
catch(Exception ex) {...}
```

Hello : nickname de l'agent
tp1.agent.HelloAgent : nom complet de la classe de l'agent
null : dans le cas où l'on ne communique pas de paramètres à l'agent, sinon tableau d'objets.

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`).
- Le nom complet d'un 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);
```

- Pour envoyer un message, il est nécessaire de créer un AID avec le nickname de l'agent destinataire.
- 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 de deux nombres. ;

```
ContainerController cc = rt.createAgentContainer(p);
ac = cc.createNewAgent("MULT",
                "tp1.agent.MultAgent", null);
ac.start();
```

- Code principal de la classe agent :

```
public class MultAgent extends Agent {
 protected void setup() {
  System.out.println(getLocalName()+ "--> Installed");
  // ajout des tâches initiales
 }
}
```

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 en tant qu'objet d'une classe qui étend la classe `Behaviour`.
- On peut écrire cette classe directement dans la classe Agent.
- Pour qu'un agent exécute une première tâche il faut ajouter le behaviour correspondant, dans le setup() de l'agent :

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

- Un agent peut avoir plusieurs (tâches) behaviours en parallèle.
- Un behaviour peut installer d'autre behaviours, peut s'arrêter.

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 ainsi doit être retiré du pool des behaviours d'un agent.

Exécution d'un agent

- Chaque agent JADE est éxécuté 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 éxécuté 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 behaviour

- Il faut prévoir des actions courtes qui puissent s'arrêter (envoi d'un message) ou qui puissent s'arrêter rapidement et reprendre (vérification de l'arrivée d'un message).
- Dans certains cas on peut structurer un behaviour en plusieurs états.
  - A chaque prise en main, le code d'un état est exécuté et rend la main. Il peut indiquer une transition vers un nouvel état.
  - 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

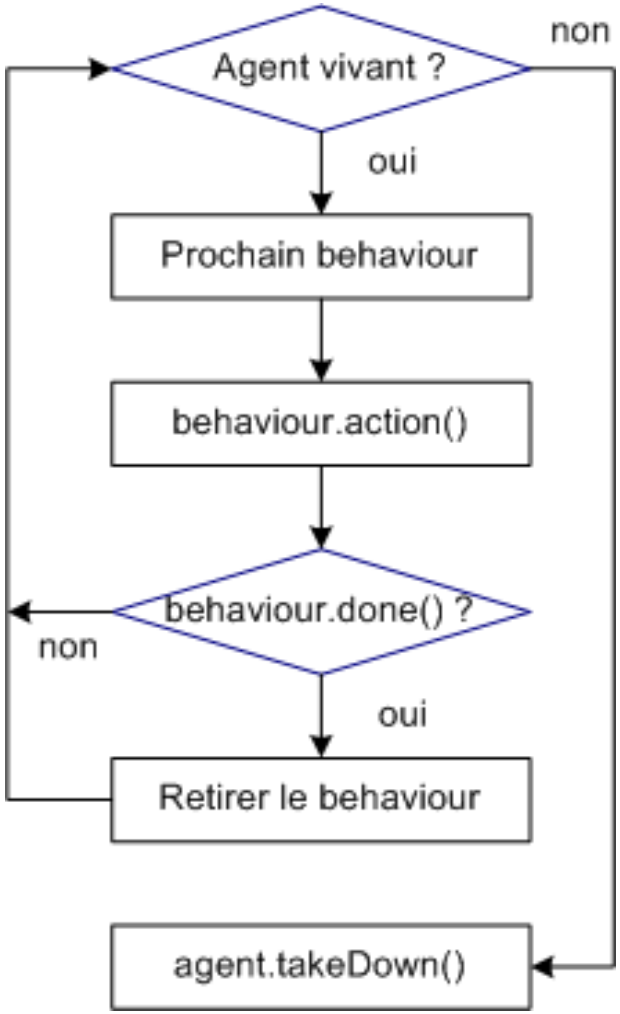
- Un seul thread Java par agent.
- Elimination des problèmes de synchronisation lors d'accès aux mêmes ressources.
- Meilleures performances car changer de behaviour est plus rapide que changer de thread Java.
- La persistence des agents et leur mobilité sont mieux assurées.

Scheduling

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

```
b1.action() ; // step = 0 ; ... ; step : 0 -> 1
b2.action() ; // teste si un message est arrivé
b3.action() ;
b1.action() ; // step = 1
b2.action() ; // teste si un message est arrivé
              // traite le message
b3.action() ;
b1.action() ;
...
```

Scheduling


Agent vivant ?
non
oui
Prochain behaviour
behaviour.action()
behaviour.done() ?
non
oui
Retirer le behaviour
agent.takeDown()

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

Communication par message

- Les messages reçus par un agent sont placés sous forme de file dans sa boite à message.
- Une des tâches d'un agent (un behaviour) est de regarder dans cette boite si un message est présent.
- Dans ce cas, l'agent peut récupérer le premier message disponible (et ainsi le retirer de sa boîte à messages) et de traiter l'information qu'il contient.
  - 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 conseillé de bloquer le behaviour.
- L'arrivée d'un nouveau message le débloque.

Attente d'un message

- Un agent peut exécuter plusieurs tâches et une ou plusieurs sont amenées à attendre des messages pour entreprendre une action.
- Sauf information complémentaire, le prochain behaviour actif récupère le premier message disponible.
- En général, on spécifie le type de message qu'un behaviour peut récupérer.

Un behaviour de l'agent regarde si un message est présent

```
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.

Un behaviour envoie un message et se met en attente d'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;
 }
```

Eléments 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);
```

On préférera un contenu de message sous forme JSON.

Agent multiplicateur

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

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

Réponse à un contenu : 10 x 25

```
private void answer(ACLMessage message) {
 String produit = message.getContent();
 ACLMessage reply = message.createReply();
 String[] parameters = produit.split("x");
 if (parameters.length == 2) {
  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("Bad message: n1 x n2");
 }
 send(reply);
}
```

Contenu : objet formaté JSON

- L'agent émetteur sérialise un objet en JSON et met la chaîne comme contenu du message.
- L'agent destinataire désérialise la chaîne en un objet, traite le message en utilisant l'objet et retourne la réponse en tant qu'objet sérialisé.

Classe encapsulant les données

```
public class Product {
  private String action;
  private Double[] args;
  public Product(String action, Double[] args) {
   this.action = action;
   this.args = args;
  }
  public String toJSON() {...}
  public static Product read(String jString) {...}
  ...
}
```

Désérialisation des données

```
// réception du message
...
String par = message.getContent();
ACLMessage reply = message.createReply();
// Désérialisation JSON
Product p = Product.read(par);
// Préparation de la réponse
...
// Envoi de la réponse
send(reply);
```

Construction de la réponse

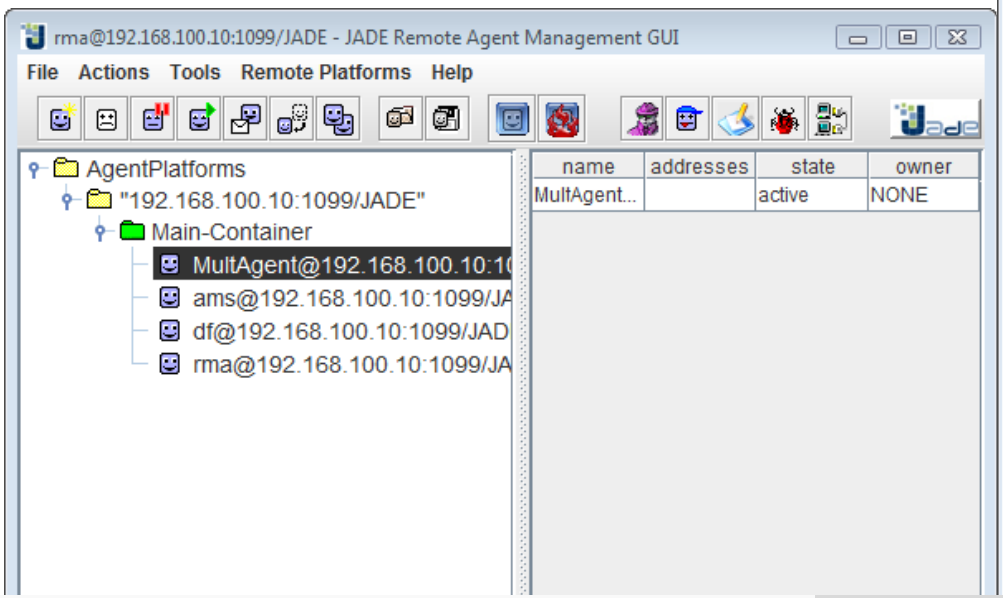
- La méthode `message.createReply()` permet de fixer automatiquement la majeure partie des éléments du message de réponse.
- Le destinataire est l'expéditeur du premier message.
- Le type du message et le contenu sont les seuls éléments à fournir.

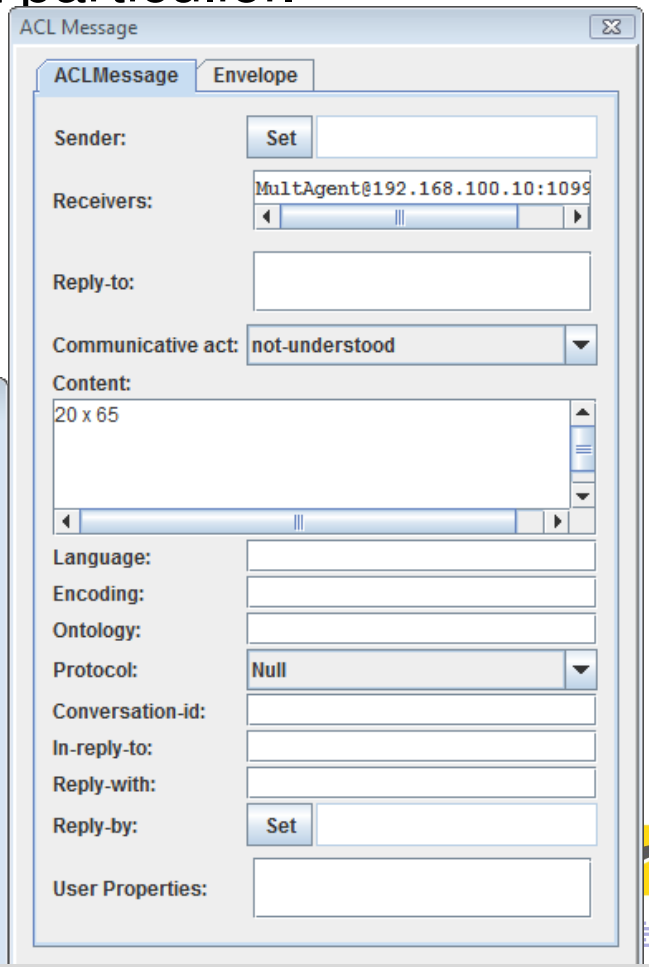
Envoi d'un message

- 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 peuvent être des machine à états (repéré par exemple pas 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.