

Plateforme JADE

Compléments

Claude Moulin

Université de Technologie de Compiègne

IA04

Sommaire

- 1 Pattern de message
 - Actes de langages
 - Patterns
 - Discussion entre agents
- 2 Description d'un agent
 - Agents Plateforme
 - Enregistrement - recherche
- 3 Transmissions d'objets
 - Sérialisation JSON
- 4 Behaviours particuliers

Sommaire

- 1 Pattern de message
- 2 Description d'un agent
- 3 Transmissions d'objets
- 4 Behaviours particuliers

- 1 Pattern de message
Actes de langages
Patterns
Discussion entre agents
- 2 Description d'un agent
Agents Plateforme
Enregistrement - recherche
- 3 Transmissions d'objets
Sérialisation JSON
- 4 Behaviours particuliers

Performatifs

- Le performatif d'un message en indique le type.
- `ACLMessage.INFORM` : l'expéditeur informe le destinataire qu'une proposition est vraie.
- `ACLMessage.REQUEST` : l'expéditeur demande au destinataire d'exécuter une action.
- `ACLMessage.FAILURE` : l'expéditeur informe le destinataire qu'il y a eu un problème lors de l'exécution d'une tâche.

Sommaire

- 1 Pattern de message**
 - Actes de langages
 - Patterns**
 - Discussion entre agents
- 2 Description d'un agent
 - Agents Plateforme
 - Enregistrement - recherche
- 3 Transmissions d'objets
 - Sérialisation JSON
- 4 Behaviours particuliers

Message JADE

- Classe : `ACLMessage`
- Expéditeur (sender)
- Liste des destinataires : méthode `addReceiver()`
 - Question : comment connaître les destinataires ?
- Le performatif : `ACLMessage.REQUEST`,
`ACLMessage.INFORM`, `ACLMessage.FAILURE`, etc.
- Le contenu (chaîne) : méthodes `setContent()` et
`getContent()`
 - Comment passer un objet dans un message ?

Pattern - 1

```
ACLMessage message =  
    new ACLMessage (ACLMessage.REQUEST) ;
```

- La création d'un message définit son type
- Deux behaviour représentant deux tâches différentes ne doivent être concernés que par un seul type de message et ignorer les autres.

```
MessageTemplate mt =  
    MessageTemplate.  
        MatchPerformative (ACLMessage.REQUEST) ;  
ACLMessage message = receive (mt) ;
```


Champs conversationnels - 1

- **conversation-id** : méthodes `setConversationId()` et `getConversationId()`
Il introduit un identifiant de conversation utilisé pour identifier les actes qui participent de la même conversation.
- Le conversation-id peut servir à définir un template utilisé plus tard dans la réception de messages :

```
MessageTemplate mt =  
    MessageTemplate.MatchConversationId(<id>);
```

Champs conversationnels - 2

- **reply-with** : méthodes `setReplyWith()` et `getReplyWith()`
 - introduit une expression qui sera utilisée par le destinataire pour identifier le message.
 - est utile lorsque plusieurs conversations sont utilisées en parallèle avec le même identificateur de conversation.
- **in-reply-to** : méthodes `setInReplyTo()` et `getInReplyTo()`
désigne une expression qui référence une précédente action.

Pattern - 2

- Le pattern est construit à partir des deux champs conversation-id et reply-with.

```
MessageTemplate mt = MessageTemplate.and(  
    MessageTemplate.MatchConversationId(<id>),  
    MessageTemplate.MatchInReplyTo(<mirt>);
```

```
ACLMessage reply = myAgent.receive(mt);
```

Sommaire

- 1 **Pattern de message**
Actes de langages
Patterns
Discussion entre agents
- 2 Description d'un agent
Agents Plateforme
Enregistrement - recherche
- 3 Transmissions d'objets
Sérialisation JSON
- 4 Behaviours particuliers

Agent Initiateur (1)

- L'agent 1 envoie un message à l'agent 2

```
ACLMessage m = new ACLMessage (ACLMessage.REQUEST) ;  
m.addReceiver (<agent 2>) ;  
m.setConversationId (<id>) ;  
String mirt = "rqt"+System.currentTimeMillis () ;  
m.setReplyWith (mirt) ;  
myAgent.send (m) ;
```

- L'agent 1 prépare le template lui servant à filtrer la réponse de l'agent 2

```
MessageTemplate mt = MessageTemplate.and(  
    MessageTemplate.MatchConversationId (<id>),  
    MessageTemplate.MatchInReplyTo (mirt) ;
```

Agent destinataire (2)

- L'agent 2 filtre les messages REQUEST et répond à l'agent 1

```
MessageTemplate mt = MessageTemplate.  
    MatchPerformative (ACLMessage.REQUEST) ;  
ACLMessage message = myAgent.receive(mt) ;  
if (message != null) {  
    ACLMessage reply = message.createReply() ;  
    ...  
    myAgent.send(reply) ;  
}  
else block() ;
```

- La méthode `createReply()` met à jour dans la réponse le conversation Id, le champ in-reply-to à partir du champ in-reply-with du message reçu.

Agent Initiateur (1)

- L'agent 1 reçoit un message réponse de l'agent 2
- Il filtre les messages à partir des deux champs conversation-id et in-reply-with grâce au template créé au moment de l'envoi du premier message et analyse la réponse.

```
// Template initial  
MessageTemplate mt = MessageTemplate.and(  
    MessageTemplate.MatchConversationId(<id>),  
    MessageTemplate.MatchInReplyTo(<mirt>);
```

- `ACLMessage reply = receive(mt);` :
filtrage à la réception

Sommaire

- 1 Pattern de message
- 2 Description d'un agent**
- 3 Transmissions d'objets
- 4 Behaviours particuliers

Sommaire

- 1 Pattern de message
 - Actes de langages
 - Patterns
 - Discussion entre agents
- 2 Description d'un agent
 - Agents Plateforme**
 - Enregistrement - recherche
- 3 Transmissions d'objets
 - Sérialisation JSON
- 4 Behaviours particuliers

Agents système : AMS

- L'agent AMS (Agent Management System) est l'agent qui supervise l'accès et l'utilisation d'une plate-forme.
- Une plateforme n'a qu'un seul AMS.
- L'AMS fournit les services de pages blanches et de cycle de vie des agents.
- Il maintient un répertoire d'identifiants d'agents (AID) et de leurs états.
- Chaque agent doit s'enregistrer auprès d'un AMS pour obtenir un AID valide.

Agents système : DF et ACC

- L'agent DF (Directory Facilitator) :
 - assure le service de pages jaunes
 - permet à un agent de trouver d'autres agents avec lesquels communiquer.
- L'agent Message Transport System, ou Agent Communication Channel (ACC) contrôle tous les échanges de messages dans une plateforme.

Sommaire

- 1 Pattern de message
 - Actes de langages
 - Patterns
 - Discussion entre agents
- 2 Description d'un agent
 - Agents Plateforme
 - Enregistrement - recherche
- 3 Transmissions d'objets
 - Sérialisation JSON
- 4 Behaviours particuliers

Enregistrement d'un agent

- Un agent peut publier dans les pages jaunes (Agent DS) une description d'agent (le nom est son AID) contenant une ou plusieurs descriptions de services qu'il est capable de remplir.
 - Chaque description de service contient le nom du service et le type du service.
 - On peut y ajouter les langages et les ontologies requis pour exploiter le service ainsi que des propriétés spécifiques du service.
- Il est nécessaire que les dénominations des types et des noms et des propriétés soient connus par l'ensemble des agents.

Enregistrement : exemple

```
DFAgentDescription dfad = new DFAgentDescription();  
dafd.setName(getAID());  
ServiceDescription sd = new ServiceDescription();  
sd.setType("Operations");  
sd.setName("Multiplication");  
dafd.addServices(sd);  
try {  
    DFService.register(this, dafd);  
}  
catch (FIPAException fe) {  
    fe.printStackTrace();  
}
```

Recherche d'agent

- Pour envoyer un message à un destinataire, il est nécessaire de connaître son nom. Coder en dur le nom des agents est cependant fortement déconseillé.
- Il est plus intéressant de rechercher un agent capable de remplir un service donné.
- L'agent DF permet de faire des recherches dans la liste des services enregistrés (par nom, type, propriété).
- Pb : comment choisir l'agent qui convient ou comment traiter les meilleures réponses si un message est envoyé à tous les agents remplissant un service donné.

Sélection : exemple

```
private AID getReceiver() {  
    AID rec = null;  
    DFAgentDescription template =  
        new DFAgentDescription();  
    ServiceDescription sd = new ServiceDescription();  
    sd.setType("Operations");  
    sd.setName("Multiplication");  
    template.addServices(sd);  
    try {  
        DFAgentDescription[] result =  
            DFService.search(this, template);  
        if (result.length > 0)  
            rec = result[0].getName();  
    } catch (FIPAException fe) {...}  
    return rec;  
}
```


Envoi du message

```
public void action() {  
    ACLMessage message =  
        new ACLMessage (ACLMessage.REQUEST) ;  
    receiver = getReceiver() ;  
    if (receiver != null) {  
        message.addReceiver(receiver) ;  
        message.setContent (...) ;  
        send(message) ;  
        step++ ;  
    }  
    else  
        System.out.println(  
            getLocalName() + "--> No receiver") ;  
}
```

Sommaire

- 1 Pattern de message
- 2 Description d'un agent
- 3 Transmissions d'objets**
- 4 Behaviours particuliers

méthode setContentObject()

- Cette méthode fixe un objet comme contenu d'un message (ACLMessage).
- Cette méthode n'est pas compatible avec la norme FIPA et est découragée.
- Il est préférable d'utiliser une sérialisation sous forme de chaîne de l'objet à communiquer.
- Sérialisation XML (Java JAXB)
- Sérialisation JSON : Parser, générateur
- `http://mvnrepository.com/artifact/com.fasterxml.jackson.core`

Classe Product

- Exemple : demande d'effectuer une opération.

```
public class Product {  
    private String action;  
    private Double[] args;  
    // Constructeurs  
    ...  
    // Accesseurs  
}
```

Classe OperationResult

- Exemple : réponse à une demande d'effectuer une opération.

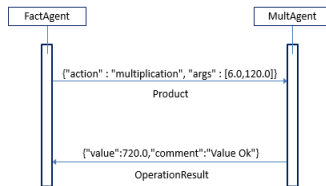
```
public class OperationResult {  
    double value;  
    String comment;  
    // Constructeurs  
    ...  
    // Accesseurs  
}
```

Processus - 1

- L'agent "FactAgent" demande à l'agent "MultAgent" d'effectuer une opération
- Au travers de l'un de ses behaviours ; il lui envoie une chaîne telle que :

```
{"action" : "multiplication", "args" : [6.0,120.0]}
```

 comme contenu d'un message, sérialisation d'un objet Product.



Processus - 2

- L'agent "MultAgent" répond dans son comportement adapté en sérialisant un objet `OperationResult` et en l'envoyant en retour à l'agent "FactAgent" comme contenu d'un message.
 - `{"value":720.0,"comment":"Value Ok"}`
 - `{"value":-2147483648.0,"comment":"unknown operator"}`

Processus - 3

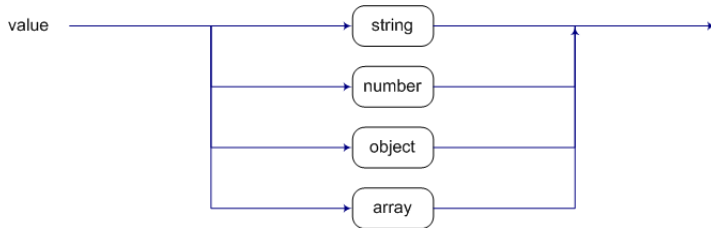
- L'agent "FactAgent" analyse la réponse dans le comportement adapté en désérialisant l'objet contenu sous forme de chaîne dans le message.
- Inconvénient :
 - la classe de l'objet passé en message doit être connue à la fois sur la station de l'agent émetteur et sur celle de l'agent destination.
 - une classe de l'agent destination est capable de désérialiser la chaîne JSON du contenu.

Sommaire

- 1 Pattern de message
 - Actes de langages
 - Patterns
 - Discussion entre agents
- 2 Description d'un agent
 - Agents Plateforme
 - Enregistrement - recherche
- 3 Transmissions d'objets**
 - Sérialisation JSON**
- 4 Behaviours particuliers

The diagram illustrates the structure of an object and an array in JSON. The top part shows an object structure: an input labeled 'object' points to a box containing '{'. From this box, a line goes to a box containing 'string', then to a box containing ':'. From the ':' box, a line goes to a box containing 'value'. From the 'value' box, a line goes to a box containing '}'. A feedback loop connects the '}' box back to the '{' box. A second line from the '{' box goes to a box containing ',', and from the ',' box, a line goes back to the 'string' box. The bottom part shows an array structure: an input labeled 'array' points to a box containing '['. From this box, a line goes to a box containing 'value'. From the 'value' box, a line goes to a box containing ']'. A feedback loop connects the ']' box back to the '[' box. A second line from the '[' box goes to a box containing ',', and from the ',' box, a line goes back to the 'value' box.

Grammaire JSON - 2



Format de données JSON

- Un objet :
 - Ensemble de couples <nom/valeur> non ordonnés, placés entre { et }.
 - Un couple <nom/valeur> est de la forme : "<nom>" : <valeur>.
 - Les couples <nom/valeur> sont séparés par , .
- Un tableau est :
 - une collection de valeurs ordonnées mises entre [et] .
 - les valeurs sont séparées par , .
- Une valeur peut être :
 - une chaîne de caractères entre guillemets,
 - un nombre
 - true ou false ou null
 - un objet, un tableau

Mapping objet JSON

- Le nom d'un couple "<nom>" : <valeur> est celui d'un champ (une propriété) de la classe.
- Le champ doit être public ou avoir au moins un accesseur (get ou set).
- La classe doit avoir un constructeur par défaut.
- Il est préférable d'utiliser une structure dynamique pour les propriétés composites (ArrayList).

Exemples

Tableau d'objets :

```
[{"value": "un"}, {"value": "deux"}, {"value": "trois"}]
```

Objet composite :

```
{  
  "value" : 200,  
  "comment" : {"w1" : "Value",  
               "w2" : "Ok"},  
  "list" : [10,  
            {"language" : "fr"},  
            20]  
}
```

Code

- Utiliser une bibliothèque de classes pour simplifier le mapping entre le texte et l'objet en mémoire.
- Etudier l'API de cette bibliothèque.
- Exemple : Jackson (codehaus)
- `http://mvnrepository.com/artifact/com.fasterxml.jackson.core`
- Exemple : `javax.json-1.0.4.jar`.

Sérialisation JSON

```
ObjectMapper mapper = new ObjectMapper();  
String s = "";  
OperationResult or =  
    new OperationResult(720.0, "Value Ok");  
try {  
    s = mapper.writeValueAsString(or);  
    // Utiliser s  
}  
catch(Exception ex) {...}
```


Sérialisation JSON dans une classe

```
public String toJSON() {  
    ObjectMapper mapper = new ObjectMapper();  
    String s = "";  
    try {  
        s = mapper.writeValueAsString(this);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return s;  
}
```

Désérialisation JSON

```
public static FactValue read(String jsonString) {  
    ObjectMapper mapper = new ObjectMapper();  
    FactValue fv = null;  
    try {  
        fv = mapper.readValue(jsonString, FactValue.class);  
    }  
    catch(Exception ex) {...}  
    return fv;  
}
```

Désérialisation JSON sous forme de Map

```
String s = ... // chaîne JSON
Map<String, Object> map;
ObjectMapper mapper = new ObjectMapper();
try {
    map = mapper.readValue(s, Map.class);
    System.out.
        println("action: " + map.get("action"));
}
catch(Exception ex) {...}
```

Sommaire

- 1 Pattern de message
- 2 Description d'un agent
- 3 Transmissions d'objets
- 4 Behaviours particuliers**

Behaviours simples

- Classe `OneShotBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui ne sont exécutés qu'une seule fois et qui ne peuvent pas être bloqués. La méthode `done()` retourne toujours `true`.
- Classe `CyclicBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui s'exécutent indéfiniment. La méthode `done()` retourne toujours `false`.

Autres simples behaviours

- Classe `WakerBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui ne sont exécutés qu'une seule fois mais après un certain délai.
- Classe `TickerBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui s'exécutent indéfiniment mais à intervalle régulier.

Behaviours composites

- Classe `CompositeBehaviour`. Cette classe abstraite est la classe de base des behaviours composites.
- Classe `SequentialBehaviour`. Cette classe exécute ses sous-behaviours de façon séquentielle et se termine lorsque le dernier est exécuté. On peut utiliser cette classe lorsqu'une tâche complexe se décompose en une suite de tâches.
- Classe `ParallelBehaviour`. Cette classe exécute ses sous-behaviours de façon concurrente et se termine lorsque soit :
 - tous ses sous-behaviours sont terminés,
 - l'un quelconque de ses sous-behaviours est terminé,
 - n de ses sous-behaviours sont terminés.

Machine à états

- Classe `FSMBehaviour`
- Cette classe exécute ses sous-behaviours selon une machine à états finis définie par l'utilisateur.
- Chaque sous-behaviour représente l'activity exécutée dans un état de l'automate.
- Lorsque le sous-behaviour correspond à un état se termine, sa valeur de terminaison (retournée par la méthode `onEnd()` est utilisée pour déterminer la transition et atteindre le prochain état exécuté au prochain tour.
- Certains sous-behaviours sont marqués comme états finals.
- Le `FSMBehaviour` se termine après terminaison de l'un des sous-behaviours état final.