MASTER WEB INTELLIGENCE

Systèmes Multi-Agents

JADE
Environnement pour la programmation multi-agent

Olivier Boissier
Olivier.Boissier@emse.fr

05 Janvier 2010

Plan

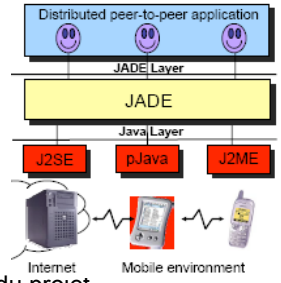
Principes et architecture de JADE

I. Vue d'ensemble
II. Caractéristiques
III. Plateforme JADE
IV. Architecture

Utilisation de la plateforme JADE
Programmation avec JADE

JADE
(Java Agent DEvelopment Framework)

- Intergiciel pour le développement d'applications pair à pair d'agents intelligents
  - Sur des plateformes fixes, téléphones mobiles, …
- Deux produits principaux :
  - Plateforme agent satisfaisant aux spécifications de la FIPA
  - API pour développer des agents en Java
- Projet Open Source, LGPL License
- Contrôlée par Telecom Italia Lab, qui reste propriétaire du projet
- Résultat des efforts conjoints de différents acteurs réunis au sein du JADE Board (fondé en 2003) dont les missions sont la promotion, la gouvernance et l'implémentation des évolutions de JADE



- Portail du projet : http://jade.tilab.com

Caractéristiques

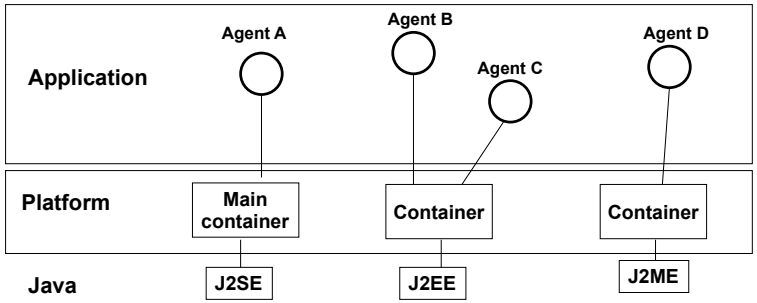
- Support à l'envoi de messages, transparent et multi-protocoles
  - Diffusion d'événements en local
  - Java RMI pour la diffusion interne à une plateforme
  - FIPA 2000 Message Transport Protocol
    - Protocole IIOP pour la diffusion inter-plateforme
    - Protocole HTTP et encodage des ACL en XML
- Modèle de concurrence à deux niveaux
  - Entre agents (pre-emptif, Threads Java)
  - Interne aux agents (co-opératif, classes de comportements "behaviour")
- Mobilité des agents
  - entre plateforme, mobilité faible
- Framework orienté objet implémentant en Java les spécifications FIPA
  - Plateforme agent
  - Langage de communication agent
  - Ontologie de management des agents
  - Protocoles d'interaction standards
  - Langages définis par les utilisateurs et les ontologies

Caractéristiques

- JADE encapsule les spécifications de la FIPA :
  - La plateforme fournit : AMS, DF, MTS et ACC
  - agent-management-ontology, contructeur d'un agent l'enregistre dans la plateforme (nom, adresse), classe DFService permet un accès au DF
  - Transport et traitement des messages
  - Extension des protocoles standards d'interaction par les méthodes handle
- Système de gestion d'événements dans le noyau de la plateforme
  - Permet l'observation de la plateforme, des messages, du transport des messages, des agents
- Outils de gestion basés sur des agents
  - Agents spéciaux (RMA, Sniffer, Introspector) qui communiquent avec FIPA ACL
  - Extensions à l'ontologie fipa-management-ontology pour y inclure des actions spécifiques
  - Ontologie particulière pour l'observation jade-introspection
- Agents utilitaires
  - *DummyAgent tool* permet à des utilisateurs d'interagir avec les agents déployés sur la plateforme
  - *Sniffer Agent* agent utilisé pour observer les messages

Plateforme JADE

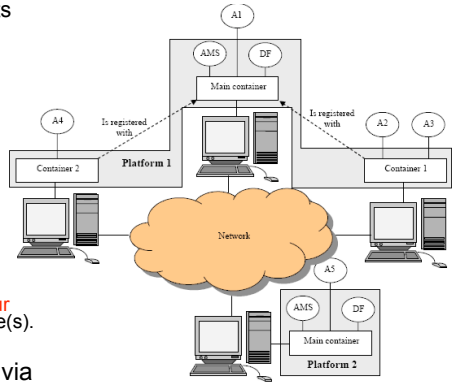
- Une *application* JADE est une *plateforme* déployée sur une ou plusieurs machines
- La plateforme héberge un ensemble d'agents, identifiés de manière unique, pouvant communiquer de manière bidirectionnelle avec les autres agents
- Chaque agent s'exécute dans un *conteneur* (container) qui lui fournit son environnement d'exécution ; il peut migrer à l'intérieur de la plateforme
- Toute plateforme doit avoir un *conteneur principal* qui enregistre les autres conteneurs
- Une plateforme est un ensemble de conteneurs actifs



Conteneurs

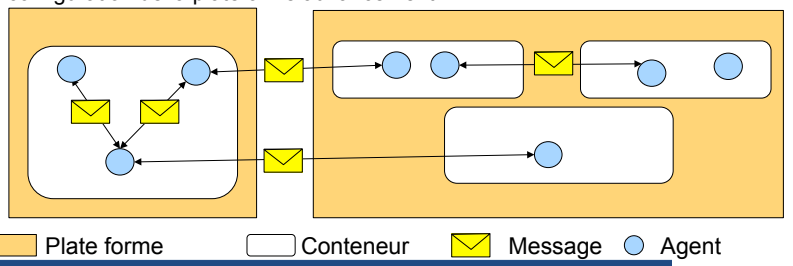
- Un **conteneur** est un environnement d'exécution JADE :
  - Environnement complet d'exécution pour un agent
  - Exécution concurrente de plusieurs agents
  - Contrôle le cycle de vie des agents
  - Assure la communication entre agents
- Un seul conteneur héberge l'AMS, le DF, c'est le conteneur principal (main container)
  - AMS (Agent Management System)
    - Service de Pages Blanches : référence automatiquement les agents suivant leur nom dès leur entrée dans le système.
  - DF (Directory Facilitator)
    - Service de Pages Jaunes : référence à leur demande les agents suivant leur(s) service(s).
- Le conteneur principal peut être répliqué via des services de réplication

AMS : Agent Management System
DF : Directory Facilitator



Agent et Communication

- 1 agent = 1 thread implémenté en JAVA selon API JADE
  - Exécute un ensemble d'actions. Les actions sont regroupées en comportements (*behaviour*)
  - Différents types de comportements : parallèle, composite, cyclique
- Interaction entre agents par envoi de messages dont le contenu est exprimé en ACL selon différents langages de contenu via différents mécanismes :
  - Au sein d'une plateforme les communications se font par RMI
  - Entre plateformes les communications se font par HTTP, IIOP, JMS, … selon la configuration de la plateforme au lancement



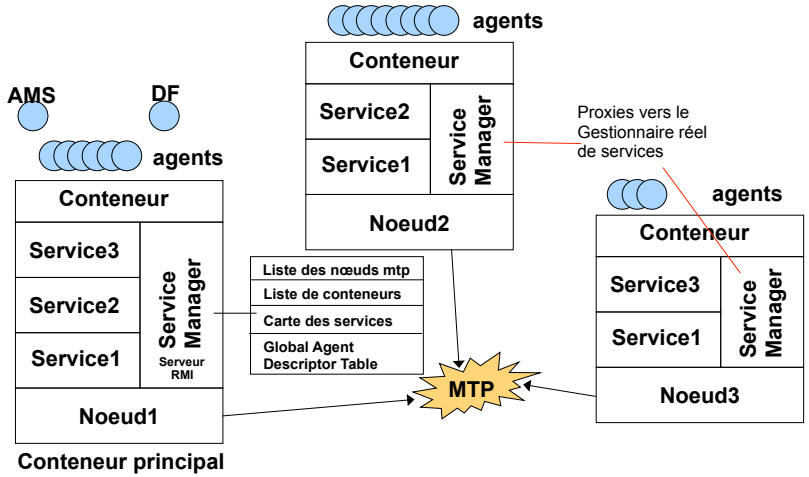
Outils utiles au débogage

- Dummy Agent
  - Visualisation des messages
  - Envoie des messages aux agents présents sur la plateforme et réceptionne leur réponse,

- Sniffer Agent
  - Visualisation de l'enchaînement des messages … et des messages eux même.
  - Vérification interactive de la correction des protocoles.

- Introspector Agent
  - Visualisation des messages envoyés/reçus,
  - Visualisation des Behaviours actifs et non-actifs,
  - Contrôle de l'état de l'agent.

Architecture de la plateforme Jade



Services Automatiques

- **Messaging** : échange des messages en ACL et gestion du Message Transfert Protocol
  - `jade.core.messaging.MessagingService`
- **Agent-Management** : gestion du cycle de vie de l'agent, support au RMA, arrêt plateforme et conteneur
  - `jade.core.management.AgentManagementService`
- **Agent-Mobility** : support à la mobilité des agents
  - `jade.core.mobility.AgentMobilityService`
- **Notification** : notifications d'événements du niveau plateforme (requis pour le Sniffer et l'Introspector)
  - `jade.core.event.NotificationService`
- Remarques : par défaut ces services sont activés, si d'autres services sont activés, alors Agent-Mobility et Notification doivent être activés explicitement.

Services optionnels

- **Persistent-Delivery** : stockage persistent des messages ACL non transmis
  - `jade.core.messaging.PersistentDeliveryService`
- **Main-Replication** : répliquer le conteneur principal pour des soucis de tolérance aux pannes. Ce service doit être activé sur chacun des hôtes hébergeant un conteneur principal
  - `jade.core.replication.MainReplicationService`
- **Address-Notification** : service d'écoute des notifications d'activation ou de suppression de copies de conteneurs principaux
  - `jade.core.replication.AddressNotificationService`
- **UDPNodeMonitoring** : service support au monitoring de plateforme par UDP
  - `jade.core.nodeMonitoring.UDPNodeMonitoringService`

Agents donnant accès aux services

- AMS: Agent Management System
  - Gestion des pages blanches
  - Gestion du cycle de vie des agents sur la plateforme
- DF: Directory Facilitator
  - Gestion des pages jaunes
  - Association entre description de services proposés et agents
  - Gestion des descriptions de services proposés par les agents
- Outils de débugage
  - Introspector, Sniffer, Dummy Agent

Plan

Principes et architecture de JADE

Utilisation de la plateforme JADE

I. Installation et lancement de la plateforme JADE

II. Outils de la plateforme JADE

III. Aspects avancés de lancement de la plateforme

Programmation avec JADE

Plan

I. Installation et lancement de la plateforme JADE

I.1. Installation et configuration

I.2. Lancement de la Plateforme

I.3. Lancement d'agents

II. Outils de la plateforme JADE

III. Aspects avancés du lancement de la plateforme

Installation et configuration

- Installation
  - Sources disponibles à http://jade.tilab.com
- Configuration
  - Ajouter dans le classpath les archives java (.jar) se trouvant dans le répertoire lib
  - Archives nécessaires
    - jade.jar
    - jadeTools.jar
  - Archives optionnelles en fonction du type de communication inter-plateformes
    - http.jar
    - iiop.jar

Lancement de la plateforme

- Lancement de base :
  - `java jade.Boot [liste agents]`
- Lancement avec interface graphique
  - `java jade.Boot -gui [liste agents]`
    - Note : l'interface graphique est représentée par le Remote Monitoring Agent (RMA)

Lancement d'agents

- Lancement d'un agent :
  - `java jade.Boot myAgt:myPackage.myAgent`
    - Note : il est nécessaire de préfixer le nom de la classe de l'agent par un nom

Plan

Exemple

- Illustration du fonctionnement de la plate forme et des différents outils qui l'accompagnent par un exemple simplifié à l'extrême,
- Contexte :
  - achat et vente de livres par des agents Vendeurs et agents Acheteurs
    - Les agents vendeurs ont une liste de livres à leur disposition ainsi que leur prix
    - Les agents acheteurs ont pour objectif de trouver un livre et de l'acheter
  - Procédure d'achat :
    - Pour acheter un livre, l'agent acheteur doit envoyer un message au DF pour lui demander la liste des vendeurs présents sur la plate forme
    - Une fois la liste obtenue, il envoie une demande de proposition aux vendeurs (cfp)
    - L'agent vendeur qui possède le livre répond par une proposition (propose) en indiquant le prix du livre
    - L'agent acheteur répond par une acceptation (accept-proposal)
    - L'agent vendeur repond par une notification (inform) vide.

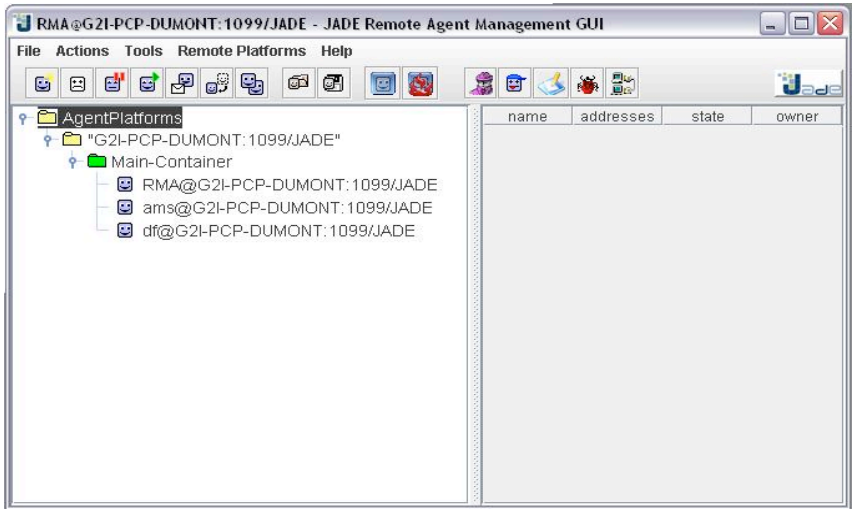
Outils : Remote Monitoring Agent

- Lancement :
  - par l'option `-gui` en ligne de commande

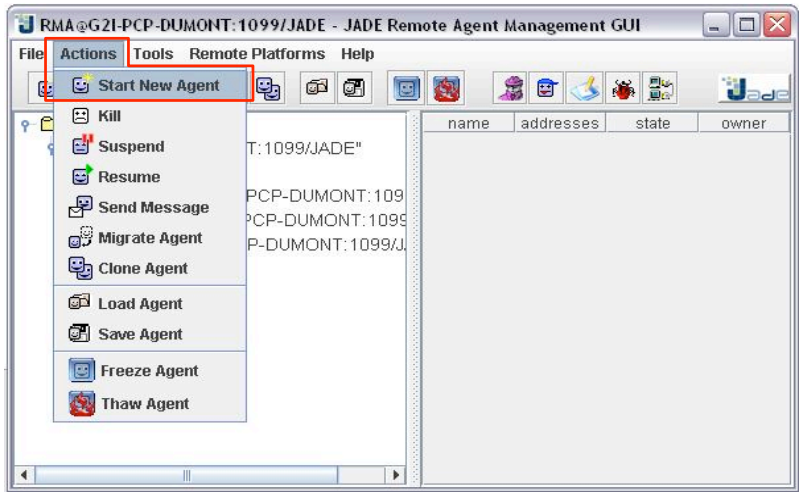
ou

  - `java jade.Boot monInterface:jade.tools.rma.rma`
- Visualisation et gestion (ajout, supression ..) de :
  - l'ensemble des conteneurs déployés au sein d'une plateforme JADE
  - des agents présents au sein de la plateforme (inscrits ou non dans le DF), par accès à l'AMS
- Possibilité d'avoir plusieurs RMA au sein d'une plateforme mais un seul par conteneur.

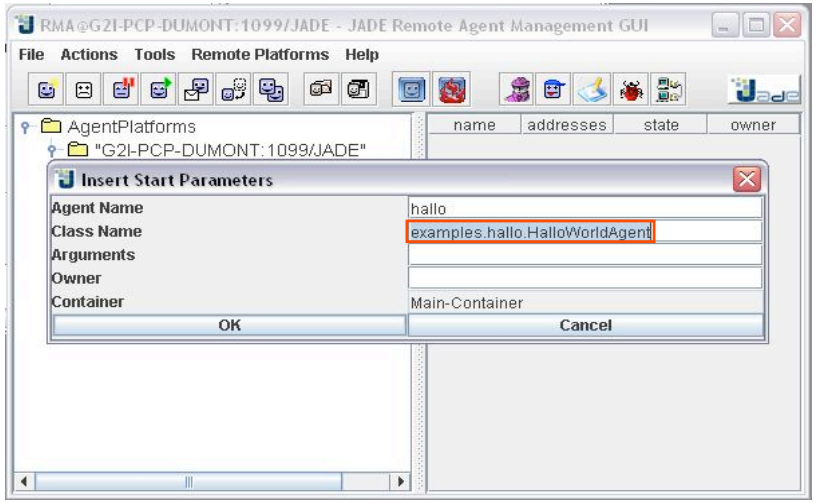
Remote Monitoring Agent



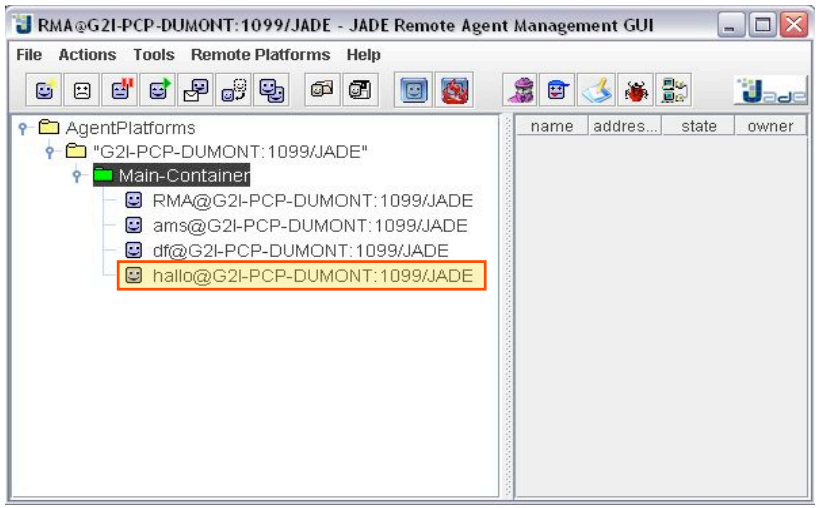
Démarrer un agent au sein du RMA (1/3)



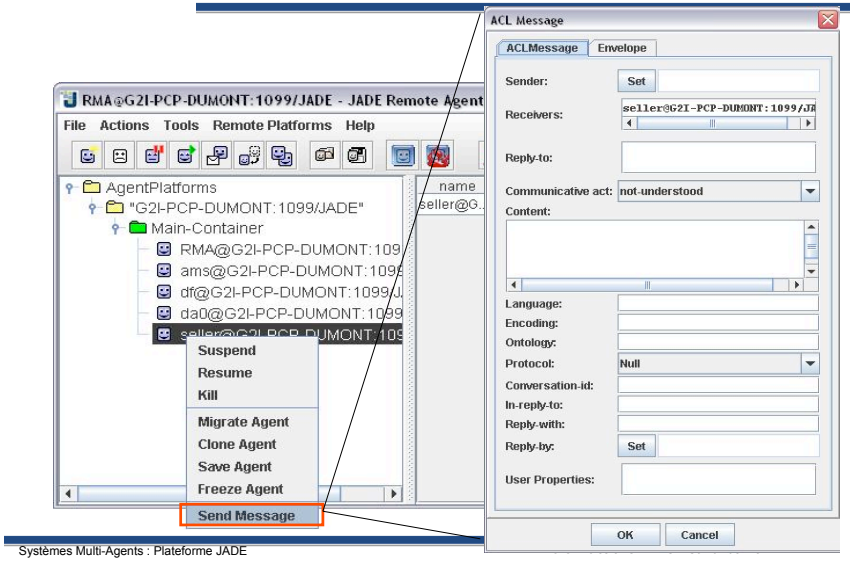
Démarrer un agent au sein du RMA (2/3)



Démarrer un agent au sein du RMA (3/3)



Envoi de messages via le RMA

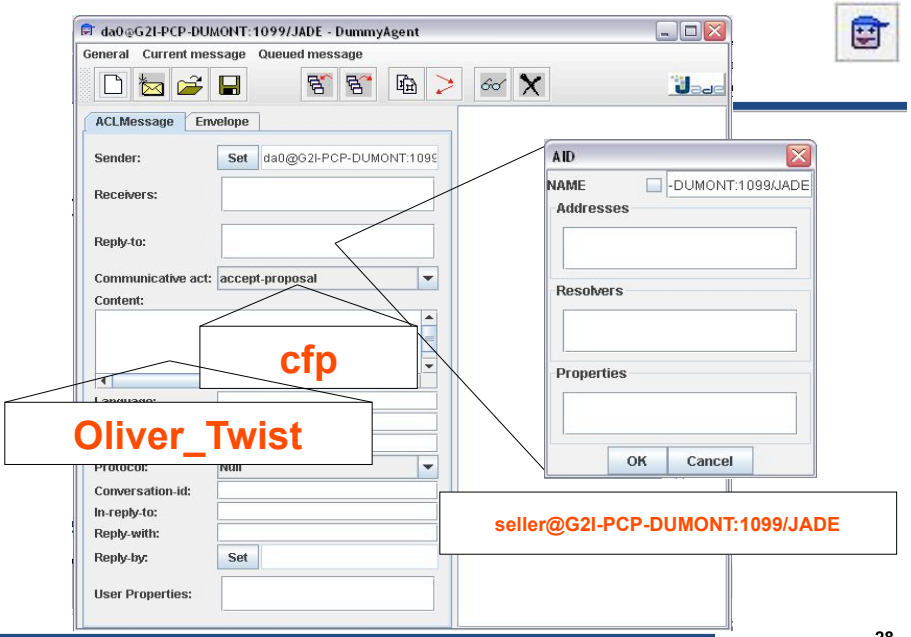


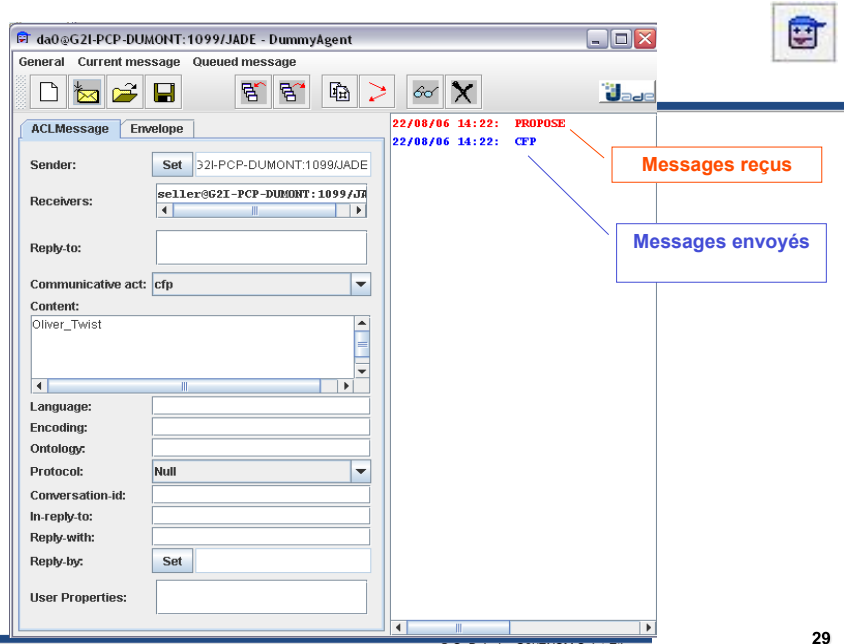
Outils : Dummy Agent

- Lancement :
  - `java jade.Boot monDummy:jade.tools.DummyAgent.DummyAgent`

  Ou
  - Par l'intermédiaire du GUI du RMA

- Agent "vide" permettant à l'utilisateur d'interagir avec les agents de la plateforme
  - Interaction avec les autres agents par envoi de messages définis par l'utilisateur,
  - les messages sont historisés et permettent de voir si les agents du système ont le comportement prévu



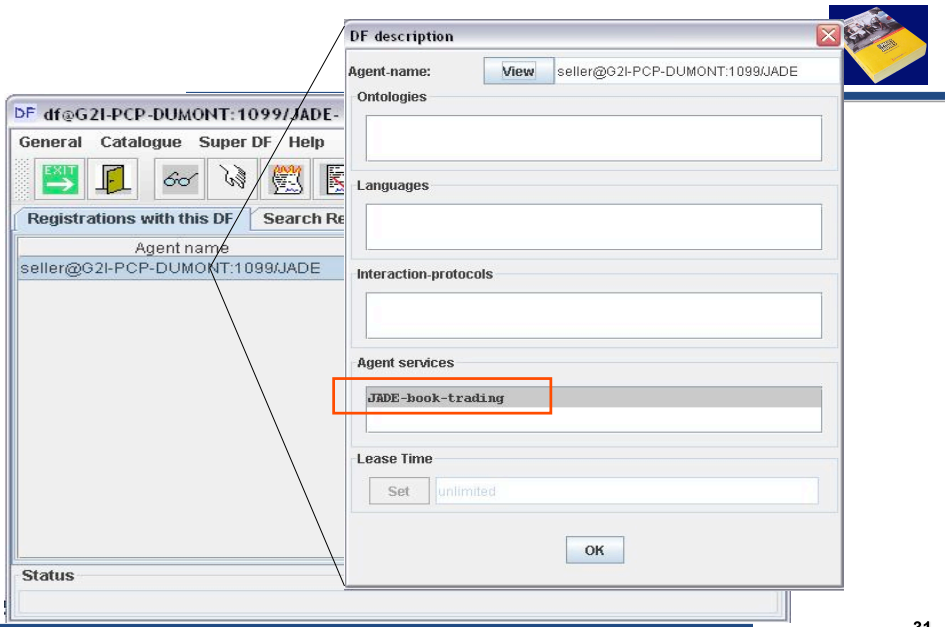


Outils : Directory Facilitator Agent

- Lancement :
  - Le DF est automatiquement lancé au démarrage de la plateforme
  - Lancement de l'interface DF GUI par l'intermédiaire du RMA
  - Lancement par envoi de message au DF :

```
(request :content (action DFName (SHOWGUI)) :ontology
 JADE-Agent-Management :protocol fipa-request)
```

- Le DF n'est disponible qu'au sein du conteneur principal de la plateforme.
- Gestion des services proposés par les agents:
  - Consultation des services mis à disposition par les agents qui se sont inscrits auprès du DF
  - Inscription et désinscription des services mis à disposition par un agent
  - Recherche d'agents qui fournissent un service particulier.

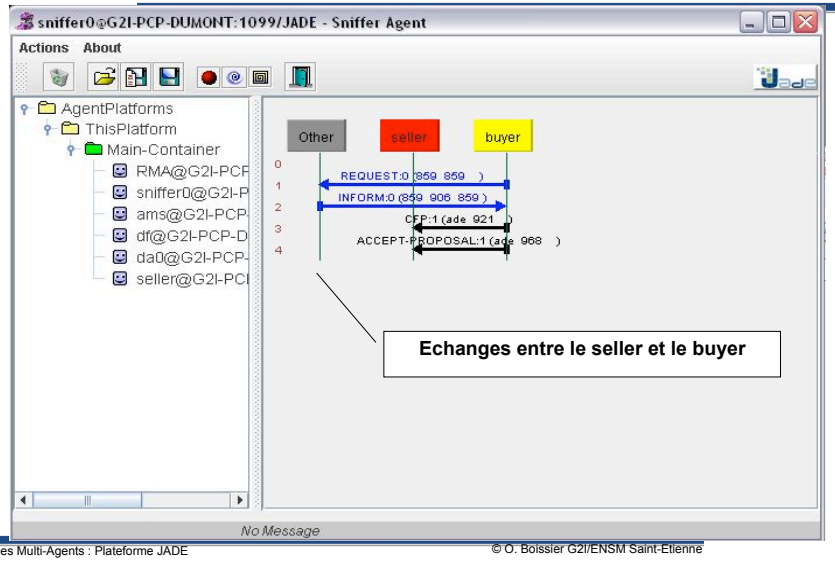


Outils : Sniffer Agent

- Lancement :
  - `java jade.Boot monSniffer:jade.tools.sniffer.Sniffer`

  Ou
  - Par l'intermédiaire du GUI du RMA
- Visualisation des entrées ou sorties d'agents de la plateforme (lorsque utilisation du fichier de configuration)
- Visualisation des messages échangés entre des agents
  - Sélection possible des agents à surveiller
  - Sélection possible des types de messages (performatif) à surveiller pour un agent

Sniffer Agent



Sniffer Agent : fichier de propriétés

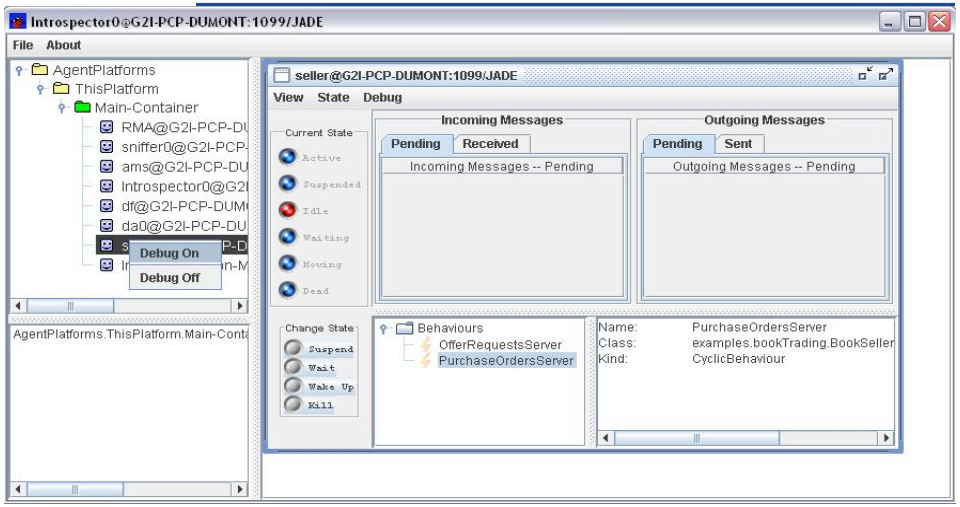
- Utilisation d'un fichier de propriétés `sniffer.properties` est possible pour contrôler le comportement du Sniffer
- preload : liste de descriptions séparées par point-virgule
  - Une description est un nom d'agent et une liste de performatifs séparés par des espaces. Si aucune liste, le Sniffer affichera tous les messages
  - Exemple :
    - `preload=da0;da1 inform propose`
    - `preload=agent?? inform`
    - `preload=*`
- clip : une liste de préfixes de noms d'agent séparés par un point-virgule qui seront retirés de l'affichage des noms d'agents
  - Exemple :
    - `clip=com.hp.palo-alto.;helper`

Outils : Introspector Agent

- Lancement :
  - `java jade.Boot monIntrospector:jade.tools.introspector.Introspector`

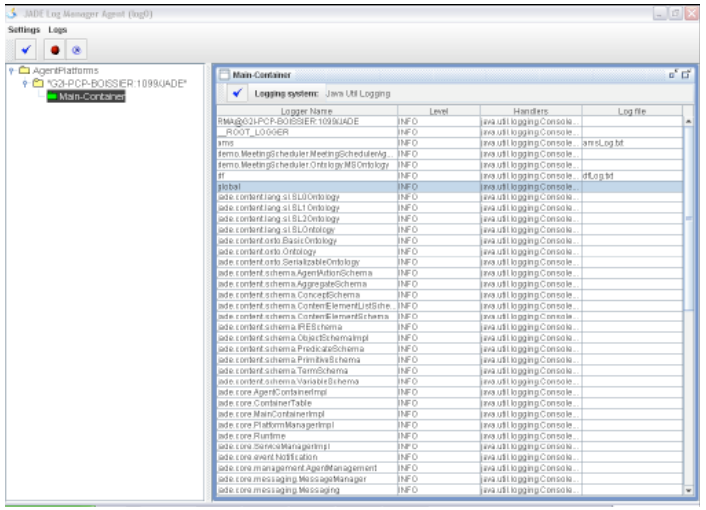
  Ou
  - Par l'intermédiaire du GUI du RMA
- Contrôle du cycle de vie d'un agent
  - Visualisation de l'état d'un agent
  - Changement de l'état d'un agent (note: il faut que l'agent implémente les méthodes correspondantes
    - takedown (), … cf. suite du cours)
- Visualisation des messages reçus, des messages emis mais aussi des messages en attente

Introspector Agent



Outils : Agent Gestionnaire de Logs

- Modification en cours d'exécution du comportement de logging (java.util.logging)
- Permet :
  - Parcours des différents logs sur le conteneur
  - Modification du niveau de logging
  - Ajouter de nouveaux fichiers de logs



Plan

Un ou plusieurs conteneurs

- Règle : tout conteneur simple doit se rattacher à un conteneur principal
- Noms des conteneurs :
  - `-name` spécification du nom du conteneur principal.
    Note : à utiliser avec précaution afin d'assurer un nom unique [par défaut est nomMachine:1099/JADE]
  - `-container-name` spécification du nom d'un conteneur simple [par défaut Container-X (X nombre généré automatiquement].
- Distinction conteneur principal / conteneur simple :
  - `-container` permet de spécifier que l'instance de JADE lancée est un conteneur simple et doit par conséquent rejoindre le conteneur principal [par défaut : conteneur principal]

Lancement d'un conteneur simple

- Lancement d'un conteneur simple sur une machine et rattachement de ce conteneur au conteneur principal situé sur la machine2
  - `java jade.Boot -container -host machine2 -port 1000 [Liste d'agents]`
- Précisions sur le conteneur principal auquel le conteneur simple se rattache
  - `-host` pour spécifier le nom de la machine où se trouve le conteneur principal [par défaut localhost].
    Note : utiliser cette option si l'on veut que la plateforme soit accessible de l'extérieur.
  - `-port` pour spécifier le port d'accès du conteneur principal [par défaut : 1099].

Lancement d'un conteneur sur une machine autre

- Lancement d'un conteneur (simple ou principal) sur une machine et déploiement de ce conteneur sur une autre machine.
  - `java jade.Boot -container -local-host machine2 -local-port 1000 -host machine1 [Liste d'agents]`
- Précisions sur les machines sur lesquelles lancer le conteneur :
  - `-local-host` spécification de l'hôte où doit se trouver le conteneur lancé [par défaut localhost].
  - `-local-port` spécification du port d'accés du conteneur lancé [par défaut 1099].

Plan

Structure d'un agent

- La classe Agent représente la classe de base commune à tous les agents définis par l'utilisateur `jade.core.Agent`
  - Un agent JADE est une instance d'une classe définie par l'utilisateur qui hérite de cette classe Agent
  - Par héritage, il a la possibilité :
    - D'interagir avec la plateforme : enregistrement, configuration, gestion à distance, …
    - D'utiliser un ensemble de méthodes qui peuvent être appelées pour implémenter des comportements particuliers de l'agent (envoi/réception de messages, utilisation de protocoles d'interaction standard, …)
- Le modèle d'exécution d'un agent est multi-tâches dans lequel les comportements (behaviour) d'un agent sont exécutés en parallèle
- Le comportement d'un agent est implémenté par une classe dérivant de la classe `Behaviour`
- Un ordonnanceur, interne à la classe Agent, ordonnance ces comportements

Création d'un agent

- Définition d'une classe d'agent par héritage de la classe jade.core.Agent,
- Implémenter la méthode `setup()` [obligatoire]
  - Invoquée au lancement de l'agent,
  - Utilisée pour :
    - ajouter des comportements à l'agent `addBehaviour()`
    - l'inscrire auprès du DF `DFService.register()`
    - déclarer les ontologies utilisées, le langage de contenu, …
    - traiter les paramètres passés en arguments `getArguments()`
    - `...`
- Implémenter la méthode `takeDown()` [optionnel]
  - Invoquée lors de la fin d'exécution de l'agent,
  - Inclue des opérations de finalisation :
    - Demander au DF de supprimer les services qui ont été inscrits par l'agent,
    - Finir de traiter les messages reçus…

Exemple : Création d'un agent

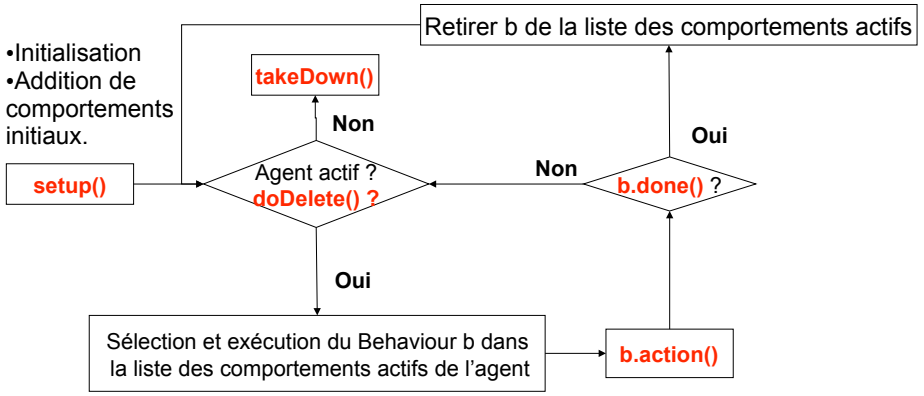
```
import jade.core.Agent;

public class SimpleAgent extends Agent {

    protected void setup(){
        // Initialisation de l'agent
        System.out.println("SimpleAgent ready");
    }

    protected void takeDown(){
        // Traitement de fin
        System.out.println("SimpleAgent done");
    }
}
```


Modèle d'exécution d'un agent

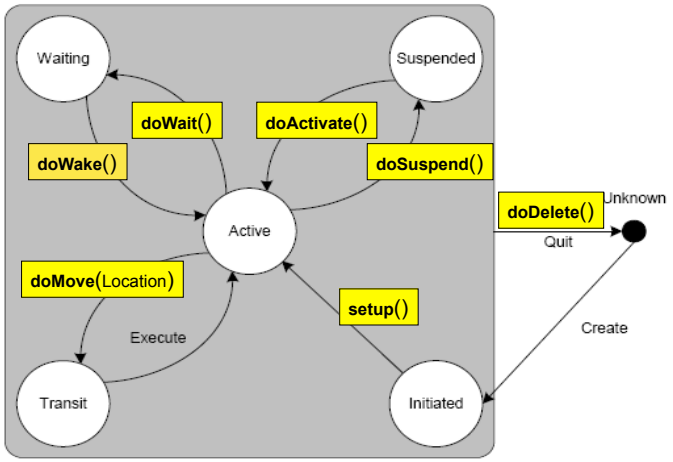


Cycle de vie d'un agent (1/2)

- Durant l'exécution d'un agent, son état peut être modifié
  - INITIATED : l'agent est lancé mais non enregistré auprès de l'AMS, aucun nom, aucune adresse
  - ACTIVE : l'agent est répertorié auprès de l'AMS et peut accéder aux services.
  - SUSPENDED : tous les behaviours de l'agent sont suspendus.
  - TRANSIT : l'agent migre vers une autre plateforme.
  - WAITING : tous les behaviours de l'agent sont temporairement interrompus.
  - DELETED : l'exécution de l'agent est terminée et n'est plus répertorié au sein de l'AMS
- JADE permet de contrôler le passage d'un agent d'un état à l'autre avec les méthodes doXXX

setup()	*INITIATED*
doSuspend() ↔ doActivate()	*SUSPENDED ↔ACTIVE*
doMove()	*ACTIVE ↔ TRANSIT*
doWait() ↔ doWake()	*WAITING ↔ ACTIVE*

Cycle de vie d'un agent (2/2)



Identification d'un agent

- L'agent est identifié auprès de l'AMS par un identifiant unique, appelé GUID (Global Unique Identifier)
- Cet identifiant est un objet de type `jade.core.AID`
  - *`AID(java.lang.String name, boolean isGUID)`*
  - `AID id = new AID(localname, AID.ISLOCALNAME);`
  - `AID id = new AID(name, AID.ISGUID);`
- Les attributs de cet objet sont :
  - name : <nickname>@<plateforme> [getLocalName(), getName()]
  - addresses : adresses de la plateforme [getAllAddresses()]
  - resolvers : services de page blanche [getAllResolvers()]
  - Ces informations sont accessibles par les méthodes de la classe `jade.core.Agent`
    - `getAID()` : retourne l'identifiant unique de l'agent,
    - `getName()` : retourne le nom de l'agent, i.e. <nickname>@<plateforme>
    - `getLocalName()` : retourne uniquement le nickname (nom local),
    - `getHap()` : retourne l'adresse de la plateforme.
    - getAMS() : retourne l'AID de l'AMS de la plateforme sur laquelle l'agent est situé.

Exemple : Identification d'un agent

```
import jade.core.Agent;
public class SimpleAgent extends Agent {

  protected void setup(){
      // Initialisation de l'agent
      System.out.println(this.getName().getAID()+
              "ready");
  }
  protected void takeDown(){
      // Traitement de fin
       System.out.println(this.getName().getAID()+
              "done");
  }
}
```


Passage d'arguments à un agent

- Une liste d'arguments peut être passée à un agent lors de son lancement

```
> java jade.Boot [options] myAgt:myPackage.MyAgent(arg1 arg2)
```

- Les arguments peuvent être récupérés par la méthode `getArguments()` de la classe Agent

```
protected void setup() {
      …
      Object[] args = getArguments();
      if (args != null) {
            for (int i=0; i.args.length; i++) { … }
      }}
```

- Note: les arguments sont temporaires et ne migrent pas avec l'agent et ne sont pas non plus clonés

Aide au développement

- Pour développer des agents JADE :
  - utiliser votre éditeur de texte préféré pour créer le code source de l'agent.
  - Compiler et exécuter cet agent dans le conteneur principal de JADE en chargeant cette classe dans le conteneur.
- Pour développer et améliorer le code d'agents JADE :
  - En ré-éditant et en recompilant, on ne peut pas recharger l'agent dans la plateforme. En effet, en gardant le nom local de l'agent, il y a un conflit de nom. Même en changeant de nom, le chargeur de classe Java ne permet pas le remplacement de classes et entraîne donc l'exécution de l'ancienne version de la classe de l'agent. Une solution est alors de relancer la plateforme !!!
  - C'est bien, mais c'est lourd surtout lorsque l'on a beaucoup d'agents. Le moyen de s'en sortir est de charger l'agent en développement dans un conteneur simple qui se connecte au conteneur principal dans lequel s'exécutent les autres agents. Il suffit ensuite de killer le conteneur simple avec l'agent en développement.

Résumé de la construction d'un agent

- Dériver la classe de l'agent de la classe jade.core.Agent,
- Dans la méthode obligatoire setup()
  - Utiliser la méthode getArguments() pour récupérer les arguments
  - Enregistrer les langages de contenu, les ontologies (voir plus loin)
  - Initier les comportements (voir plus loin)

Structure d'un message ACL (1/3)

- Un message est une instance de la classe `ACLMessage` appartenant au package `jade.lang.acl`
- La définition d'un `ACLMessage` vérifie les spécifications FIPA.
- Il est composé au minimum :
  - d'un performatif,
  - d'un ensemble de destinataires,
- et contient de préférence :
  - un contenu,
  - un expéditeur,
  - un protocole d'interaction,
  - une id de conversation,…
- Il peut spécifier un langage de contenu et/ou une ontologie.

Structure d'un message ACL (2/3)

- Les valeurs des différents attributs d'un `ACLMessage` sont accessibles par les méthodes `getXXX` / `setXXX` / `addXXX` correspondantes

Performatif	`getPerformative()`	`setPerformative()`
Destinataire	`addReceiver()`	`removeReceiver()`
Contenu	`getContent()` ou `getContentObject()` ou `getByteSequenceContent()`	`setContent()` ou `setContentObject()` ou `setByteSequenceContent()`
Expéditeur	`getSender()`	`setSender()`
Protocole	`getProtocol()`	`setProtocol()`
Id de conversation	`getConversationId()`	`setConversationId()`
Langage	`getLanguage()`	`setLanguage()`
Ontologie	`getOntology()`	`setOntology()`
…	…	…

Structure d'un message ACL (3/3)

- Les performatifs définissent le type d'action souhaité par l'agent qui expédie le message.
- Les performatifs sont des variables statiques de la classe `ACLMessage`
  - Exemples :
    - INFORM : envoi d'information/croyance,
    - FAILURE : action non réalisable,
    - REQUEST : demande d'information,
    - SUBSCRIBE : souscription à une source d'information,…
  - Les performatifs disponibles sous Jade, sont ceux définis par la FIPA, pour plus d'information :
    - http://www.fipa.org/specs/fipa00061/SC00061G.html
    - http://www.fipa.org/specs/fipa00037/SC00037J.html
- Le performatif est spécifié en argument du constructeur de l'ACLMessage
  - Exemple : inform = new ACLMessage(ACLMessage.INFORM)

Création et envoi d'un message (1/2)

- Un message est créé comme instance de la classe `ACLMessage`.

```
ACLMessage inform = new ACLMessage(ACLMessage.INFORM);
inform.setContent("contenu");
inform.setProtocol("information");
```

- Suite à la réception d'un message, il peut également être créé en utilisant la méthode `createReply()`
  - remplir le contenu et déclarer le performatif,
  - Les autres champs sont instanciés automatiquement à partir du message reçu.

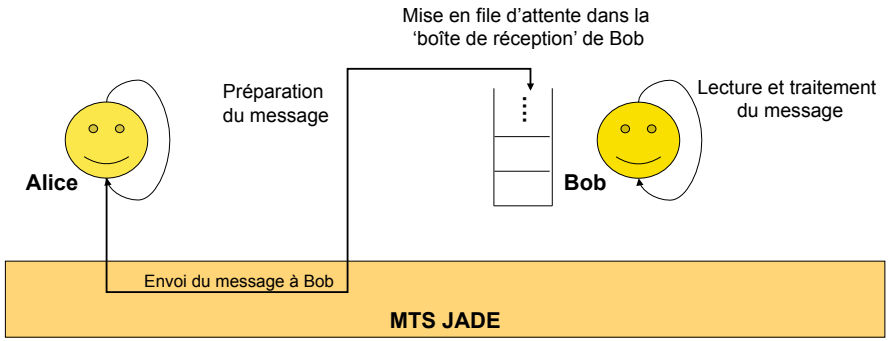
```
reply = msg.createReply();
```


Création et envoi d'un message (2/2)

- Ajouter les destinataires, en utilisant la méthode `addReceiver()`
- Envoi du message par appel de la méthode `send()` de la classe Agent.

```
inform.addReceiver(new AID("destinataire", AID.ISLOCALNAME));
inform.addReceiver(new AID("destinataire2", AID.ISLOCALNAME));
send(inform);
```


Réception de messages (1/5)



Réception de messages (2/5)

- Réception bloquante d'un message :
  - La méthode `ACLMessage m = blockingReceive()` se bloque jusqu'à ce qu'un message arrive. L'agent passe dans l'état '`Waiting`' jusqu'à réception d'un nouveau message.
  - TOUS les comportement seront arrêtés jusqu'à la réception d'un nouveau message.
- Réception non-bloquante d'un message :
  - La méthode `ACLMessage m = receive()` retourne `null` si la file de message est vide
  - L'agent reste dans l'état '`Active`'
    - SON comportement reste activé, et sera exécuté de façon cyclique, même si aucun message n'a été reçu…
    - La solution : utiliser la méthode `block()`, qui suspend LE comportement jusqu'à la réception d'un nouveau message.

Réception de messages (3/5)

- Utiliser `blockingReceive()` dans les méthodes `setup()` et `takeDown()`

```
ACLMessage receive = blockingReceive();
   // Traitement du message
```

- Utiliser `receive()` / `block()` dans le corps des **Behaviours**

```
ACLMessage receive = receive();
if(receive != null){
      // Traitement du message
}
else{
  block();
}
```


Réception de messages (4/5)

- Par défaut, les méthodes `blockingReceive()` et `receive()` récupèrent tous les messages présents dans la 'boîte de réception'
- Réception sélective de messages
  - Utilisation de la classe `MessageTemplate` définie au sein du package `jade.lang.acl`,
    - La classe `MessageTemplate` permet de créer des filtres sur chaque attribut du message : `MessageTemplate.Match<Attribute>()`
    - Ces différentes conditions peuvent être combinées par les méthodes `not()`, `or()` et `and()`.
- L'envoi et la réception de messages peuvent être mis en place au sein de comportements de l'agent

Exemple : Réception de messages (5/5)

```
MessageTemplate perfmt =
   MessageTemplate.MatchPerformative(ACLMessage.INFORM);
MessageTemplate protmt = MessageTemplate.MatchProtocol("ping");
MessageTemplate template = MessageTemplate.and(perfmt, protmt);
ACLMessage receive = receive(template);
if(receive != null){
   ACLMessage inform = receive.createReply();
   inform.setPerformative(ACLMessage.INFORM);
   inform.setContent("pong");
   send(inform);
}
else {
   block();
}
```


Services du DF

- L'envoie d'un message nécessite de connaître l'AID du/des destinataire(s)
- Dans JADE, les inscriptions des agents ne sont pas simultanées.
- Les référencements possibles sont :
  - Soit par leur nom au sein du service de pages blanches (AMS)
  - Soit par le(s) service(s) proposé(s) enregistrés au sein du service de pages jaunes (DF) : `jade.domain.DFService`
    - Contrairement à l'AMS ce référencement n'est pas automatique
    - Chaque agent doit s'inscrire auprès du DF : `DFService.register()`
    - Chaque agent doit se désinscrire du DF : `DFService.deregister()`
- Le DF permet de rechercher les agents offrant un type de service désiré : `DFService.search()`

Référencement auprès du DF (1/2)

- Pour se référencer, l'agent doit décrire:
  - Les services qu'il fournit
    - Type, nom, langage(s), ontologie(s), protocole(s), propriété(s),
    - Classe `jade.domain.FIPAAgentManagement.ServiceDescription`
    - Classe `jade.domain.FIPAAgentManagement.DFAgentDescription`

```
// Enregistrement aupres du directory facilitator
try {
   DFAgentDescription dfd = new DFAgentDescription();
   dfd.setName(this.getAID());
   ServiceDescription sd = new ServiceDescription();
   sd.setType("participant");
   dfd.addServices(sd);

   DFService.register(this, dfd);
}
catch (FIPAException fe) {fe.printStackTrace();}
```

- Il est recommandé de se référencer à l'entrée dans le système

Référencement auprès du DF (2/2)

- Inversement, l'agent doit se désinscrire du DF à sa sortie du système :
  - Méthode `deregister()`

```
try{
  DFService.deregister(myAgent);
}
catch(FipaException fe){
  fe.printStackTrace();
}
```


Recherche au sein du DF

- Recherche de la description d'agents fournissant un service particulier

```
// Description du service des experts
DFAgentDescription participants = new DFAgentDescription();
ServiceDescription sde = new ServiceDescription();
sde.setType("participant");
participants.addServices(sde);

try{
   ACLMessage reply = msg.createReply();
   reply.setPerformative(ACLMessage.INFORM);
   DFAgentDescription[] results = DFService.search(agent, participants);
   for (int i = 0; i < results.length; ++i){
        reply.addReceiver(results[i].getName());
   }
   agent.send(reply);
}
catch (FIPAException e) {e.printStackTrace();}
```

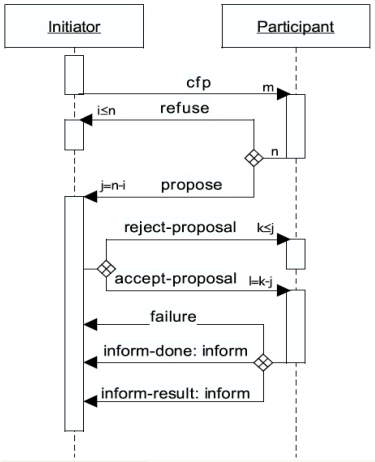

Protocoles d'interaction

- Les protocoles spécifient l'enchaînement des messages entre les agents.
- Ils définissent les messages à envoyer en précisant :
  - Le performatif
  - Le contenu
  - Le(s) destinataire(s)

  et établissent les réponses à ses messages.
- Jade propose des protocoles prédéfinis disponibles dans le package `jade.proto`
  - Ce package contient des protocoles 'clé en main'…
  - … répondant aux spécifications qui ont été établies par la FIPA
- Pour plus d'informations : http://www.fipa.org/repository/ips.php3

Détails du Contract Net Protocol

- Appartient aux protocoles du package jade.proto
- Développer des agents utilisant le CNP, nécessite de définir les comportements :
  - De l'Initiateur de l'interaction
    - `jade.proto.ContractNetInitiator`
  - Du Participant à l'interaction
    - `jade.proto.ContractNetResponder`
- et de surcharger les méthodes :
  - `handle<Performatif>`
  - `registerHandle<Performatif>`



Détails du Contract Net Protocol

- Initiateur :
  1. Recherche un bien dont le prix ne dépasse pas une valeur seuil donnée (**CFP**),
  2. Accepte la proposition qui maximise l'écart entre le seuil et les prix proposés par les différents participants puis prévient le vainqueur (**accept-proposal**).
  3. Refuse les autres propositions en prévenant les participants vaincus (**reject-proposal**).
- Participant *:*
  1. Répond à la demande de l'Initiateur
     - **propose** s'il possède le bien en spécifiant le prix demandé,
     - **refuse** sinon.
  2. Informe l'Initiateur du résultat de l'exécution.
     - **inform** si l'exécution est réussie,
     - **failure** sinon.

Agent Initiateur : Initialisation de l'agent

```
import jade.core.Agent;
import jade.core.AID;
...
public class Initiateur extends Agent {
   private double prix;
   protected void setup(){
      Object[] args = getArguments();
      if (args != null && args.length == 2) {
        prix = Double.parseDouble((String)args[1]);
        ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
        cfp.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
        cfp.setReplyByDate(new Date(System.currentTimeMillis() + 10000));
        cfp.addReceiver(new AID("participant", AID.ISLOCALNAME));
        cfp.setContent((String)args[0]);
        addBehaviour(new ContractNetInitiator(this, cfp) {
         // Traitement des messages : surcharge des méthodes handle });
        } else  {System.out.println("Usage : <titre> <seuil>");}
   } }
```


Agent Initiateur : Traitement des messages (1/3)

- Réception d'un message PROPOSE

```
protected void handlePropose(ACLMessage propose, Vector v) {
        System.out.println("Info : L'agent
     "+propose.getSender().getName()+" propose
     "+propose.getContent());
   }
```

- Réception d'un message REFUSE

```
protected void handleRefuse(ACLMessage refuse) {
        System.out.println("Info : L'agent
     "+refuse.getSender().getName()+" ne possède pas l'objet
     désiré");
   }
```

Agent Initiateur : Traitement des messages (2/3)

- Réception d'un message FAILURE

```
protected void handleFailure(ACLMessage failure) {
    if (failure.getSender().equals(myAgent.getAMS())) {
        System.out.println("Le destinataire n'existe pas");
    }
    else {
        System.out.println("Erreur : Action non réalisée"
        +failure.getSender().getName()+")");
    }
}
```

- Réception d'un message INFORM

```
protected void handleInform(ACLMessage inform) {
    System.out.println("Succes : Achat réalisé avec succès
    auprès de l'agent "+inform.getSender().getName());
}
```


Agent Initiateur : Traitement des messages (3/3)

- Préparation des réponses

```
protected void handleAllResponses(Vector reponses, Vector acceptes) {
    double bestPrice = 0;
    ACLMessage accept = null;
    for(Enumeration e = reponses.elements(); e.hasMoreElements();){
        ACLMessage msg = (ACLMessage) e.nextElement();
        if (msg.getPerformative() == ACLMessage.PROPOSE) {
            ACLMessage reply = msg.createReply();
            reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
            acceptes.addElement(reply);
            double price = Double.parseDouble(msg.getContent()) - prix;
            if (price < bestPrice) {
                bestPrice = price;
                accept = reply;
            }
        }
    }
    if (accept != null)
        accept.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
}
```


Agent Participant : Initialisation de l'agent

```
import jade.core.Agent;
import jade.core.AID;
...
public class Participant extends Agent {
    private Hashtable have;
    protected void setup() {
        MessageTemplate template = MessageTemplate.and(
    MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET),
    MessageTemplate.MatchPerformative(ACLMessage.CFP));
        have = new Hashtable();
        Object [] args = getArguments();
        int i = 0;
        while(i<args.length){
     have.put(args[i++], args[i++]);
        }
        addBehaviour(new ContractNetResponder(this, template) {
         // Traitement de messages : surcharge des méthodes handle } );
   } }
```


Agent Participant : Traitement des messages (1/2)

- Réception d'un message CFP

```
protected ACLMessage handleCfp(ACLMessage cfp)
  throws RefuseException, FailureException,
  NotUnderstoodException {
  String title = cfp.getContent();
  if(have.containsKey(title)){
     double price = Double.parseDouble((String) have.get(title));
     ACLMessage propose = cfp.createReply();
     propose.setPerformative(ACLMessage.PROPOSE);
     propose.setContent((new Double(price)).toString());
     return propose;
  } else {
     throw new RefuseException("Objet non disponible");
  } }
```

Agent Participant : Traitement des messages (2/2)

- Réception d'un message Accept-Proposal

```
protected ACLMessage handleAcceptProposal(ACLMessage cfp,
  ACLMessage propose, ACLMessage accept)
  throws FailureException {
  String title = cfp.getContent();
  if (have.containsKey(title)) {
     ACLMessage inform = accept.createReply();
     inform.setPerformative(ACLMessage.INFORM);
     have.remove(title);
     return inform;
  }
  else {throw new FailureException("Vente impossible")}
}
```


Comportements

- Les comportements sont définis par dérivation
  - de la classe `jade.core.behaviours.Behaviour`
  - ou d'une de ses classes dérivées
- Les méthodes suivantes doivent être surchargées :
  - `void action()`
    - La fonctionnalité du comportement !
    - Doit être rapide !!
  - `boolean done ()`
    - Retourne vrai lorsque la tâche implémentée par le comportement est terminée
- Attribut manipulable : `myAgent`
  - Utilisé pour envoyer/recevoir des messages et ajouter des comportements

Comportements

- Les comportements "behaviours" définissent le comportement de l'agent.
- L'ajout d'un comportement est réalisé par la méthode `addBehaviour()`
  - Utilisé dans la méthode `setup()`
  - ou dans un autre comportement de l'agent
- L'ordonnanceur interne d'un agent exécute un comportement, action par action jusqu'à ce que la condition de fin `done()` du comportement soit satisfaite
  - Gestion d'une liste de comportements actifs,
  - Choix d'un comportement
  - Exécution de la méthode `action()` du comportement
  - Exécution de la méthode `done()` du comportement pour tester la fin du comportement
  - Si le comportement est terminé, retrait de la liste de comportements actifs, sinon insertion du comportement dans la liste des comportements actifs

Méthodes d'un comportement

- Ajout/Suppression d'un comportement
  - `addBehaviour()`
  - `removeBehaviour()`
- Actions exécutées au lancement/terminaison du comportement
  - `onStart()`
  - `onEnd()`
- Suspension de l'exécution du comportement
  - `block(), block(long milliseconds),`
- Reprise de l'exécution du comportement
  - `restart()`
- Réinitialisation du comportement
  - `reset(), reset(…)`:

Exemple de comportement

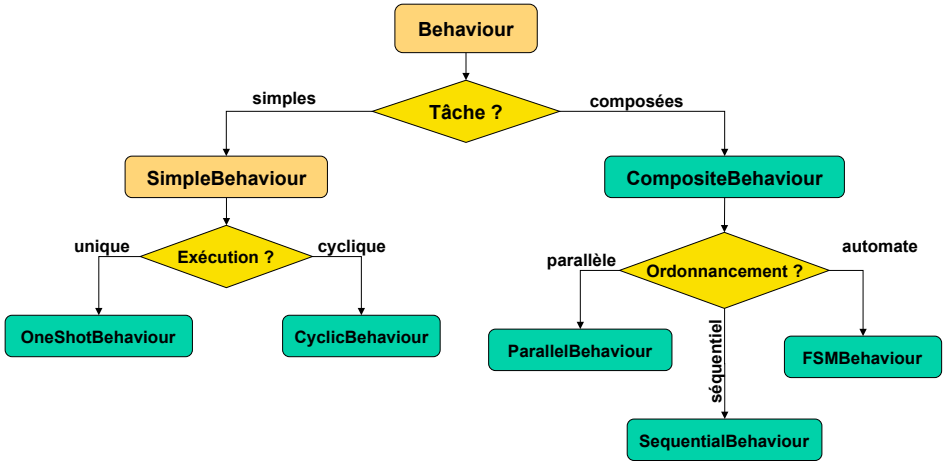
```
public class BehaviourExampleAgent extends Agent {
      protected void setup () {
            addBehaviour(new TrivialBehaviour());
      }

      public class TrivialBehaviour extends Behaviour {

            public action() {
             System.out.println(''exemple de comportement'');
            }

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


Types de comportements



Initiateur : behaviour (1/3)

```
import jade.core.behaviours.SimpleBehaviours;

public class InitiateurBehaviour extends SimpleBehaviour {
   private boolean finished = false;
   private Initiateur init;
   public InitiateurBehaviour(Initiateur agent){
     // Constructeur
   }
   pulic void action(){
     // Description du comportement de l'agent
   }

   public boolean done(){
     // Renvoie true ssi la méthode action ne doit plus être invoquée
     return finished;
   }
 }
```


Initiateur : behaviour (2/3)

- Envoi du 'Call For Proposal' à l'agent Participant

```
public InitiateurBehaviour(Initiateur agent){
  init = agent;

  // Création du CFP
  ACLMessage cfp = new ACLMessage(ACLMessage.CFP);

cfp.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT
_NET);
  cfp.addReceiver(new AID("participant", AID.ISLOCALNAME));
  cfp.setContent(init.titre);

  // Envoi du message
  init.send(cfp);
}
```

Initiateur (3/3) : behaviour de traitement de messages

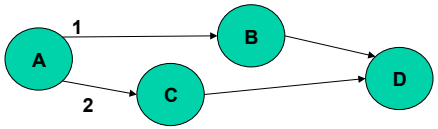
```
public void action(){
   ACLMessage receive =
        init.receive(MessageTemplate.MatchProtocol(FIPANames.Interaction
        Protocol.FIPA_CONTRACT_NET));
   if(receive != null){
      if(receive.getPerformative() == ACLMessage.REFUSE);
      if(receive.getPerformative() == ACLMessage.FAILURE);
      if(receive.getPerformative() == ACLMessage.INFORM);
      if(receive.getPerformative() == ACLMessage.PROPOSE){
        // Evaluer la propositions
        ACLMessage reply = receive.createReply();
        if(ok){
           reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
           finished = true;
        }
        else {reply.setPerformative(ACLMessage.REJECT_PROPOSAL);}
        init.send(reply); }
   else{block();} }
```


Composite Behaviour

- Les comportements complexes sont des dérivations de la classe `CompositeBehaviour` :
  - `SequentialBehaviour` et `ParallelBehaviour`
    - Exécution en séquentiel :
      - Arrêt lorsque tous les sub-behaviours sont terminés
    - Exécution en parallèle
      - Arrêt lorsque un ou plusieurs comportement sont terminés,
      - Arrêt conditionné par le programmeur.
    - Ajout de 'sous-comportements' : `addSubBehaviour()`
    - Suppression : `removeSubBehaviour()`
- Avantages :
  - Décomposition de tâches complexes,
  - Simplification de la programmation.

FSM Behaviour

- Enchaînement des 'sous-comportement' à la manière d'un automate



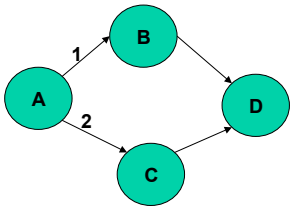

```
FSMBehaviour fsm = new FSMBehaviour(this);

fsm.registerFirstState(new behaviourA(), "A");
fsm.registerState(new behaviourB(), "B");
fsm.registerState(new behaviourC(), "C");
fsm.registerLastState(new behaviourD(), "D");

fsm.registerTransition("A", "B", 1);
fsm.registerTransition("A", "C", 2);
fsm.registerDefaultTransition("B", "D");
fsm.registerDefaultTransition("C", "D");

addBehaviour(fsm);
```


OneShotBehaviour




```
public class behaviourA extends OneShotBehaviour {
   private Agent agent;
   private int exit;

   public A(Agent agent){
     this.agent = agent;
   }

   public void action(){
     if(agent.getLocalName().equals("A")){
        exit = 1;
     }
     else {
        exit = 2;
     }
   }
   public int onEnd() {
        return exit;
   }
}
```

Ticker/Waker Behaviour

- Jade propose deux autres behaviours :
  - `TickerBehaviour`
    - Hérite de `SimpleBehaviour`
    - Exécution régulière d'une action :
      - Implémentation de la méthode `onTick()`
  - `WakerBehaviour`
    - Exécution unique de l'action à un temps T :
      - Implémentation de la méthode **`handleElapsedTimeOut()`**
- *Utilité* : sauvegarde, ping, automatisation de tâches,…

Aperçu de JADE API

- jade.core :
  - Implémentation du noyau de la plateforme JADE
  - Classe Agent qui doit être spécialisées par les programmeurs d'application.
  - jade.core.behaviours : classe de comportements Behaviours qui implémentent les tâches ou les intentions d'un agent.
- jade.content :
  - ensemble de classes support aux ontologies définies par les utilisateurs et les langages de contenu.
  - jade.content.lang.sl : SL codec2, parser et encodeur.
- jade.lang.acl :
  - traitement du Agent Communication Language selon les spécifications de la FIPA.
- jade.proto :
  - Classes pour la modélisation des protocoles d'interaction standard (*fipa-request*, *fipa-query, fipa-contract-net, fipa-subscribe,* …), ainsi que les classes pour aider les programmeurs d'application à créer leurs propres protocoles.
- jade.FIPA :
  - Module IDL défini par la FIPA pour le transport de message respectant le protocole IIOP.
- jade.wrapper :
  - wrappers des fonctionnalités de haut niveau de JADE permettant d'utiliser JADE comme une bibliothèque, où des applications Java externes lancent des agents et des conteneurs.

Aperçu de JADE API

- jade.domain package :
  - Ensembe des classes qui représentent les entités de gestion des agents définies par la FIPA, en particulier l'AMS, le DF, ainsi que toutes les constantes définies par la FIPA
  - jade.domain.FIPAAgentManagement : Ontologie FIPA-Agent-Management et toutes les classes représentant ses concepts.
  - jade.domain.JADEAgentManagement : Extensions JADE pour la gestion des agents (espionnage des messages, contrôle du cycle de vie des agents, …), incluant l'ontologie correspondante et toutes les classes correspondantes.
  - jade.domain.introspection : Concepts utilisés pour les communications entre outils JADE (Sniffer et Introspector par exemple) et le noyau JADE.
  - jade.domain.mobility : Ensemble des concepts utilisés pour la communication sur la mobilité.
- jade.gui :
  - Ensemble de classes génériques pour la création d'interfaces graphiques pour afficher et éditer des Agent-Identifiers, Agent Descriptions, ACLMessages, …
- jade.mtp :
  - Interface Java que tout protocole de transport de message doit implémenter pour être intégré dans le framework JADE, implémentation de l'ensemble de ces protocoles.
- jade.tools :
  - Classes implémentant des outils pour simplifier l'administration et le développement d'applications
- jade.util (resp. jade.util.leap) :
  - Classes utilitaires pour la gestion des propriétés notamment (resp. remplacement des collections pour J2ME)