

TP n°3

Projet - Exploration de l'Environnement

1 Principales caractéristiques de la plateforme fournie

L'archive disponible *DedaleEtu* est un projet Maven¹. Maven est un outil facilitant la gestion et la production de projet Java. Il vous permettra (notamment) de télécharger automatiquement les différentes bibliothèques dont vous aurez besoin, leur code source et la javadoc par l'appel de la commande *maven install*.

Dans sa configuration actuelle, les bibliothèques exploitées sont *JADE* (plateforme agent), *Junit* (tests-unitaires), *GraphStream*² (graphes dynamiques) et *Dedale* (environnement du projet). La bibliothèque *Dedale* sera progressivement mise à jour durant le semestre, à mesure que de nouvelles fonctionnalités vous seront nécessaires.

Le code source fourni pour lancer la version de base du projet comprend la classe *Principal* (utilisée pour la création de l'environnement et de la plateforme) et un exemple d'agent : *DummyExploAgent*.

- La fonction *main()* de la classe *Principal* vous permettra, outre de déclarer vos agents via la méthode *createAgents()* introduite lors du précédent TP, de paramétrer votre environnement.

```
public static void main(String[] args){

    //0) Create the environment
    //env= new Environment(ENVtype.GRID,15,null);
    env= new Environment(ENVtype.DOROGVTSEV,15,null);

    //1), create the platform (Main container (DF+AMS) + containers + RMA and SNIFFER)
    rt=emptyPlatform(containerList);

    //2) create agents and add them to the platform.
    agentList=createAgents(containerList);

    //3) launch agents
    startAgents(agentList);
}
```

Comme indiqué dans le sujet du projet, l'environnement dans lequel vos agents vont évoluer est un graphe non-orienté. Les noeuds du graphe représentent les pièces, et les

1. Si Maven n'est pas installé dans votre version d'eclipse, faire *Help/Install new software/work with all available sites*, filtrer avec le mot clé "Maven" et installer m2e.

2. <http://graphstream-project.org/>

arrêtes les couloirs qui les lient. Deux agents ne peuvent être simultanément sur un même noeud. Le premier paramètre du constructeur *Environment()* correspond au type d'environnement (une grille ou un graphe pour le moment), le second à la taille de celui-ci, et le dernier au fichier décrivant la carte à charger le cas échéant.

- L'agent *DummyExploAgent* est rattaché à l'environnement lors de son initialisation (méthode *setup()*), puis doté de deux comportements :
 - *RandomWalkBehaviour* : Permet à l'agent d'évoluer dans, et d'interagir avec, l'environnement.
 - *SayHello* : Contacte un autre agent pour tenter de lui transmettre sa position courante

2 API avec l'environnement

A l'instar de l'agent *DummyExploAgent*, tous vos agents doivent impérativement hériter de la classe *AbstractAgent*. C'est elle qui fournira à vos agents les méthodes d'interaction avec les autres agents et l'environnement. EN voici les principales :

- ***void deployAgent(Environment env)***
Fonction à appeler uniquement dans la méthode *setup()* de vos agents pour les lier à l'environnement
- ***String getCurrentPosition()***
Retourne la position courante de l'agent
- ***List<Couple<String,List<Attribute>>> observe()***
Retourne l'ensemble des observables depuis la position courante de l'agent sous la forme d'une liste de couple (position, liste attribut/valeur)
- ***boolean moveTo(String myDestination)***
Permet à votre agent de se déplacer dans l'environnement jusqu'à la position fournie en paramètre (si atteignable). Afin de garantir la cohérence de votre agent et de vous faciliter le travail d'implémentation, il est plus que recommandé que cette fonction, lorsque elle est appelée, soit la dernière méthode de votre *behaviour*.
- ***void sendMessage(ACLMessage msg)***
Fonction d'envoi de message qui gère le rayon de communication de agents. Vous devez utiliser **exclusivement** celle-ci.
- ***int getBackPackFreeSpace()***
Retourne l'espace libre dans le sac à dos de l'agent.
- ***int pick()***
Permet de récupérer tout ou partie du trésor présent sur la position courante (en fonction de la capacité d'export de l'agent)

Les deux dernières fonctions (relatives aux trésors) ne seront pas utilisées durant les premières semaines, et d'autres seront ajoutées ultérieurement. La Javadoc vous donnera le détail complet de l'ensemble de ces fonctions, et le code du *DummyExploAgent* illustre l'utilisation de celles-ci.

3 Etape 1 : Exploration et représentation de l'environnement

Proposez un mécanisme permettant à vos agents d'explorer collectivement l'environnement. Le protocole que vous proposerez devra être en mesure de fonctionner quel que soit le nombre d'agents de votre équipe et le rayon de communication de ceux-ci ³.

Réfléchissez consciencieusement aux différentes situations que vos agents sont susceptibles de rencontrer, dessinez le protocole envisagé et identifiez les rôles associés. Analysez les différentes façon de décomposer ceux-ci en behaviours en tenant compte de l'impact des différentes solutions sur l'information à représenter et partager entre les behaviours d'un agent ainsi qu'entre les agents eux-mêmes (Complexité en temps, mémoire, nombre de message, taille des messages,..)

à noter que le contenu d'un message entre agent n'est pas limité à une chaîne de caractère. L'utilisation de la méthode *msg.setContentObject(Object o)* vous permet de transmettre tout objet **sérialisable**.

4 Comment un agent peut-il obtenir une liste de noms d'agents ?

Trois possibilités : (les méthodes 1 et 2 sont à préférer)

1. Disposer en local de la liste des différents agents qui vous intéressent. Cela nécessite que lorsqu'un nouvel agent arrive dans votre équipe, il en informe les agents présents et que ceux-ci mettent à jours leurs connaissances. Attention, comment faire lorsque un agent quitte l'équipe (de manière voulue, ou non..)
2. Interroger les pages jaunes Comme indiqué précédemment, cela nécessite que vos agents déclarent leurs services/capacités au DF puis que chaque agent interroge DF lorsqu'il souhaite trouver les agents en mesure de répondre à l'un de ces besoin. Remarque : DF devient alors un point de centralisation partiel.

<http://jade.tilab.com/doc/programmersguide.pdf>, page 9 et suivantes.

— Pour s'enregistrer sur le DF :

```
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID()); /* getAID est l'AID de l'agent qui veut s'enregistrer*/
ServiceDescription sd = new ServiceDescription();
sd.setType( "explorer" ); /* il faut donner des noms aux services qu'on propose
(ici explorer)*/
sd.setName(getLocalName() );
dfd.addServices(sd);

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

— Et pour rechercher les agents qui proposent un service sur le DF :

```
DFAgentDescription dfd = new DFAgentDescription();
```

3. Pour cette première séance, la rayon de communication est fixé de manière homogène à 3

```

ServiceDescription sd = new ServiceDescription();
sd.setType( "explorer" ); /* le même nom de service que celui qu'on a déclaré*/
dfd.addServices(sd);
DFAgentDescription[] result = DFService.search(this, dfd);
System.out.println(result.length + " results" );
if (result.length>0)
    System.out.println(" " + result[0].getName() );

```

3. Interroger l'AMS

Similaire au DF, mais un peu difficile de trouver les fonctions d'appel, donc je vous donne un début d'exemple ci-dessous.

```

QueryAgentsOnLocation queryLocation = new QueryAgentsOnLocation();
queryLocation.setLocation(cont);
Action a = new Action();
a.setActor(getAMS());
a.setAction(queryLocation);
ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
msg.setOntology(MobilityOntology.NAME);
msg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
msg.setLanguage(FIPANames.ContentLanguage.FIPA_SL);
msg.addReceiver(getAMS());
try{
getContentManager().fillContent(msg,a);
    send(msg);
}catch(Exception e) { e.printStackTrace(); }

```

```

MessageTemplate mt = MessageTemplate.MatchSender(getAMS());
ACLMessage response = blockingReceive(mt); /*Warning blockingReceive block the agent,
not just the behaviour*/
Result results = null;
try {
    results = (Result)
        getContentManager().extractContent(response);
} catch (UngroundedException e) { e.printStackTrace();
    } catch (CodecException e) {
        e.printStackTrace();
    } catch (OntologyException e) {
        e.printStackTrace();
    }
}
List agentlis=new ArrayList();
agentlist = (List) results.getItems();
for(int i=0;i<agentlist.size();i++){
    agentlis.add(agentlist.get(i));
}
System.out.println("Agent name"+agentlist.get(0));

```