TD Applications Réparties

Programmation A³
Agent AnyTime Anywhere

Sommaire

Table des matières

1. Modèle de programmation par agents AAA

Dans le cadre de la programmation répartie, on peut considérer que les objets mis en œuvre dans une telle application sont conformes à un modèle unique où chacun d'eux sont passifs et en attente de requêtes auxquelles ils donneront suite par des réactions appropriées. Ce modèle est très proche de celui d'objet, il s'ensuit que son implantation avec un environnement objets se fera aisément. Les agents AAA (noté A^3) sont donc des objets java "réactifs" (et passifs car ils ne disposent pas, a priori, d'autonomie opérationnelle) qui se comportent conformément au modèle "événement $\prod$ réaction" : Un événement est une transition d'état significative à laquelle un ou plusieurs agents vont devoir réagir.

Dans ce modèle, un événement est représenté par une notification ; Une notification est un objet qui est transmis à l'agent destinataire afin qu'il exécute la réaction appropriée. En ce sens il constitue la réification d'un envoi de message dans le modèle objet traditionnel. L'envoi de notification représente le seul moyen de communication entre agents ; Les notifications transitent entre agents via un bus à message qui abstrait toute la mécanique opératoire que cela suppose.

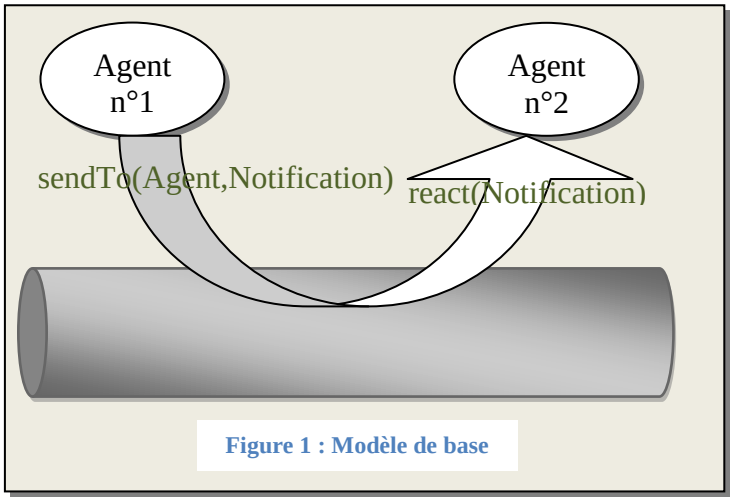

Figure 1 : Modèle de base

1.1. Propriété de l'infrastructure

1.1.1. Propriétés des agents

- **Persistance** : L'état d'un agent est persistant.

Les agents A3 sont persistants : La durée de vie d'un agent n'est pas liée à la durée de vie de l'exécution qui l'a créé. L'état d'un agent n'est pas perdu lorsque la machine sur laquelle il s'exécute, s'arrête ou tombe en panne. L'état est sauvegardé sur un support persistant.

- **Atomicité** : La réaction d'un agent est atomique.

Cela veut dire qu'une réaction se termine normalement ou bien qu'elle est annulée, dans ce cas l'agent retrouve l'état qu'il avait au démarrage de cette réaction sinon un nouvel état est associé à l'agent en fonction de la réaction qu'il a eu. En particulier, toutes les notifications envoyées durant une réaction ne sont effectivement émises que lorsque la réaction est validée. Pour cela un modèle transactionnel permet d'assurer cette propriété.

1.1.2. Propriétés des Notifications

- **Persistance** : L'état d'un agent est persistant.

Les notifications A^3 sont persistantes : La durée de vie d'une notification n'est pas liée à la durée de vie de l'exécution qui l'a créé. L'état est sauvegardé sur un support persistant.

1.1.3. Propriétés des communications

Dans l'infrastructure de communication, les notifications sont transformées en messages. L'infrastructure de communication est basée sur un bus logiciel (Message Oriented Middleware - MOM) qui assure des propriétés d'asynchronisme, de fiabilité et d'ordonnancement sur les communications.

- **Asynchronisme** : l'exécution des agents producteurs et consommateurs d'événements peuvent être découplées dans le temps.

Un agent A1 peut envoyer une notification n1 à un agent A2 même si ce dernier n'est pas en cours d'exécution. La propriété d'asynchronisme fournie par le bus, associée à la persistance des messages assure que l'agent A2 recevra bien la notification n1 lorsqu'il sera en cours d'exécution.

- **Fiabilité** : Toute notification envoyée est garantie d'arriver à destination malgré l'occurrence de panne machine ou réseau.

Ce type de fiabilité doit être différencié de celle fournie par TCP qui ne traite seulement que le cas de perte de message, de doublons et d'ordonnancement. Il ne prend pas en compte le cas de panne machine ou de réseau.

- **Causalité** : Les notifications sont délivrées à leurs destinataires suivant leur ordre causal d'émission. Ceci peut être représenté par le schéma ci-contre où l'on dispose de 3 agents dont le premier (a1) envoie des notifications (n1 et n2) à destination des 2 autres. On dénote la précédence (le fait qu'une action précède une autre dans le temps) par l'opérateur <. L'ordonnancement causal consiste alors à garantir que :

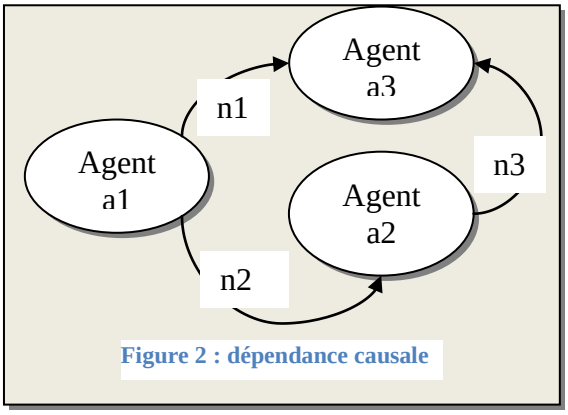

Figure 2 : dépendance causale

Sendto(a1,n1) < SendTo(a1,n2) => react(a3,n1) < react(a3,n3)

1.2. Modèle programmatique

1.2.1. Les agents

Tout agent doit hériter de la classe Agent qui contient le comportement de base de tous les agents. Chaque agent est identifié de manière unique à l'aide d'un AgentId. Les méthodes suivantes sont définies dans Agent :

- **public Agent(short to)** : un constructeur d'agent, "to"[1] indique le site de déploiement de l'agent
- **public Agent(short to, String n)** : idem, n représente le nom symbolique de l'agent.
- **public final void deploy()** : Cette méthode permet de déployer l'objet sur son site d'exécution et ainsi d'en faire un agent.
- **public String toString()** : Permet d'obtenir la représentation textuelle de l'agent.
- **public final AgentId getId()** : Permet d'obtenir l'identificateur unique de l'agent.
- **public void react(AgentId from, Notification not)** : Chaque agent doit définir une méthode "react" qui spécifie le comportement de l'agent. Lorsqu'une notification n_i arrive, la méthode "react" est appelée avec pour paramètre : l'id de l'agent émetteur de la notification et la notification n_i. C'est dans la méthode "react" que sont analysées les notifications entrantes et que les actions correspondantes sont effectuées. Cette méthode doit être redéfinie par les sous classes d'Agent afin de la spécialiser de façon appropriée.
- **protected final void sendTo(AgentId id, Notification not)** : Permet d'envoyer la notification not à l'agent id. Pour assurer la propriété de persistance, tous les objets java associés aux agents doivent être Serializable.

1.2.2. Les notifications

Une notification est un objet java Serializable. Toutes les notifications doivent étendre la classe Notification.java. Pour pouvoir émettre des notifications externes à destination d'agents (internes), comme on ne se trouve pas dans une réaction on ne dispose pas de sendTo, on s'adresse directement au composant "Channel" du bus à messages dont la mission est l'acheminement des dits messages en utilisant le code suivant : Channel.sendTo(*AgentId*, *Notification*);

1.2.3. Les serveur d'agents

Tous les agents s'exécutent dans des serveurs d'agents. Un serveur d'agents est un unique flot d'exécution qui gère un ensemble d'agents. Plusieurs serveurs peuvent résider sur la même machine. Tous les serveurs d'agents sont interconnectés et peuvent communiquer entre eux. Chaque serveur possède un thread d'exécution en charge d'activer la réaction d'un agent en fonction des notifications reçues. Cela implique qu'une seule réaction d'agent s'exécute à la fois au sein d'un serveur d'agent. Chaque serveur d'agents met en œuvre une part du bus logiciel.

1.2.4. Fichier de configuration

Dans la version actuelle du bus à agents, tous les serveurs d'agents sont connus statiquement et doivent être enregistrés dans un fichier de configuration "a3servers.xml". Ce fichier de configuration doit être disponible sur tous les sites exécutant un serveur d'agent. A ce fichier au format "xml" est associé un fichier décrivant sa structure "a3config.dtd". Voici un exemple de fichier a3servers.xml qui définit deux serveurs d'agents (0 et 1) qui s'exécutent sur la même machine.

[1] Une contrainte inexpliquée

```
<?xml version="1.0"?>
<!DOCTYPE config SYSTEM "a3config.dtd">
<config>
    <domain name="D1" />
    <server id="0" name="s0" hostname="localhost">
      <network domain="D1" port="11731"/>
    </server>
    <server id="1" name="s1" hostname="localhost">
          <network domain="D1" port="11732"/>
    </server>
</config>
```

1.3.5. Lancement d'un serveur d'agent

Java fr.dyade.aaa.agent.AgentServer num rac

// num : numéro du serveur d'agent

// rac : racine de persistance du serveur d'agent, c'est un répertoire contenant les images persistantes des agents et notifications gérés par le serveur.

// Ex : java fr.dyade.aaa.agent.AgentServer 1 s1

Il est possible de lancer un serveur d'agents par programme de la manière suivante :

```
  ...
 main () {
   ...
   try {
    AgentServer.init(args); // initialisation du serveur
    /*
    * création des objets localement, réalisation des initialisations
    * nécessaires et déploiement de ceux-ci sur les serveurs qui leur ont
    * été respectivement affectés.
    */
    AgentServer.start(); // lancement du serveur
   }catch(Exception e) {
    e.printStackTrace(System.out);
   }
  }
  ...
```

Le programme principal de votre application doit utiliser la seconde manière afin de pouvoir déployer les agents sur les différents serveurs et accéder au bus à messages pour émettre des notifications externes. Les arguments fournis à "init" sont les mêmes que ceux devant être fournis à "AgentsServer" : le numéro du serveur et sa racine de persistance. Dans tous les cas votre classpath doit contenir l'accès aux librairies : a3.jar et xerces.jar.

1.3. L'infrastructure d'exécution A3

1.3.1. Une vue d'ensemble

L'infrastructure A^3 est basée sur un bus à Message qui a pour rôle d'acheminer les notifications et de provoquer la réaction de l'agent destinataire. Celui-ci est mis en œuvre de façon distribuée, ce qui implique qu'il existe sur chaque site une entité le représentant. Lors de l'émission d'une notification, le bus de la JVM d'émission encapsule la notification dans un message et l'achemine de manière fiable grâce à un système de queues persistantes. A l'arrivée, le message est stocké dans une queue de notification entrante. Le bus de la JVM du destinataire prend le premier message de la queue et appelle la réaction de l'agent destinataire.
Le schéma ci-dessous montre l'organisation :

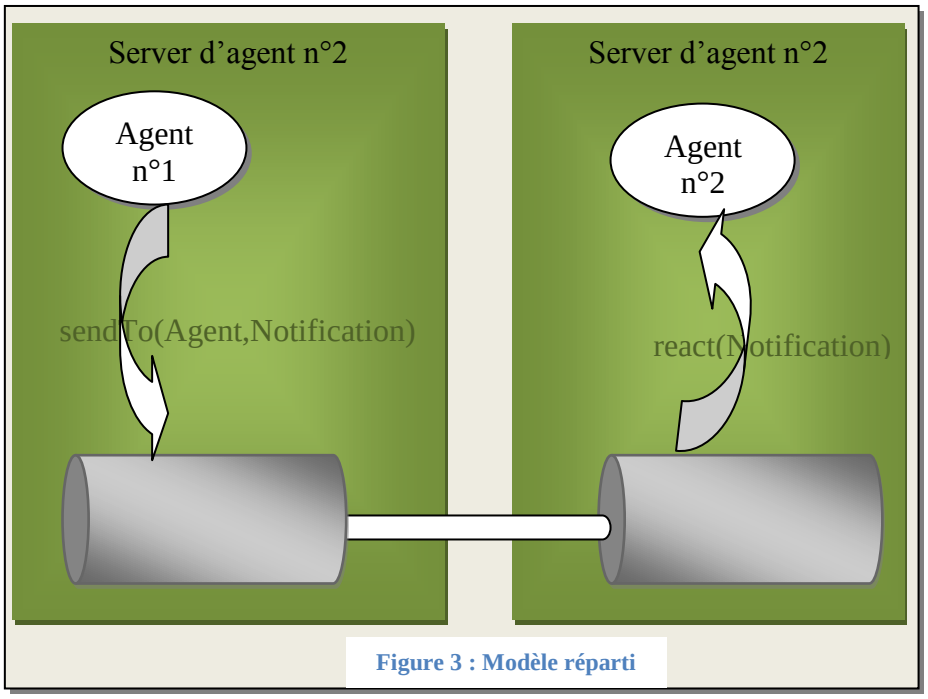

Figure 3 : Modèle réparti

1.3.2. Description détaillée du bus à Message

Le schéma ci-dessous montre le détail de l'organisation d'un composant réparti du bus à message.

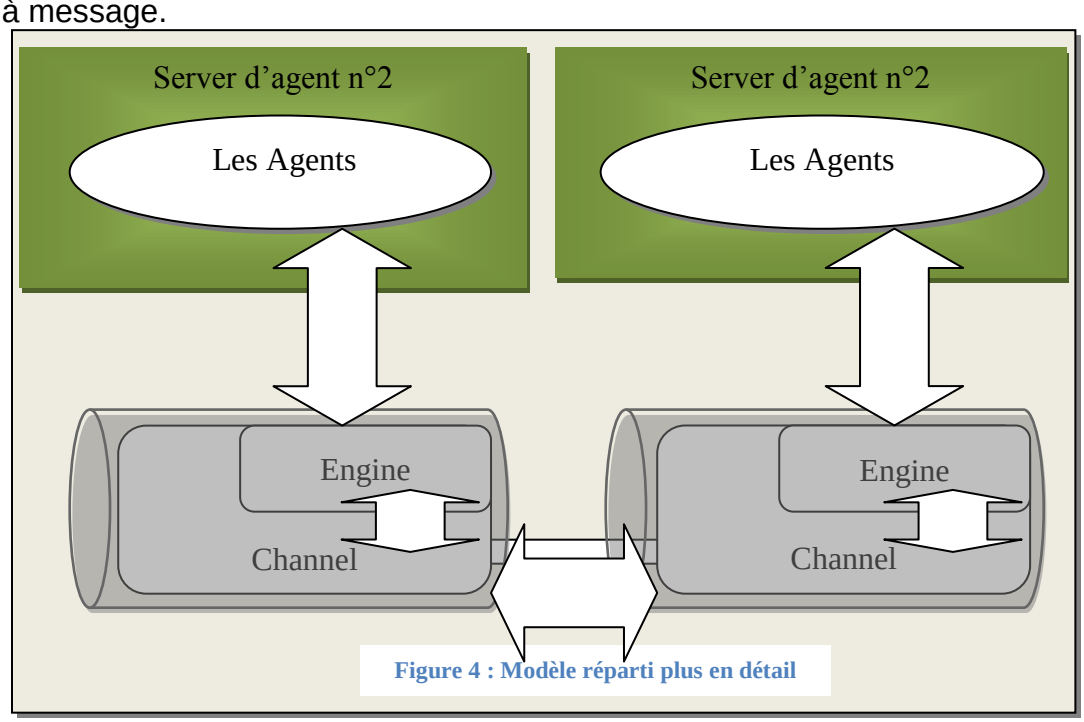

Figure 4 : Modèle réparti plus en détail

2. Un exemple

L'exemple suivant a pour but de montrer les principes de programmation des agents A^3. Dans cet exemple, deux agents vont être créés et déployés :

- Un agent HelloWorld qui doit réagir à une notification de type HelloWorldNot en affichant le message " helloworld " à l'écran.
- Un agent HelloWorldClient qui réagit à une notification de type StartNot en envoyant une notification HelloWorldNot à un agent helloworld.

2.1. Les Classes de l'application

2.1.1. L'agent HelloWorld.java

```
package work.TP.IS.tp1.v1;

import java.io.*;
import fr.dyade.aaa.agent.*;

public class HelloWorld extends Agent {
   public HelloWorld(short to) {super(to);}
   public void react(AgentId from, Notification n) throws Exception {
      …
   }
}
```

2.1.2. La notification HelloWorldNot.java

```
package work.TP.IS.tp1.v1;
import fr.dyade.aaa.agent.*;
public class HelloWorldNot extends Notification {
   …
}
```

2.1.3. L'agent HelloWorldClient.java

```
package work.TP.IS.tp1.v1;
import java.io.*;
import fr.dyade.aaa.agent.*;
public class HelloWorldClient extends Agent {
   public AgentId dest;
   public HelloWorldClient(short to) {super(to);}
   public void react(AgentId from, Notification n) throws Exception {
      …
   }
}
```

2.1.4. La notification StartNot.java

```
package work.TP.IS.tp1.v1;
import fr.dyade.aaa.agent.*;
public class StartNot extends Notification {
   …
}
```

2.2. Travail à réaliser

2.2.1. Réaliser le programme principal

Ecrivez le programme principal "Launch" qui permet de déployer dans 2 serveurs différents 2 agents ag0 et ag1 respectivement de type HelloWorldClient et HelloWorld de telle sorte que ag0 envoie ses notifications à ag1.

2.2.2. 2.2. Lancer l'exemple

Il s'agit de lancer l'exemple décrit plus haut pour vérifier la validité de l'installation de l'environnement A^3. La chaîne Hello World doit s'afficher à l'écran. Pour cela :

- Lancer le premier serveur d'agent : java fr.dyade.aaa.agent.AgentServer 1 s1
- Lancer le programme principal : java work.TP.IS.tp1.v1.Launch 0 s0

NB : Les serveurs d'agents doivent avoir accès au fichier de config a3server.xml et à a3config.dtd. Le classpath doit contenir le chemin vers les fichiers jar nécessaires (a3.jar et xerces.jar) ainsi que vers le package "work.tp.is.tp1.v1". Lors de chaque exécution veillez à effacer les racines de persistance des serveurs d'agents et les fichiers d'audit si nécessaire !

2.2.3. Vérification des propriétés

Ecrivez des programmes et des scénarios permettant de vérifier les propriétés des agents A^3 :

- Asynchronisme des communications (mode déconnecté)
- Persistance des agents
- Atomicité de la réaction
- Fiabilité des communications
- Ordonnancement causal des notifications

NB : Pour plus de clarté, vous pourrez faire des versions différentes dans des sous-packages différents pour chaque cas à traiter.