MAGIQUE

UN TUTORIEL

Équipe SMAC
http://www.cristal.univ-lille.fr/SMAC/

Version 0.3.0.1
le 29/04/2016

UNIVERSITE LILLE 1
U.F.R. d'I.E.E.A. Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX

Contents

AVERTISSEMENT

Ce document est un premier brouillon de ce qui deviendra un tutoriel pour MAGIQUE ce qui explique que certains passages doivent être fortement remaniés (le chapitre “Introduction” par exemple).

Il a cependant le mérite d'exister et de faciliter une première mise en main de MAGIQUE en en présentant les principaux concepts.

Vos remarques seront les bienvenues, elles nous permettront de l'améliorer d'autant plus efficacement. Envoyez les à :

XXX

MAGIQUE et quelques petits exemples sont récupérables à l'adresse :

`http://www.cristal.univ-lille.fr/MAGIQUE/`

Prérequis On supposera que le lecteur est raisonnablement familier avec le langage JAVA, sans qu'il ait besoin d'en être un spécialiste.

Il est fortement conseillé d'examiner la documentation de l'API en même temps que l'on consulte ce document, afin de pouvoir étudier les détails (la syntaxe précise par exemple) de chaque méthode ou classe qui ne seront pas nécessairement décrits ici.

Chapter 1

Introduction

- Qu'est-ce qu'un agent ?
  Un *agent* est une entité logicielle autonome et réactive. Il possède ses propres connaissances et son propre savoir-faire qu'il met au service des autres. Il a la capacité de communiquer que ce soit avec d'autres agents ou un opérateur humain.

- Qu'est-ce qu'un système multi-agents ?
  Si la notion d'agent est discutable et discutée, celle de *système multi-agents* est beaucoup plus claire : un système multi-agents (ou SMA) est un ensemble d'agents logiciels ou humains qui communiquent entre eux et travaillent ensemble pour résoudre un objectif commun. Travailler ensemble signifie aussi bien collaborer que négocier ou rivaliser selon le problème.

- MAGIQUE est une API[1] JAVA qui permet de développer facilement des agents. Selon les cas les agents peuvent tourner sur une même machine ou sur des machines physiquement distribuées sur un réseau. MAGIQUE est un environnement de développement pour SMA mais n'est pas un SMA lui-même. MAGIQUE permet de masquer les aspect techniques de communication entre agents, le multi-threading et la gestion des flux. MAGIQUE banalise la communication entre agents en permettant un appel "à la cantonade". Avec MAGIQUE, inutile de savoir qui possède une compétence, il suffit de savoir qu'elle existe. Cela permet une grande réutilisabilité des agents indépendamment du contexte.

- MAGIQUE permet *a priori* de ne faire communiquer que des agents issus de MAGIQUE.

L'API MAGIQUE est fournie sous forme d'un package nommé `magique.jar`. Il est nécessaire que ce package soit placé dans le `CLASSPATH` de votre système. Imaginons que vous ayez créé à la racine de votre système un sous-répertoire nommé `smac`, il vous faudra entrer la commande :

- Sous Unix
  ```
  setenv CLASSPATH=${CLASSPATH};/smac/magique.jar%$
  ```

- Sous Windows
  ```
  set CLASSPATH=%CLASSPATH%;\smac\magique.jar
  ```

Dans MAGIQUE les agents sont définis au sein d'une plate-forme d'accueil, la philosophie de la chose est d'avoir une plate-forme par machine (et donc tous les agents dans la même JVM[2]), même si cela n'est pas une contrainte imposée. Schématiquement, une application MAGIQUE peut fonctionner selon deux modes d'exécution différents :

- Centralisée
  Dans ce cas, tous les agents lancés tournent dans la même plate-forme.

[1] Application Programming Interface
[2] JAVA Virtual Machine

- Distribuée
  Dans ce cas, des agents tournent sur des machines différentes. Il y a alors une plate-forme sur chaque machine et les agents distants communiquent entre eux via leurs plates-formes respectives (grâce au mécanisme RMI[3]).

Bien sûr, le mélange des genres est possible. On peut définir une application dans laquelle certains groupes d'agents sont sur la même machine (donc a priori dans la même plate-forme) et d'autres agents sont sur des machines différentes.

Puisque la description d'un agent MAGIQUE se trouve dans une API, il est nécessaire, dès que l'on utilise l'une des classes de cette API, de déclarer dans le programme correspondant l'accès à cette classe. Tout code JAVA décrivant un agent MAGIQUE contiendra au moins la ligne `import fr.lifl.magique.*;` Nous verrons par la suite qu'il existe d'autres paquetages dans l'API qui contiennent notamment des utilitaires.

1.1 Survol et principaux concepts

MAGIQUE signifie "Multi-AGent hiérarchIQUE" (cf. `http://www.lifl.fr/SMAC`). MAGIQUE est à la base un modèle d'organisation d'agents qui propose une *organisation* hiérarchique. Cette structure permet de proposer un mécanisme de délégation de compétences entre agents, facilitant ainsi le développement.

Dans MAGIQUE, un agent est une entité possédant un certain nombre de compétences. Ces compétences permettent à un agent de tenir un *rôle* dans une application multi-agents. Les compétences d'un agent peuvent évoluer dynamiquement (par échanges entre agents) au cours de l'existence de celui-ci, ce qui implique que les *rôles* qu'il peut jouer (et donc son statut) peuvent également évoluer au sein du SMA. Un agent est construit dynamiquement à partir d'un agent élémentaire "vide", par enrichissement/acquisition de ses compétences.

Du point de vue de l'implémentation, une compétence peut être perçue comme un composant logiciel regroupant un ensemble cohérent de fonctionnalités. Les compétences peuvent donc être développées indépendamment de tout agent et donc réutilisées dans différents contextes. Une fois ces compétences créées, la construction d'un agent MAGIQUE se fait très simplement par un simple mécanisme d'enrichissement de l'agent à construire avec ces compétences. L'ajout de nouvelles compétences (et donc fonctionnalités) dans une application SMA est donc facile.

Concrètement, MAGIQUE existe sous la forme d'une API JAVA permettant le développement de systèmes multi-agents (SMA) hiérarchiques. Dans ce cas, MAGIQUE correspond à un *framework* générique pour le développement de SMA. Le concepteur de SMA peut donc s'appuyer sur les fonctionnalités offertes par MAGIQUE telles que la communication entre agents, la distribution de l'application, l'évolution dynamique, etc. Il a la charge de développer les *compétences* applicatives dont il a besoin.

Pour toute personne connaissant le langage JAVA, l'apprentissage de MAGIQUE est très simple et l'utilisation de l'API permet de mettre en œuvre très rapidement et facilement des applications SMA distribuées. En outre, le mécanisme de délégation et la modularité induite par les compétences facilitent la réutilisabilité des agents et de leurs compétences dans différents contextes ou applications.

Enfin un environnement graphique permet la création des agents et de la structure organisationnelle du SMA qui les regroupent ainsi que le déploiement et l'exécution de ce SMA sur un réseau de machines hétérogènes.

- plate-forme (agent plate-forme)
- agent
- compétence
- hiérarchie et SMA
- requêtes

[3] Remote Method Invokation

Chapter 2

Les premiers pas

2.1 Au début était l'agent

Dans MAGIQUE, un agent est une instance de la classe `Agent`. Cela lui permet automatiquement d'avoir un agenda pour gérer ses communications, d'avoir le multi-threading pour fonctionner en parallèle avec ses semblables, de récupérer le dispositif de gestion des communications et de posséder des liens de communication vers l'extérieur.

Tout agent MAGIQUE doit être nommé. Cela se fait lors de la création de l'agent et donc dans son constructeur. Chaque agent doit contenir au moins un constructeur auquel on passe une chaîne de caractères contenant son nom. C'est par ce nom qu'il peut être identifié. Ce nom doit donc être unique dans le système constitué par l'ensemble de tous les agents, pour éviter toute ambiguïté de nommage[1].

Le constructeur de l'agent doit impérativement appeler le constructeur de la super-classe `agent` en passant cette chaîne pour que l'agent soit bien connu du système. On aura donc toujours dans le code d'une classe définissant un agent `MyAgent` quelque chose comme :

```
public MyAgent(String name) { super(name);}
```

En général, l'agent contient des méthodes[2] réactives que d'autres agents peuvent appeler et une méthode pro-active, nommée le plus souvent `action` dans MAGIQUE par convention, qui contient la description de la tâche à accomplir par l'agent. On peut bien sûr construire des agents qui n'ont que des méthodes réactives, ou, par opposition uniquement une méthode pro-active. Tout est possible.

2.1.1 Premier agent : première compétence

Afin de démarrer en douceur, commençons, comme il se doit, par créer une entité permettant d'afficher le message `"hello world"` à l'écran.

Pour cela, il nous faut d'abord définir la compétence (nous détaillerons par la suite cette notion) pro-active contenant la méthode publique "`action`" pour l'agent. Cette compétence, appelée ici `MyAgentAction`, doit hériter de la classe `MagiqueActionSkill`[3] :

[1] Pour facilité cette unicité, le nom de l'agent est en fait défini par le nom donné à sa création auquel on vient ajouter le nom (en fait l'adresse IP) de la machine qui l'héberge et le numéro de port du serveur RMI de la plate-forme - par défaut 4444. Le nom d'un agent est donc de la forme `monDonné@hote.domain.pays:4444`.

[2] le terme exact que nous devrions utiliser est "*compétences*" - cf. chapitre 3 - mais pour ne pas troubler pour l'instant le lecteur familiarisé avec la programmation objet, nous conserverons dans un premier temps, le terme *méthode*

[3] Il existe deux autres possibilités qui sont "d'implémenter" l'interface `Skill` ou d'étendre la classe `ActionSkill`, nous détaillerons ultérieurement les différences.

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class MyAgentAction extends MagiqueActionSkill {

    public MyAgentAction(Agent a) { super(a); }

    public void action() {
        System.out.println("hello world");
    }
}// MyAgentAction
```

Ensuite, il faut créer l'agent. Dans MAGIQUE un agent doit obligatoirement être attaché à une *plate-forme* d'accueil.

2.1.2 Plate-forme

La plate-forme constitue en quelque sorte le support système de MAGIQUE et permet par exemple la gestion de la communication entre agents distants. Une plate-forme est un objet de la classe `Platform` du paquetage `fr.lifl.magique.platform`.

Le principe est plutôt de n'avoir qu'une plate-forme par JVM et plus précisément par machine, celle-ci hébergeant alors tous les agents présent sur cet hôte. Cependant rien n'empêche d'en mettre en route plusieurs sur une même machine[4] mais il faut alors les distinguer en précisant au constructeur de la plate-forme le numéro du port du serveur RMI.

Un agent ne pouvant exister sans plate-forme, c'est celle-ci qui le crée. La méthode `createAgent` de `Platform` permet cela.

2.1.3 Premier programme

Nous savons donc créer un agent et disposons de sa compétence pro-active. Mais il reste à la lui faire *acquérir*, cela est réalisé grâce à la méthode `setAction`.

Il faut ensuite décider quand démarrer cette pro-activité. C'est la méthode `start` de l'agent qui permet de donner cet ordre de démarrage. Un agent qui n'aurait pas de méthode pro-active mais uniquement des méthodes réactives ne doit pas "exécuter" `start` pour fonctionner. Celle-ci sert uniquement à démarrer la méthode `action` (penser à la méthode `run` de `Thread`).

Comme pour le langage JAVA, les programmes MAGIQUE ont besoin d'un "main" pour s'exécuter. Le `main` JAVA permet la création de la JVM, le main MAGIQUE permettra la création de la plate-forme déjà mentionnée. MAGIQUE étant une surcouche de JAVA, il faudra cascader les deux "main". Ceci sera fait automatiquement, il suffit de respecter le schéma suivant :

- créer comme main MAGIQUE une classe héritant de `fr.lifl.magique.AbstractMagiqueMain`
- vous devez concrétiser la méthode abstraite `theRealMain(String[] args)`, dans le corps de cette méthode vous placez ce que vous voulez voir faire par le programme. Entre autre la création des agents (qui sera automatiquement réalisée par la plate-forme) et leur acquisition de compétences,
- créer le main JAVA qui appelle la méthode statique `go` de la classe `fr.lifl.magique.Start` avec comme argument le nom de la classe contenant le `theRealMain` MAGIQUE à exécuter. L'exécution de cette méthode créera la plate-forme et lancera la méthode `theRealMain`.

  On peut préciser comme paramètre à cette méthode, un port RMI différent du port par défaut pour la plate-forme ainsi qu'un tableau de paramètres sous forme de chaînes de caractères pour `theRealMain`

  Cette dernière étape peut être ignorée si l'on utilise le main de la classe `fr.lifl.magique.Start` avec comme argument la classe contenant

[4] Cela peut par exemple permettre de tester la distribution d'agents sans nécessairement avoir plusieurs machines sous la main.

`theRealMain` et, de manière optionnelle, le numéro de port et/ou un tableau d'arguments (comme précédemment).

Voici donc le premier programme MAGIQUE :

```
package chap2;
import fr.lifl.magique.*;

public class BasicAgentImp extends AbstractMagiqueMain {
   public void theRealMain(String args[])  {
      Agent firstAgent = createAgent("monAgent");
      firstAgent.setAction(new MyAgentAction(firstAgent));
      firstAgent.start();
   }

   // si pas d'utilisation du main de fr.lifl.magique.Start:
   public static void main(String args[])  {
      fr.lifl.magique.Start.go("chap2.BasicAgentImp");
   }

}
```

Maintenant que toutes les classes nécessaires à cette petite application sont écrites, il nous reste à effectuer la compilation puis l'exécution.

- `javac chap2/BasicAgentImp.java`
  permet d'effectuer la compilation de la classe `BasicAgentImp` ainsi que la compilation des classes nécessaires, donc de `MyAgentActionSkill`[5].
- et pour exécuter notre petit programme.
  - `java chap2.BasicAgentImp`
  - `java fr.lifl.magique.Start chap2.BasicAgentImp`
    et dans ce cas pas de `main` nécessaire dans le fichier `BasicAgentImp.java`

Il est important de noter qu'un agent n'est pas un objet. Une fois le message affiché, l'agent ne s'arrête pas a priori ! Il est persistant et attend la suite des événements. Celui-ci étant pour l'instant seul au monde, il ne lui arrivera plus grand chose, vous pouvez (pour l'instant) donc l'arrêter par un CTRL-C d'une grande élégance. D'une manière générale, notez bien que la fin de la méthode `action` ou d'une quelconque autre compétence de l'agent ne signe pas l'arrêt de mort de celui-ci.

2.2 Puis ils furent deux et communiquèrent.

Un système multi-agent commence à deux agents, et dès qu'il y en a deux, il faut leur permettre de communiquer. Il est donc nécessaire de les mettre en relation (les "connecter") et ensuite de leur permettre de s'envoyer des messages (nous diront aussi requêtes pour éviter une connotation trop objet). MAGIQUE dispose d'un certain nombre de primitives de requêtes que nous présenterons progressivement.

Le moyen le plus simple de mettre en relation deux agents est de les connecter directement entre eux.

2.2.1 Connexion directe

Quand plusieurs agents sont connectés entre eux, ils peuvent exploiter leurs méthodes (compétences !) respectives grâce à la méthode `perform()` de la classe `Agent`. Dans le cas d'une connexion directe ente deux agents, `perform` nécessite comme argument le nom de l'agent destinataire et une chaîne de caractères

[5] pour cela vous assurer que le paquetage `chap2` est dans votre `CLASSPATH`, ou bien dans la mesure ou le répertoire courant ("`.`") est la plupart du temps dans le `CLASSPATH`, placez vous dans le répertoire parent de `chap2` et compilez par `javac chap2/BasicAgentImp.java`

contenant le nom de la méthode à appeler, plus des éventuels arguments qui doivent impérativement être des objets (i.e. pas de type primitif). Les arguments peuvent être passés par l'intermédiaire d'un tableau d'objets, soit être énumérés, si il y en a moins de quatre.

Afin d'illustrer notre propos, continuons sur les exemples d'école. La première application répartie effectuée dans les livres d'initiation consiste à écrire deux entités qui communiquent pour un semblant de jeu de Ping-Pong. Un agent sait faire `ping` qui consiste à afficher le message "`ping`" à l'écran et un agent sait faire `pong` qui consiste à afficher le message "`pong`" à l'écran. À chaque fois que l'un des agents exécutent sa méthode (`ping` ou `pong`) il invoque ensuite la méthode correspondante chez son collègue, pour lui "renvoyer la balle". Notons qu'ici les agents sont purement réactifs et qu'ils n'ont pas de but en soi, donc pas de méthode `action` particulière. Par contre nous avons besoin de deux compétences : une compétence "faire ping" et une "faire pong"... On obtient donc pour la compétence `PingSkill`[6] :

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class PingSkill extends MagiqueDefaultSkill {
    public PingSkill(Agent a){ super(a); }

    public void ping() {
       System.out.println("ping");
       // requête sur la compétence pong de "agentPong"
       perform("agentPong","pong");
    }
} // PingSkill
```

et de la même manière pour la compétence `PongSkill`:

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class PongSkill extends MagiqueDefaultSkill {
    public PongSkill(Agent a){ super(a); }

    public void pong() {
       System.out.println("pong");
       // requête sur la compétence ping de "agentPing"
       perform("agentPing","ping");
    }
} // PongSkill
```

Une fois nos deux compétences écrites (les agents sont induits car il suffira d'en créer deux "creux" et de leur enseigner ces compétences), il nous reste à écrire une application MAGIQUE qui les crée et les relie entre eux pour qu'ils puissent communiquer.

Nous décrivons donc une méthode `theRealMain` qui crée les deux agents `"ping"` et `"pong"`, leur enseigne leur compétence et les connecte grâce à la méthode `connectTo`, puis lance le jeu en demandant l'exécution de la compétence `ping`. Ici les deux agents tournent donc sur la même plate-forme (et donc dans la même JVM).

[6] Cette compétence hérite de la classe `MagiqueDefaultSkill`, comme pour les compétence pro-active, nous détaillerons plus tard le autres possibilités qui sont l'"implémentation" de l'interface `Skill` et l'héritage de la classe `DefaultSkill`

```
package chap2;
import fr.lifl.magique.*;

public class TestPingPong extends AbstractMagiqueMain {
    public void theRealMain(String args[]) {
        // création des agents par la plate-forme
        Agent ping = createAgent("agentPing");
        Agent pong = createAgent("agentPong");

        // acquisition de leur compétence
        ping.addSkill(new chap2.PingSkill(ping));
        pong.addSkill(new chap2.PongSkill(pong));

        // connexion des agents (automatiquement mutuelle)
        ping.connectTo("agentPong");

        // lancement du jeu
        ping.perform("ping");
    }

} // TestPingPong
```

Maintenant que nos trois classes sont écrites nous pouvons effectuer la compilation et l'exécution de notre application Ping-Pong.

- `javac TestPingPong.java`
  Pour effectuer la compilation de la classe `TestPingPong` ainsi que la compilation des classes nécessaires (donc de `PingSkill` et `PongSkill`).

- `java fr.lifl.magique.Start chap2.TestPingPong`
  Pour exécuter notre petite application.

Comme pour la précédente application, celle-ci ne s'arrête jamais puisque pour l'instant rien n'est prévu pour arrêter l'application. Effectuez un CTRL-C du plus bel effet pour tout stopper.

Cependant, cette approche comporte certains inconvénients. On le voit dans la méthode `ping`, de la compétence `PingSkill`, il y a une invocation **explicite** d'un agent dénommé `"agentPong"` et qui doit pouvoir exploiter pour nous une compétence `pong`. Donc tout agent possédant cette compétence ne pourra fonctionner que si il est connecté à un agent nommé `superPong`... La présence d'un agent possédant la compétence `PongSkill` mais porteur d'un autre nom, ne serait d'aucune utilité pour notre agent `Ping`, bien qu'il serait tout aussi capable de répondre à la requête...

De plus, on devine bien qu'apparaît ici le problème de la réutilisabilité dans différents contextes (c'est-à-dire différentes applications multi-agents) d'agents possédant la compétence `PingSkill`. Car en fait ce qui est le plus souvent[7] important pour un agent `ping`, c'est plus qu'il existe un autre agent susceptible de répondre à sa requête d'exécution d'un `pong`, que de savoir que cet agent s'appelle nécessairement `agentPong`...

C'est pourquoi MAGIQUE propose un mécanisme basé sur l'utilisation d'un agent appelé *superviseur* qui va mettre en liaison différents agents compétents et permettre une invocation de méthode (compétence) sans nécessairement se préoccuper de qui la possède.

2.2.2 Utilisation d'un superviseur

Nous allons reprendre l'exemple précédent mais en reliant cette fois nos deux agents à un superviseur.

Le superviseur est un agent comme les autres excepté le fait qu'il a la responsabilité de la communication entre les agents qu'il gère. La définition d'un agent superviseur se fait par ralliement d'au moins un autre agent à celui-ci. Ce ralliement se fait par la méthode `connectToBoss()` de la classe `Agent`. Quand plusieurs agents sont reliés à un superviseur, ils peuvent faire des requêtes sur leurs méthodes respectives

[7] pas toujours

grâce à la méthode `perform` de la classe `Agent`. `perform`, dans sa forme la plus simple, nécessite comme argument une chaîne de caractères contenant le nom de la méthode à appeler. Vous n'avez pas à vous soucier de savoir qui va l'exécuter, c'est le superviseur qui s'en charge.

Le superviseur n'a pas besoin d'être un agent complexe pour effectuer ce travail de communication. Il en possède les capacités simplement puisque il est un agent (i.e. "instance ce de `Agent`")! Il est donc possible pour gérer les communications entre agents d'utiliser un superviseur qui n'a pas de compétence particulière (i.e. autres que celles par défaut de tout agent, comme la communication, l'interprétation de requêtes, etc.).

Les compétences `PingSkill2` et `PongSkill2` sont quasiment identiques aux précédentes `PingSkill` et `PongSkill`, à un détail près : les paramètres de la méthode `perform` où il n'est plus nécessaire de préciser l'agent destinataire. Donc pour la compétence `PingSkill2` :

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class PingSkill2 extends MagiqueDefaultSkill {
   public PingSkill2(Agent a){ super(a); }

   public void ping() {
     System.out.println("ping");
     // requête sur la compétence pong d'un agent "anonyme"
     perform("pong");
   }
} // PingSkill
```

et de la même manière pour la compétence `PongSkill2`:

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class PongSkill2 extends MagiqueDefaultSkill {
   public PongSkill2(Agent a){ super(a); }

   public void pong() {
      System.out.println("pong");
      // requête sur la compétence ping d'un agent "anonyme"
      perform("ping");
    }
} // PongSkill
```

Comme nous l'avons dit, il nous faut attacher ces agents à un superviseur qui gérera leurs communications. Ici le superviseur n'a *a priori* rien d'autre à faire que de gérer les communications, il est donc constitué d'un agent complètement vide (i.e. sans compétence particulière).

Il nous faut donc écrire une application qui crée nos agents, les place dans une plate-forme et les relie entre eux pour qu'ils puissent communiquer. Nous décrivons donc une méthode `theRealMain` qui crée les trois agents, connecte cette fois-ci les deux agents `"agentPing"` et `"agentPong"` au troisième agent qui fera office de superviseur, puis lance le jeu en demandant l'exécution de la méthode `ping()` (qui peut être "appelée" via `perform` par n'importe lequel des agents sans que cela ait de.

```
package chap2;

import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class TestPingPong2 extends AbstractMagiqueMain {

    public void theRealMain(String args[])  {
        // création des agents par la plate-forme
        Agent sup = createAgent("superviseur");
        Agent ping = createAgent("agentPing");
        Agent pong = createAgent("agentPong");

        // raccordement à la plateforme
        ping.addSkill(new chap2.PingSkill(ping));
        pong.addSkill(new chap2.PongSkill(pong));

        // connexion des agents à sup en tant que superviseur
        ping.connectToBoss("superviseur");
        pong.connectToBoss("superviseur");

        // lancement du jeu
        sup.perform("ping");
        // ou ping.perform("ping") ou pong.perform("ping")
    }

} // TestPingPong2
```

La compilation et l'exécution se déroule de la même manière que pour la version précédente (et il faut toujours un `Ctrl+C` pour l'interrompre.

Le fonctionnement est en apparence le même mais on comprend que cette fois, le nom de l'agent possédant la compétence `pong` (resp. `ping`) n'est plus important, l'important c'est qu'il existe un agent capable de faire un `pong` (i.e. compétent). Pour le vérifier, il vous suffit de changer le nom des agents dans le `main` précédent, de le recompiler et de relancer l'application. Tout se passe de la même manière... Ce qui n'est pas le cas si vous faites de même avec la première version.

On comprend bien que l'on a gagné en réutilisabilité de la compétence `PingSkill2`, dont le seul prérequis à son "bon fonctionnement" est l'existence d'un agent (éventuellement le même) sachant faire `pong`, qui qu'il soit...

Nous devons cependant expliquer comment il est possible de faire une requête à une "méthode" sans préciser qui doit l'accomplir. Cela est évidemment rendu possible par la présence du superviseur, qui crée une structure hiérarchique pour les agents et gère le routage des messages entre agents. C'est l'objet du paragraphe suivant.

2.3 Le routage des messages

Lorsqu'un agent MAGIQUE appelle une méthode par `perform`, le routage se fait de la manière suivante :

- L'agent regarde s'il connaît cette méthode. Si oui il l'exécute.
- Si non, il regarde si il a une équipe à sa charge (s'il est superviseur) dont l'un des agents possède la compétence recherchée, cet agent peut être présent à une profondeur quelconque de la hiérarchie qu'il contrôle. Si c'est le cas il transmet la requête à l'un des agents situé immédiatement en-dessous de lui et qui contrôle une sous-hiérarchie dans laquelle se trouve (au moins) un agent compétent (éventuellement cet agent lui-même). Cet agent répète alors le même processus.
- S'il n'y en a aucun, il transmet la requête à son propre superviseur qui effectue le même raisonnement ... et ainsi de suite.

- Enfin le dernier cas, s'il est lui même le big boss (la racine de la hiérarchie) et qu'il n'y a donc pas (encore) d'agent compétent dans cette hiérarchie, il met la requête en attente jusqu'à ce qu'un agent compétent rejoigne la structure ou qu'un agent déjà présent apprenne la compétence requise.

Cela amène une remarque importante quant à la conception d'applications MAGIQUE. Si, dans un `perform`, une erreur se glisse dans le nom de la méthode appelée, aucune erreur ni de compilation, ni d'exécution ne se produira ! Cette requête sera simplement mise en attente d'un nouvel arrivant ou l'acquisition d'une nouvelle compétence. Ce sont des agents, pas des objets !

Nous avons choisi ici de lancer l'application dans le `main` par l'appel à `sup.perform("ping")`. Grâce au système de routage de MAGIQUE, n'importe quel autre agent du système aurait donc aussi pu être utilisé. Cette instruction peut donc être remplacée par `pong.perform("ping")` ou `ping.perform("ping")` sans aucun problème de fonctionnement.

D'une manière identique, nous avons décidé d'utiliser un superviseur vide et de lancer l'application dans le `main`. Il est bien sûr possible de lancer l'application dans la méthode `action` du superviseur ou même de l'un des agents (cette vision correspond d'ailleurs plus à la philosophie de la chose, même si ici, il est sans doute un peu fort de parler de pro-activité). Dans ce cas, il faut effectuer un `start` de l'agent concerné dans le main pour que sa méthode `action` soit exécutée.

Enfin, nous avons choisi un superviseur vide pour relier les deux autres agents. Il est tout à fait possible de lui faire faire une action, et pourquoi pas, l'action `pong`. Le superviseur traite alors `pong` et l'autre agent traite la méthode `ping`. On passe d'une application à trois agents, à une application à deux agents. C'est au concepteur de l'application de choisir la répartition qui lui semble la plus adaptée : trois ou deux agents, superviseur vide ou pas, c'est un choix de conception, pas une contrainte de MAGIQUE. Notre application précédente pourrait donc être écrite ainsi :

d'abord la compétence pro-active du superviseur :

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class SuperPingAction extends MagiqueActionSkill {
    public SuperPingAction(Agent a) { super(a); }

    public void action() { perform("pong"); }
}
```

et pour `theRealMain` ...

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class TestPingPong3 extends AbstractMagiqueMain {
  public void theRealMain(String args[])   {

     // création des agents par la plate-forme
     Agent superPing = createAgent("superPing");
     Agent pong = createAgent("agentPong");

     // raccordement à la plateforme
     superPing.addSkill(new chap2.PingSkill2(superPing));
     superPing.setAction(new chap2.SuperPingAction(superPing));
     pong.addSkill(new chap2.PongSkill2(pong));

     // connexion de pong à superPing en tant que superviseur
     pong.connectToBoss("superPing");

     // lancement du jeu
     superPing.start();
  }
} // TestPingPong3
```

L'agent `pong` ne change pas par rapport à la version précédente.

2.4 Le système de trace

Du fait du concept de fonctionnement d'un agent qui doit être autonome et réactif, la mise à point d'une application n'est pas toujours aisée. Nous l'avons dit, une erreur dans le nom de la méthode appelée par un `perform` n'a aucune incidence ni à la compilation, ni à l'exécution. Il est donc parfois utile (pour savoir si un agent s'est bien connecté à son superviseur, pour savoir si un agent a bien envoyé un message ou pour savoir si un agent a bien reçu le message qui lui était destiné) d'avoir une trace du fonctionnement des agents.

L'API MAGIQUE contient pour cela une méthode statique de la classe `Agent` nommée `setVerboseLevel` qui a pour paramètre un entier indiquant le niveau de trace. Il existe différents niveaux utilisés par l'API de 1, quasi muet, à 5, pour le plus bavard.

Chaque `main` implémentant une application MAGIQUE peut donc inclure une instruction de la forme `Agent.setVerboseLevel(2);` pour activer la trace dans la JVM correspondante.

Une solution plus élégante consiste à autoriser un paramètre au lancement de l'agent et l'utiliser pour affecter le niveau de trace. De cette manière, sans changer le code de l'agent ni recompiler quoi que ce soit, il est possible de lancer une exécution avec le niveau de trace souhaité. Dans un tel cas, la méthode `theRealMain` contiendra une instruction de la forme :

```
if (args.length==1)
   Agent.setVerboseLevel(Integer.parseInt(args[0]));
```

L'argument du main JAVA étant passé au `theRealMain` MAGIQUE.

Remarque. Notons que pour "tracer" les problèmes dus à une mauvaise orthographe des noms de méthodes, un message apparaît dès le niveau de trace 2 indiquant une méthode inconnue dans la hiérarchie. Il est cependant important de ne pas oublier lors de l'apparition de ce message, que le statut "`unknown`" d'une requête peut évoluer lors de l'arrivée d'un nouvel agent ou l'acquisition d'une nouvelle compétence dans le SMA. Ce message ne correspond donc pas nécessairement à une erreur, il indique simplement qu'à cet instant précise, il n'y a pas d'agent compétent connu dans la hiérarchie. Il ne faut donc pas être nécessairement surpris si ce message se produit lors de la création de la hiérarchie alors que les agents ne sont pas nécessairement encore tous activés et connectés.

2.5 Enfin les agents s'éloignèrent.

Jusqu'à présent, tous les agents fonctionnaient dans la même plate-forme et donc sur la même machine hôte. Dans de nombreuses applications multi-agents, il est nécessaire de répartir les agents sur le réseau. Par exemple en plaçant un agent par machine.

Les agents étant alors distribués, il est nécessaire au moment du rattachement d'un agent à son superviseur de préciser la localisation de celui-ci. Cela se fait en précisant l'adresse IP (et le port) de la plate-forme sur laquelle on peut le trouver.

En fait la méthode `connectToBoss` que nous avons déjà utilisée, permet la connexion distante d'agents. Si nous n'en avons pas encore pris conscience, c'est parce que nous avons utilisé sans le dire certaines facilités syntaxiques permises par l'API MAGIQUE. En effet, nous avons vu que les méthodes de connexion prennent pour argument le nom de l'agent auquel on se connecte. Or comme nous avons déjà eu l'occasion de le signaler rapidement, le nom réel des agents n'est pas celui fourni à la création mais celui-ci se voit étendu par le nom de l'hôte et le port de la plate-forme. Ainsi l'agent de nom `superPing` créé sur une plate-forme de port `4444` (valeur par défaut) sur un hôte appelé `host.domain.fr` par exemple a pour nom réel : `"superPing@host.domain.fr:4444"` (on peut avoir le numéro IP à la place du nom d'hôte - "*hostname*"). L'utilisation du nom court (le nom donné à la création) n'est qu'une facilité exploitable lorsque les agents sont sur la même plate-forme. D'ailleurs d'un point de vue "propreté" de conception, il faudrait en fait utiliser systématiquement le nom complet (le vrai nom quoi).

La connexion à un superviseur ressemble donc à :

```
connectToBoss("nomSuperviseur@machine.reseau.pays:port");
```

À partir de ce nom passé en paramètre, MAGIQUE peut donc facilement localiser l'agent et ainsi réaliser la connexion même distante. Donc pour conclure : qu'un agent veuille prendre contact avec un agent local ou un agent distant, cela est transparent, il procède exactement de la même manière.

Il nous est donc maintenant possible de distribuer notre Ping-Pong.

Les compétences `PingSkill` et `PongSkill` ne changent pas. Si nous voulons lancer chacun de ces agents sur une machine différente, il faut bien sûr un `theRealMain` différent pour chacun d'entre eux.

```
package chap2;
import fr.lifl.magique.*;

public class SuperImp extends AbstractMagiqueMain {
      //args[0] = verbose level
    public  void theRealMain(String[] args) {
        if (args.length==1)
            Agent.setVerboseLevel(Integer.parseInt(args[0]));

        Agent a = createAgent("super");
        a.perform("ping");
    }
}
```

Le `main` pour chacun des deux autres agents décrit la connexion au superviseur en utilisant la notation longue.

```
package chap2;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class PongImp extends AbstractMagiqueMain {
    // args[0]=super hostName:port + args[0] = verbose level
    public void theRealMain(String[] args) {
        if (args.length==2)
            Agent.setVerboseLevel(Integer.parseInt(args[1]));

        Agent a = createAgent("pong");
        a.addSkill(new PongSkill2(a));
        a.connectToBoss("super@"+args[0]);
    }
}
```

Remarque. Si vous voulez pouvoir faire tourner ces programmes sur une seule machine, il est nécessaire de préciser des ports différents pour le serveur RMI.

Il reste à compiler ces trois implémentations de nos agents et les lancer sur trois machines (ou trois JVM pour tester sur une seule machine) différentes pour apprécier cette magnifique partie de Ping-Pong.

- `javac SuperImp.java` pour compiler le superviseur et son implémentation.
- `javac PingImp.java` et `javac PongImp.java` pour compiler nos deux agents réactifs et leurs implémentations respectives.
- `java fr.lifl.magique.Start chap2.SuperImp.`.

  (ou `java fr.lifl.magique.Start chap2.SuperImp 4444 3` pour avoir un niveau de verbose de 3 (passé comme premier argument à `theRealMain`, ici comme nous l'avons précisé, le numéro de port (4444) doit être indiqué car l'argument (3) est un entier)
- `java fr.lifl.magique.Start chap2.PingImp superHost:4444` sur une seconde machine différente du réseau avec `superHost:4444` qui est le nom IP de la machine sur laquelle a été lancé `SuperImp` suivi du port de la plate-forme

  (ou `java fr.lifl.magique.Start chap2.PingImp 5555 superHost:4444` sur la même machine mais dans un shell séparé pour les tests, on peut également préciser un numéro de verbose)

  (ou encore `java fr.lifl.magique.Start chap2.PingImp superHost:4444 3` pour un niveau de verbose de 3)
- `java fr.lifl.magique.Start chap2.PongImp superHost:4444` sur une troisième machine différente du réseau

  (ou `java fr.lifl.magique.Start chap2.PongImp 6666 superHost:4444` sur la même machine mais dans un shell séparé pour les tests)

2.6 Les erreurs classiques

Arrivé à ce niveau d'utilisation de Magique, vous allez être confrontés à vos premières erreurs d'écriture. Les erreurs peuvent avoir différentes origines :

- Les erreurs JAVA.
  En général, le compilateur JAVA fait bien son travail. Une ou plusieurs erreurs apparaissent à la compilation et elles sont normalement suffisamment explicites pour vous guider vers la solution.
  - Le classpath n'est pas correct
    `Package fr.lifl.magique not found in import`

- J'ai oublié le `import` magique dans mon fichier
  `Undefined variable or class name: Agent`
- Un type primitif (`int`,`char`,`double`) est utilisé dans une des méthodes de communication MAGIQUE. Rappelons que seuls des objets sérialisables peuvent être paramètre de requêtes.

- Les erreurs de logique et de conception de l'application. À ce stade, nous ne pouvons plus rien pour vous. L'erreur dépend de votre application, de ce que vous souhaitez obtenir, du choix de décomposition des agents que vous avez choisi, bref, vous êtes le seul à savoir.

- Les erreurs d'utilisation de l'API.

  Ces erreurs sont plus difficiles à détecter car aucun message d'erreur n'est affiché.

  - Une classe de compétence doit toujours être déclarée avec l'attribut `public`. Si ce n'est pas le cas, les autres agents ne peuvent pas accéder aux méthodes de cette compétences. Un niveau 3 de `verbose` vous permet de voir que le message est bien parti d'un agent mais n'a jamais été traité par le destinataire.
  - Le nom d'une méthode utilisé dans un `ask` ou un `perform` est erroné. Rappelons la philosophie MAGIQUE : un agent est persistant et il apprend tout au long de sa vie. Si il ne sait pas traiter une méthode aujourd'hui, peut-être en sera t-il capable demain. Il n'y a donc aucune raison de provoquer une erreur quand un nom de méthode inconnu survient (contrairement à la programmation objets). L'agent MAGIQUE stocke cette requête en attendant des jours plus heureux. Si un jour, cette méthode devient connue de l'agent, elle sera exécutée. Un niveau 2 de `verbose` vous permet de voir qu'une requête est inconnue dans le SMA à l'instant où elle est traitée et qu'elle est donc mise en attente. Attention cela ne correspond pas nécessairement à une erreur.
  - Une erreur dans le nom de la machine utilisé dans le `connectToBoss`. En exécution distribuée le nom de machine utilisé dans le `connectToBoss` doit évidemment correspondre à la machine sur laquelle le superviseur associé a effectivement été lancé.

    Il est plus conseiller de passer le nom de machine des `connectTo...` en paramètre des "`theRealMain`" afin de faciliter le changement de machine support.

    Un niveau 5 de `verbose` vous permet de voir que le message est bien parti d'un agent mais n'a jamais été traité par le destinataire.

Chapter 3

La notion de service

L'approche agent réclame qu'un agent doit pouvoir changer de comportement et donc modifier dynamiquement l'ensemble de ses compétences (en en rajoutant ou en enlevant par exemple). C'est pourquoi MAGIQUE contient la notion de service ou de compétence.

Une compétence est un ensemble de fonctionnalités qui peuvent être exploitées par un agent. Nous voulons que tout soit exprimable en terme de compétences, même la manière de gérer les compétences elles-mêmes.

D'un point de vue plus pragmatique, une compétence devrait être vue comme un composant dont l'interface publique désigne les capacités qu'un agent peut exploiter.

La granularité et le degré de complexité d'une compétence ne peuvent pas être définitivement énoncés.

La capacité d'analyser un message XML ou la capacité d'additionner deux entiers peuvent chacune représenter une compétence bien que leurs complexités seraient très probablement considérées comme se situant à des niveaux différents.

De plus, savoir si il faut grouper dans une seule compétence les quatre opérations arithmétiques de base (addition, soustraction, multiplication et division) ou les séparer en quatre compétences ne peut pas être clairement établi. Cependant, il devrait être possible d'avoir un assentiment général sur le fait que les capacités d'analyse XML et l'addition doivent se situer dans des compétences différentes. Une compétence doit en effet représenter un ensemble *cohérent* de capacités.

Convenir qu'à une compétence doit correspondre une et une seule capacité (ou réciproquement) pourrait sembler raisonnable, mais la réponse n'est pas aussi simple, et dans tous les cas cela risque de ne pas résister à la réalité des programmeurs... Les problèmes qui apparaissent ici sont les mêmes que ceux habituellement (et universellement) rencontrés en génie logiciel, et en conception objet en particulier, concernant la décomposition en objets.

Un service doit pouvoir être ajouté ou retiré **dynamiquement** à un agent[1]. MAGIQUE contient à cet effet les méthodes `addSkill()` et `removeSkill()`. Un agent peut bien sûr contenir plusieurs services ou même n'en contenir aucun. Une classe de service doit tout d'abord implémenter (directement ou par héritage) l'interface `Skill` définie dans le package `fr.lifl.magique.Skill`. Une classe de service est une classe "normale" qui possède donc des attributs et des méthodes. Ce sont toutes (et uniquement celles-là) les méthodes publiques de cette classe qui pourront être exploitées par un agent qui possédera cette compétence. Posséder une compétence c'est "apprendre" une instance de la classe définissant la compétence.

La philosophie de conception préconisée par MAGIQUE consiste à démarrer la construction d'une application à partir d'agents déjà existants (éventuellement vides) et de leur adjoindre les services nécessaires au travail. Le concepteur construira peu à peu un ensemble de services et un ensemble d'agents de base implémentant certains d'entre eux.

À titre d'exemple, reprenons notre Ping-Pong mais cette fois-ci avec la notion de service. Il y a dans cette application deux services : savoir faire `ping` et savoir faire `pong`. Nous ne détaillons que le premier, le second étant sensiblement identique.

[1] Les aspects dynamiques de MAGIQUE seront traités plus en détail ultérieurement.

Quelques précisions sont nécessaires avant de développer notre premier service. Si le service ne fait référence à aucun agent, le service implémente l'interface `Skill`. Si par contre, comme c'est le cas ici, il est nécessaire de faire référence à une méthode MAGIQUE (ici `perform`) il faut que l'agent puisse récupérer un pointeur sur l'agent qui l'implémentera. L'API contient à cet effet une classe particulière nommée `MagiqueDefaultSkill` qui s'occupe de faire le lien avec l'agent qui implémente le service. Cela se fait simplement en ajoutant au service un attribut `myAgent` qui contiendra une référence à l'agent qui contiendra le service. Cet attribut sera affecté par le constructeur du service auquel on passera le nom de l'agent au moment de la création.

```
package chap3;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class PingSkill extends MagiqueDefaultSkill {
    public PingSkill(Agent a){ super(a); }

    public void ping() {
        System.out.println("ping");
        // requête sur la compétence pong d'un agent "anonyme"
        perform("pong");
    }
} // PingSkill
```

L'implémentation de l'agent `Ping` peut par exemple se faire à partir d'un agent de base :

```
package chap3;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class PingImp extends AbstractMagiqueMain {
   // args[0] = super host name:port + args[1] = verbose level
   public void theRealMain(String[] args)   {
      if (args.length==2)
         Agent.setVerboseLevel(new Integer(args[1]).intValue());

      Agent a = createAgent("ping");
      a.addSkill(new PingSkill(a));

      a.connectToBoss("super@"+args[0]);
   }
}
```

L'agent `Pong` doit subir une modification identique : il faut créer un service `PongSkill` puis créer un agent qui apprendra ce service. Le superviseur par contre ne change pas. Il est bien sûr toujours possible de lancer l'ensemble des agents dans la même JVM comme dans le chapitre précédent.

Chapter 4

Différentes méthodes de communication

Jusqu'à présent nous n'avons utilisé qu'une seule méthode de communication entre agents. Elle est rudimentaire puisqu'*a priori* elle ne possède pas d'argument et elle ne renvoie pas de résultat. Bien sûr, l'API MAGIQUE propose d'autres méthodes de communication. Tout d'abord, commençons par les paramètres.

4.1 Le passage de paramètres

Toute méthode d'un agent qui est accessible de l'extérieur de l'agent doit impérativement n'utiliser que des objets sérialisables. Il est donc impossible d'utiliser des `int`, `char` ou autres `double`. Tout doit être encapsulé dans un objet sérialisable pour passer sans problème dans les flux de communication JAVA. Il en va de même si la méthode possède une valeur de retour.

L'appel à partir d'un agent d'une méthode avec paramètre se fait par une version polymorphe de la méthode `perform` qui contient comme nouveau paramètre un tableau d'objets contenant les paramètres à passer à la méthode :

```
perform(String,object[])
```

La syntaxe usuelle en MAGIQUE est donc :

```
Object[] params = { .... } // liste des paramètres
perform("nom_de_methode",params);
```

ou

```
perform("nom_de_methode",new  Object[] { .... });
```

Néanmoins, afin de faciliter le travail du concepteur, l'API MAGIQUE contient des définitions polymorphes de toutes les méthodes de communication jusqu'à 4 arguments. En résumé si vous avez au plus 4 arguments, vous pouvez directement faire `perform("method",arg1,arg2, ...)`. Par contre à partir de cinq arguments le tableau d'objets est nécessaire et le premier argument dans un `perform` où le destinataire n'est pas nommé ne peut pas être de type `String`.

La méthode `perform` est utilisée pour les méthodes qui ne renvoie pas de méthode. Pour les méthodes qui renvoient un résultat, deux autres méthodes de communication existent :

- `askNow()`; appel de méthode synchrone avec résultat. Quand cette méthode est appelée, l'exécution de l'agent se fige jusqu'à obtention du résultat.
- `ask()`; appel de méthode asynchrone avec résultat. Après l'appel de cette méthode, l'agent continue son exécution. Si quelqu'un répond à cette requête, la réponse est rangée dans un tableau de réponses propre à chaque agent. C'est l'agent lui-même qui doit explicitement aller chercher la réponse.

Pour chacune de ces deux méthodes, le traitement des paramètres est identique au `perform`; il est nécessaire d'utiliser un tableau d'objets pour passer les paramètres.

Pour illustrer notre propos, nous allons modifier PingPong. Jusqu'à présent notre PingPong ne s'arrêtait jamais. Nous souhaitons maintenant qu'il s'arrête au bout de 10 messages. Il existe plusieurs moyens pour stopper cette partie. On peut par exemple ajouter dans l'un des deux agents `ping` ou `pong` un compteur incrémenté à chaque message écrit. On s'arrange ensuite pour que quand ce compteur arrive à 10, l'agent n'exécute plus le `perform`, ce qui stoppe les messages respectifs.

Une autre solution, plus "agent" consiste à considérer le compteur comme un agent à part entière. Les agents `ping` et `pong` font alors appel à `compteur` pour savoir s'ils continuent ou pas leurs messages. Bien sûr, la demande d'autorisation au compteur doit être bloquante, on utilisera donc pour cela la méthode `askNow`. Pour montrer le passage de paramètres on considérera de plus que chaque agent incrémente d'un certain pas une valeur affichée avec chaque message et envoie cette valeur à son congénère qui fait ensuite de même.

Voici la compétence compteur :

```
package chap4;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class Compteur implements Skill { // c'est un service
    int cpt = 0;
    public void incr() { cpt++; }
    // Boolean and not boolean since everything must be object
    public Boolean arret() {
      if (cpt==10) {
        return Boolean.TRUE;
      }
      else {
        return Boolean.FALSE;
      }
    }
}
```

et l'implémentation de l'agent compteur se fait de manière maintenant classique :

```
package chap4;
import fr.lifl.magique.*;

public class CompteurImp extends AbstractMagiqueMain {
    // args[0] = super host name :port | arsg[1] = verbose level
    public void theRealMain(String[] args) {
        if (args.length==2)
            Agent.setVerboseLevel(Integer.parseInt(args[0]));

        Agent a = createAgent("compteur");
        a.addSkill(new Compteur());

        a.connectToBoss("super@"+args[0]);
    }
}
```

Les services de nos agents `Ping` et `Pong` sont alors modifiés de la manière suivante :

```
package chap4;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class PingSynchroSkill extends MagiqueDefaultSkill {

   public PingSynchroSkill(Agent a){ super(a); }

   public void ping(Integer val) {
     // asknow returns an object then cast is needed
     boolean answer = ((Boolean) askNow("arret")).booleanValue();

     if (!answer) {
        System.out.println("Ping"+val);
        perform("incr");
        // args must be object => Integer and not int
        perform("pong",new Integer(val.intValue()+1));
     }
   }
}
```

L'implémentation des nouveaux agents se fait aussi très classiquement :

```
package chap4;
import fr.lifl.magique.*;

public class PingSynchroImp extends AbstractMagiqueMain {
   //args[0] = super hostname :port |arsg[1] = verbose level
   public void theRealMain(String[] args) {
       if (args.length==2)
           Agent.setVerboseLevel(Integer.parseInt(args[1]));

       Agent a = createAgent("ping");
       a.addSkill(new PingSynchroSkill(a));
       a.connectToBoss("super@"+args[0]);
   }
}
```

L'écriture du second agent, la compilation et l'exécution ne posent maintenant plus aucune difficulté. Seule l'implémentation de notre superviseur nécessite une petite modification. Il doit maintenant appeler la méthode `ping` avec un argument.

```
package chap4;
import fr.lifl.magique.*;

public class SuperImp extends AbstractMagiqueMain {
   public void theRealMain(String[] args) {
     if (args.length==1)
        Agent.setVerboseLevel(Integer.parseInt(args[0]));

     Agent a= createAgent("super");
     // args must be Object => 10 is converted into an Integer
     a.perform("ping",new Integer(10));
   }
}
```

On notera encore une fois que cette décomposition à quatre agents est arbitraire et nous aurions aussi bien pu enseigner la compétence compteur au superviseur ou encore à `ping` ou `pong`. C'est une simple question de choix d'implémentation et de niveau de décomposition de l'application.

L'application Ping-Pong décrite jusque maintenant n'est pas synchrone. Il se peut donc que plusieurs *ping* apparaissent avant un quelconque *pong*. Afin de remédier à ce problème nous pouvons remplacer le `perform("pong",params)` par une méthode `askNow` qui assurera la synchronisation.

4.1.1 encore d'autres méthodes de communication ...

MAGIQUE contient encore bien d'autres méthodes de communication. Nous allons tenter de les résumer ici. Pour chaque méthode de communication, il existe dans MAGIQUE deux formes différentes :

- Appel non nommé
  On passe en premier argument le nom de la méthode invoquée sous forme de chaîne de caractères et dans les arguments suivants les paramètres nécessaires à cette méthode.
  Exemple : `perform("pong",params);`

- Appel nommé
  On passe en premier argument le nom de l'agent concerné puis le nom et les paramètres de la méthode invoquée comme précédemment.

  Exemple : `perform("paul","calcul",params);` ou `perform("paul@buxus.lifl.fr:4444","calcul",params);` si l'on veut être plus précis et être certain d'éviter tout confusion sur le nom au cas où deux agents sur deux machines différentes auraient le même nom de création.

Sans les détailler, les différentes méthodes d'envoi de messages en MAGIQUE sont donc :

- `ask`; appel de méthode asynchrone avec résultat :
  - `ask(String)` appel non nommé d'une méthode sans params
  - `ask(String,Object[])` appel non nommé d'une méthode avec paramètres
  - `ask(String,String)` appel nommé d'une méthode sans paramètres
  - `ask(String,String,Object[])` appel nommé d'une méthode avec paramètres
- `askNow`; appel de méthode synchrone avec résultat
- `perform`; appel de méthode sans résultat
- `broadcastToAll`; envoi d'un message à toute mon équipe (moi compris)
- `broadcastToBasis`; envoi d'un message à toutes les feuilles de mon équipe.

4.2 Messages synchrones ou asynchrones

Jusqu'à présent, les seules invocations de compétences à d'autres agents ont été effectuées soit avec `perform` qui n'attend aucune réponse, soit avec `askNow` qui bloque l'agent demandeur jusqu'à obtention de la réponse. Ce dernier type de message est appelé synchrone car il synchronise l'exécution de l'agent demandeur et de l'agent receveur à l'obtention de la réponse. MAGIQUE contient néanmoins une autre méthode de communication qui permet de poser une question à un autre agent sans attendre la réponse. C'est la méthode `ask`.

Imaginons un problème très simple. Deux agents `f` et `g` doivent chacun effectuer un calcul et renvoyer ensuite un réponse sous forme d'un entier. Un troisième agent collecte ces deux informations pour en faire la somme. Ce problème, très générique, est indépendant du calcul. Nous considérerons donc des agents qui possède un service qui renvoie un simple nombre après une pause p (symbolisant le temps calcul). Un tel service s'écrit trivialement de la manière suivante :

```
package chap4;
import fr.lifl.magique.*;

public class CalculTrivial implements Skill {

    // renvoie n après p millisec
    public Integer calcul(Integer n, Integer p) {
        System.out.println("debut de calcul");
        try { Thread.sleep(p.intValue()); }
          catch (Exception e){}
        System.out.println("fin de calcul");
        return n;
    }
} // CalculTrivial
```

4.2.1 Traitement synchrone

Pour bien fixer les choses, montrons tout d'abord comment l'ensemble fonctionne si on utilise classiquement `askNow` pour obtenir un traitement synchrone. L'agent collecteur des deux résultats de `f` et `g` doit appeler la méthode `calcul` de ces deux agents puis afficher le résultat. Ce service s'écrit donc :

```
package chap4;
import java.util.*;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class CalculCompletSynchrone
             extends MagiqueDefaultSkill {

    public CalculCompletSynchrone(Agent a) { super(a); }

    public void complet() {
        System.out.println("debut de complet");
        long deb = (new Date()).getTime();
        int cumul = 0;
        cumul+= ((Integer) askNow("f","calcul",
                                  new Integer(15),
                                  new Integer(15000))
                 ).intValue();
        cumul+= ((Integer) askNow("g","calcul",
                                  new Integer(10),
                                  new Integer(10000))
                 ).intValue();
        System.out.println("Cumul = " + cumul);
        long fin = (new Date()).getTime();
        System.out.println((fin - deb) + " ms");
    }
} // CalculComplet
```

On constate que ce service demande à l'agent `f` d'effectuer son calcul pendant 15s et renvoyer 15 tandis qu'il demande à l'agent `g` d'effectuer son calcul pendant 10s et renvoyer 10.

L'implémentation de ces différents agents est maintenant classique. Nous détaillons l'implémentation de l'agent `f` (`g` étant quasiment identique), l'implémentation du superviseur et l'implémentation d'un agent `lanceur` pour démarrer le calcul.

```
package chap4;
import fr.lifl.magique.*;

public class FImp extends AbstractMagiqueMain {
    // args[0] = sup host name :port | arsg[1] = verbose level
    public void theRealMain(String[] args) {
        if (args.length==2)
            Agent.setVerboseLevel(Integer.parseInt(args[1]));

        Agent a = createAgent("f");

        a.addSkill(new CalculTrivial());
        a.connectToBoss("sup@"+args[0]);
    }
} // FImp
```

```
package chap4;
import fr.lifl.magique.*;

public class SuperImp2 extends AbstractMagiqueMain {
    public void  theRealMain(String[] args) {
        if (args.length==1)
            Agent.setVerboseLevel(Integer.parseInt(args[0]));

        Agent a = createAgent("sup");
        a.addSkill(new CalculCompletSynchrone(a));
    }

} // SuperImp
```

```
package chap4;
import fr.lifl.magique.*;
public class LanceurImp extends AbstractMagiqueMain {
    // args[0] = super hostname :port |arsg[1] = verbose level
    public void theRealMain(String[] args) {
        if (args.length==2)
            Agent.setVerboseLevel(Integer.parseInt(args[1]));

        Agent a = createAgent("lanceur");

        a.connectToBoss("sup@"+args[0]);
        a.perform("complet");
    }

} // LanceurImp
```

Après compilation de l'ensemble, le lancement de cette application se fait en tapant dans chacun des shells prévus à cet effet les commandes :

`java fr.lifl.magique.Start chap4.SuperImp2 4444 3`

`java fr.lifl.magique.Start chap4.FImp 5555` *SuperImphost:port*

`java fr.lifl.magique.Start chap4.GImp 6666` *SuperImphost:port*

`java fr.lifl.magique.Start chap4.LanceurImp 7777` *SuperImphost:port*

Au bout de 25s vous obtiendrez le résultat dans la fenêtre du superviseur. On constate donc que les exécutions de `f` et `g` ont été faites en série et non en parallèle. Ceci vient du fait que le `askNow` à `g` ne peut se faire que quand le `askNow` de `f` s'est terminé. L'exécution d'une méthode `askNow` est bloquante.

4.2.2 Traitement asynchrone

C'est la méthode `ask` qui permet le traitement asynchrone. Le résultat à cette requête peut être obtenu plus tard par l'utilisation des méthodes `isAnswerReceived` et `returnValue` à qui on passe la question pour laquelle on souhaite une réponse. Pour cela il faut avoir conservé une référence de la question. Un objet particulier nommé `request` est prévu à cet effet. `isAnswerReceived` renvoie un booléen indiquant si une réponse est arrivée pour la question concernée, `returnValue` renvoie la réponse obtenue pour la question concernée. Si la réponse n'est pas encore arrivée, cette méthode attend la réponse. Elle est donc bloquante.

Notre service de calcul complet devient donc

```
package chap4;
import java.util.*;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class CalculCompletAsynchrone
             extends MagiqueDefaultSkill {

    public CalculCompletAsynchrone(Agent a) { super(a); }

    public void complet() {
        System.out.println("debut de complet");
        long deb = (new Date()).getTime();
        int cumul = 0;
        Request rf=createQuestion("calcul",
                                  new Integer(10),
                                  new Integer(10000));
        Request rg=createQuestion("calcul",
                                  new Integer(15),
                                  new Integer(15000));
        ask("f",rf);
        ask("g",rg);

        cumul += ((Integer) returnValue(rf)).intValue();
        cumul += ((Integer) returnValue(rg)).intValue();

        System.out.println("Cumul = " + cumul);
        long fin = (new Date()).getTime();
        System.out.println((fin - deb) + " ms");
    }
} // CalculCompletAsynchrone
```

Cette fois-ci l'exécution (en utilisant cette fois `SuperImp3`) ne prend plus que 15s, c'est à dire le temps de la méthode qui prend le plus de temps. `f` et `g` ont cette fois été effectués en parallèle.

4.3 Test de primarité

L'un des exercices favoris en arithmétique consiste à distribuer le calcul de longues suites numériques ou de méthodes de calcul coûteuses en temps, sur des machines distantes réparties sur le net. On trouvera par exemple des agents spécialisés dans le "cassage" du code RSA 256 ou 512 (`www.distributed.net`), l'identification des grands nombres de premiers (`www.mersenne.org`) ou l'écoute d'intelligence extraterrestre (`www.seti.org`). Illustrons la manière dont cela peut se faire avec MAGIQUE.

Pour illustrer notre propos, nous allons écrire un SMA permettant de distribuer le calcul du test de primarité d'un nombre sur un ensemble d'agents répartis sur le réseau.

La première étape consiste à écrire un service de recherche d'un diviseur d'un nombre parmi les entiers compris entre une borne minimum et une borne maximum. Appelons cette méthode `cherche` et considérons

qu'elle renvoie un `Integer` contenant -1 s'il n'y a pas de diviseur entre ces bornes, et le diviseur s'il y en a un.

```
package chap4;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class ChercheDiv implements Skill {
    // recherche un diviseur de N entre bInf et bSup
    // renvoie le premier diviseur trouvé ou -1 si aucun
    public Integer cherche(Integer bInf, Integer bSup,
                           Integer n) {
        int bi=bInf.intValue();
        int bs=bSup.intValue();
        int nb=n.intValue();
        System.out.println("cherche un diviseur entre " +
                           bi + " et " + bs);
        // to be sure bi is odd
        if (bi%2==0) bi++;
        for (int i=bi; i<=bs; i+=2)
            if (nb % i == 0) {
                System.out.println("diviseur : " + i);
                return new Integer(i);
            }
        return new Integer(-1);
    }
} // ChercheDiv
```

La partie la plus délicate concerne l'agent collecteur. Celui-ci doit compter le nombre d'agents disponibles dans son équipe, découper les intervalles de nombres à tester en tranches égales pour chaque agent, lancer les requêtes en parallèle à chacun de ces agents et enfin collecter les résultats.

```
package chap4;
import fr.lifl.magique.*;
import fr.lifl.magique.util.*;
import fr.lifl.magique.skill.*;
import java.util.*;

public class TestPrimal extends MagiqueDefaultSkill {
  public TestPrimal(Agent a){super(a);}

  // renvoie True si N est premier, False sinon, pour cela
  // il divise le calcul entre tous les agents de son équipe
  public Boolean prem(Integer n) {
     int nb = n.intValue();
     // pour mémoriser les requêtes
     Vector questions = new Vector();
     int nbAg = ((Team) askNow("getMyTeam")).size();
     int taille = (int) (Math.sqrt(nb)/nbAg)+1 ;
     // boucle d'envoi  à chaque agent de l'équipe
     int tranche = 3;
     Enumeration e=((Team) askNow("getMyTeam")).getMembers();
     while (e.hasMoreElements()) {
       String agent = (String) e.nextElement();
       Request r = createQuestion("cherche",
          new Integer(tranche),
          new Integer(tranche+taille-1),
          n);
       ask(agent,r);
       tranche = tranche + taille;
       // stocke les questions pour pouvoir les récupérer
       questions.addElement(r);
     }

     // boucle de lecture des réponses qui reviennent
     boolean answered = false;
     // tant que je n'ai pas trouve de diviseur
     while (!answered && questions.size()!=0) {
       // parcourir toutes les questions
       for (int i=0; i<questions.size() && !answered ;i++) {
         Request q = (Request) questions.elementAt(i);
         // si q existe et qu'une rep vient d'arriver
         if (q!=null && isAnswerReceived(q)) {
            Integer answer = ((Integer) returnValue(q));
            answered = !answer.equals(new Integer(-1));
            questions.removeElementAt(i);
         }
       }
     }
     return (new Boolean (!answered));
  }
} // TestPrimal
```

Il ne reste plus maintenant qu'à implémenter ces deux types d'agent, ce qui se fait de manière maintenant très classique.

```
package chap4;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class ChercheurImp extends AbstractMagiqueMain{
   public void theRealMain(String[] args) {
       if (args.length!=3) {
         System.out.println("ChercheurImp takes 'nom' and "+
                            "'superviseur host:port' and "+
                            "'verbose level' as arguments");
         System.exit(1);
       }
       Agent.setVerboseLevel(Integer.parseInt(args[2]));
       Agent a = createAgent(args[0]);

       a.addSkill(new ChercheDiv());
       a.connectToBoss("collecteur@"+args[1]);
   }
} // ChercheurImp
```

```
package chap4;
import fr.lifl.magique.*;

public class CollecteurImp extends AbstractMagiqueMain {
   public  void theRealMain(String[] args) {
     if (args.length == 2) {
       Agent.setVerboseLevel(Integer.parseInt(args[1]));
     }

     Agent a = createAgent("collecteur");

     a.addSkill(new TestPrimal(a));

     // on attend que tout le monde soit arrivé !
     System.out.println("\n\nTapez <Entree> quand tous "+
                        "les chercheurs seront lances");
     try { System.in.read(); }
         catch(java.io.IOException e) {}
     System.out.println("ok, c'est parti !!");

     // test d'un nombre
     Integer nb;
     if (args.length >=1) {
         nb = new Integer(args[0]);
     } else {
         nb = new Integer(217483647);
     }
     Boolean rep = (Boolean) a.askNow("prem",nb);
     System.out.println(nb + " Premier ? : " + rep);
   }
} // CollecteurImp
```

L'exécution de ce SMA se fait très simplement. On lance l'agent `Collecteur` sur une machine et autant d'agents `Chercheur` qu'il est possible d'en mettre sur chacune des machines du réseau. Plus on place d'agents `Chercheur` plus le calcul sera rapide. Une fois les agents `Chercheur` connectés, on lance le calcul au niveau de l'agent `Collecteur` qui fournit ensuite la réponse finale.

Avec ce petit SMA vous découvrirez par exemple que les plus grands nombres premiers gérables par un `Integer` JAVA sont 2147483647, 2147483629, 2147483587, 2147483579, 2146483563 et 2147483549.

Bien sûr le code présenté ici est très sommaire. Les agents `Chercheur` ne s'arrêtent pas quand l'un d'entre eux a trouvé un diviseur. Ils ne se déconnectent pas quand l'application est terminée et ils ne peuvent

pas faire de reprise sur panne en milieu de calcul. Chacun peut facilement imaginer les améliorations possibles de ce programme mais elles sortent du cadre de ce tutoriel.

4.3.1 Un autre point de vue

L'agent `Collecteur` précédemment décrit a été réalisé en utilisant la méthode `ask` qui permet le traitement asynchrone des requêtes. Une autre possibilité consiste à remplacer chaque `ask` par deux appels `perform` entre les deux agents concernés. Le `Collecteur` appelle la méthode `cherche` de l'agent `Chercheur` et celui-ci appelle une méthode spéciale stockant la réponse chez le `Collecteur` une fois le calcul effectué. De cette manière il n'y a plus de `ask` dans l'application. Certains lecteurs préféreront le code précédent, d'autres préféreront celui-ci. Dans ce cas, c'est une affaire de goût.

Chapter 5

Synthèse : un petit exemple complet

Dans cette partie, nous allons synthétiser les notions rencontrées jusque maintenant à travers un petit exemple complet. Évidemment, l'exemple que nous allons étudier sera volontairement très rudimentaire (on pourrait même peut être contester son caractère vraiment agent). L'intérêt de sa simplicité est de permettre de montrer l'ensemble du code et de présenter ainsi les principaux concepts de programmation de SMA avec MAGIQUE. Nous allons montrer ici que, pour toute personne connaissant le langage JAVA, le surcoût de la mise en œuvre d'une conception avec MAGIQUE est quasiment nul tout en permettant une distribution facile de l'application sur un réseau.

5.1 Le problème

Il s'agit d'un calcul de vérification de la conjecture dite de Collatz ou de Syracuse ou encore le problème $3x + 1$. Le problème étudié est le suivant :

> *Considérons la fonction f qui a tout entier n associe :*
>
> $$f(n) = \begin{cases} \frac{n}{2} & \textit{si } n \textit{ est pair} \\ 3n+1 & \textit{si } n \textit{ est impair} \end{cases}$$
>
> *La conjecture affirme qu'à partir de tout n, on peut toujours atteindre 1 après suffisamment d'itérations de la fonction f :*
>
> $$\forall n \in \mathbb{N}, \exists k \in \mathbb{N}, \underbrace{f(...(f}_{k \textit{ fois}}(n)..) = 1$$

Il s'agit donc de disposer d'un SMA qui pour un entier n donné produit la suite des valeurs des itérations de la fonction jusque 1.

5.2 Les compétences

Le compétences nécessaires à la résolution de ce problème peuvent être facilement identifiées :

`ParitySkill` le test de la parité d'un entier

`MultiplierSkill` multiplication de deux entiers

`AdderSkill` addition de deux entiers

`DividerSkill` division de deux entiers

`CollatzSkill` calcul de la conjecture

Il faut donc écrire le code de chacune de ces compétences. Pour cela, il suffit de créer une classe qui hérite directement ou indirectement de l'interface `Skill` du paquetage `fr.lifl.magique.skill`. Chacune des "méthodes" publiques de cette classe pourra être alors utilisée par tout agent qui possédera cette compétence.

Pour les quatre premières compétences, rien de particulier à ajouter par rapport à un code JAVA classique. On trouve ci-dessous le code complet des compétences `ParitySkill` et `AdderSkill`. Les autres sont similaires.

```
package chap5;

public class ParitySkill implements fr.lifl.magique.Skill {
    public Boolean isEven(Integer x) {
       return new Boolean( (x.intValue()%2) == 0 );
    }
} // ParitySkill
```

```
package chap5;

public class AdderSkill implements fr.lifl.magique.Skill {
    public Integer add(Integer x, Integer y) {
       return new Integer(x.intValue()+y.intValue());
    }
} // AdderSkill
```

La compétence `CollatzSkill` nécessite un peu plus d'explications. En effet, pour accomplir sa tâche elle doit faire appel aux autres compétences. C'est ici qu'interviennent les primitives MAGIQUE d'invocation de compétences mentionnées précédemment. Dans ce cas particulier puisqu'il s'agit de calcul nécessitant que la réponse à une requête soit connue avant de poursuivre, c'est la méthode `askNow` qui doit être utilisée. Ainsi, pour invoquer la méthode `isEven` de la compétence `ParitySkill`, sur l'`Integer x` il faut écrire :

```
Boolean value = (Boolean) askNow("isEven",x);
```

On obtient pour la compétence `CollatzSkill`, le code complet est donné ci-dessous (rappel : seules les méthodes publiques sont accessibles et donc considérées comme "méthodes de la compétences").

```
package chap5;

import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class CollatzSkill extends MagiqueDefaultSkill {
   private Integer x;
   public CollatzSkill(Agent myAgent) {
     super(myAgent);
   }
   private Boolean testParity(Integer x) {
     return (Boolean) askNow("isEven",x);
   }
   private Integer xByTwo(Integer x) {
     return (Integer) askNow("quotient",x,new Integer(2));
   }
   private Integer threeTimesPlus1(Integer x) {
     Integer y = (Integer) askNow("mult",x,new Integer(3));
     return (Integer) askNow("add",y,new Integer(1));
   }
   public void conjecture(Integer x) {
     this.x = x;
     conjecture();
   }
   private void conjecture() {
     while (x.intValue() != 1) {
       perform("display",new Object[]{"x  = "+x.intValue()});
       if (testParity(x).booleanValue()) {
           x = xByTwo(x);
       }
       else {
           x = threeTimesPlus1(x);
       }
     }
     perform("display",new Object[]{"x = 1 ** finished"});
   }
} // CollatzSkill
```

On peut noter aussi l'invocation de la compétence `display`, grâce à la primitive `perform` car ici il n'y a pas de réponse attendue. Il s'agit de déléguer à la compétence qui convient (par défaut `DisplaySkill` présente dans tout agent MAGIQUE gère cette invocation) la réalisation de l'affichage demandé. Il suffit donc d'adapter cette compétence pour obtenir l'affichage pour tout format ou type de support.

5.3 La hiérarchie et les agents

Il s'agit de répartir les compétences parmi les agents du SMA puis d'organiser ces agents dans une hiérarchie. Nous allons imposer arbitrairement la structure du SMA et la répartition des compétences. Nous travaillerons ici avec sept agents : un pour chacune des compétences précédemment citées, un agent qui supervisera les quatre agents possédant les compétences arithmétiques de base et un superviseur global du SMA. D'autres choix auraient bien sûr pu être faits.

On dispose donc d'un SMA hiérarchique correspondant à la structure présentée à la figure 5.1. Rappelons que l'existence de cette structure hiérarchique permet la gestion automatique de la gestion des invocations des compétences. Ainsi lorsque la compétence `CollatzSkill` requiert la compétence `ParitySkill`, l'agent `collatz` n'a pas besoin de connaître explicitement l'agent `parity`, MAGIQUE gère l'acheminement de la requête. En fait la seule chose qui importe est de savoir que la compétence `ParitySkill` est présente dans le SMA.

Venons-en à la création de nos agents. MAGIQUE se base sur la notion de plate-forme, qui est en fait l'équivalent pour MAGIQUE de la JVM. Cette plate-forme prend en charge les problèmes liés à la distribution

Figure 5.1: la structure hiérarchique choisie

qui est ainsi masquée pour le développeur. A priori, on ne trouve qu'une plate-forme par machine même si cela n'est pas une obligation. Un agent est donc créé par une plate-forme et il faut ensuite faire son "éducation" en lui enseignant les compétences que l'on souhaite lui donner.

Le code ci-dessous présente la création de l'agent possédant la seule compétence `ParitySkill`. Cet agent doit se rattacher dans la hiérarchie à son superviseur. C'est ce qui est fait par l'appel à la méthode `connectToBoss`. Ici on suppose que ce superviseur se trouve dans la même plate-forme (ce superviseur n'a besoin d'aucune compétence particulière, il n'est là que pour chapeauter les agents "mathématiques") et qu'il doit lui même être connecté au superviseur global appelé `"super"` placé dans une autre plate-forme. Cette simple commande de connexion va suffire pour gérer les communications distantes entre les agents. On note que cette requête est la même que les agents soient distants ou pas. Il s'agit simplement de demander la création d'une relation d'accointance hiérarchique.

```
package chap5;
import fr.lifl.magique.*;

public class ParityImp extends AbstractMagiqueMain{
    // args[0] = super host name
    public void theRealMain(String[] args) {
    if (args.length == 2) {
       Agent.setVerboseLevel(Integer.parseInt(args[1]));
    }
              // création du superviseur "mathématique"
    Agent supermath = createAgent("supermath");
              // connexion à son superviseur
    supermath.connectToBoss("super@"+args[0]);
              // création de l'agent pour la parité
    Agent parity = createAgent("parity");
             // enseignement de la compétence de parité
    parity.addSkill(new chap5.ParitySkill());
             // connexion à son superviseur
    parity.connectToBoss("supermath");
  }
}
```

Pour les autres agents il en va de même. Il faut juste savoir quels agents évoluent sur la même machine et donc dans la même plate-forme.

```
package chap5;
import fr.lifl.magique.*;

//args[0] = supermath hostName
public class AdderImp extends AbstractMagiqueMain{
  public void theRealMain(String[] args) {
    if (args.length == 2) {
       Agent.setVerboseLevel(Integer.parseInt(args[1]));
    }
    Agent adder = createAgent("adder");
    adder.addSkill(new AdderSkill());
    adder.connectToBoss("supermath@"+args[0]);
  }
}
```

```
package chap5;
import fr.lifl.magique.*;
//args[0] = supermath hostName
public class MultiplierImp extends AbstractMagiqueMain {
  public void theRealMain(String[] args) {
    if (args.length == 2) {
       Agent.setVerboseLevel(Integer.parseInt(args[1]));
    }
    Agent multiplier = createAgent("multiplier");
    multiplier.addSkill(new chap5.MultiplierSkill());
    multiplier.connectToBoss("supermath@"+args[0]);
  }
}
```

```
package chap5;
import fr.lifl.magique.*;
//args[0] = supermath hostName
public class DividerImp extends AbstractMagiqueMain{
  public void theRealMain(String[] args) {
    if (args.length == 2) {
       Agent.setVerboseLevel(Integer.parseInt(args[1]));
    }
    Agent divider = createAgent("divider");
    divider.addSkill(new chap5.DividerSkill());
    divider.connectToBoss("supermath@"+args[0]);
  }
}
```

Pour que le calcul se fasse, il faut invoquer la compétence `conjecture`, cela peut être fait par exemple après la création de l'agent `collatz` (cf. code ci-dessous). Mais avec le mécanisme de délégation, cette invocation aurait pu être fait par n'importe quel autre agent.

```
package chap5;
import fr.lifl.magique.*;
// args[0] = super host name
public class CollatzImp extends AbstractMagiqueMain {
  // args[0] = super host name | args[1] = conjecture strating value
  // args[2] = verbose level
  public void theRealMain(String[] args) {
    if (args.length == 3) {
       Agent.setVerboseLevel(Integer.parseInt(args[1]));
    }
    Agent colAgent = createAgent("collatz");
    colAgent.connectToBoss("super@"+args[0]);
    colAgent.addSkill(new chap5.CollatzSkill(colAgent));
             // calcul de la conjecture avec x=17
    Integer x;
    if (args.length < 2) {
x = new Integer(17); //default
    }
    else {
x = new Integer(args[1]);
    }
    colAgent.perform("conjecture",x);
  }
}
```

Si en plus des programmes, `ParityImp`, `CollatzImp`, on dispose des autres `AdderImp`, `MultiplierImp`, `DividerImp`, `SuperImp` écrits sur le même principe (en supposant donc que tous ces agents se trouvent sur des plates-formes - donc a priori machines - différentes), on peut maintenant en exécutant ces programmes sur leur machine respectives "exécuter" ce SMA.

5.4

L'exécution de ce SMA se fait par les commandes suivantes dans différentes JVM (bien entendu éventuellement sur différentes machines, si tous ces exemples sont lancées sur une même machine, il ne faudra pas oublier de préciser les ports) :

- `java fr.lifl.magique.Start chap5.SuperImp`
- `java fr.lifl.magique.Start chap5.ParityImp` *superHost:port*
- `java fr.lifl.magique.Start chap5.CollatzImp` *superHost:port*
- `java fr.lifl.magique.Start chap5.AdderImp` *superMathHost:port*
- `java fr.lifl.magique.Start chap5.MultiplierImp` *superMathHost:port*
- `java fr.lifl.magique.Start chap5.DividerImp` *superMathHost:port*

Chapter 6

Dynamicité

Un point important pour un SMA est que celui-ci puisse évoluer dynamiquement. L'évolution peut se situer sur le plan individuel : les agents peuvent apprendre ou perdre des compétences, ou sur le plan organisationnel : la structure du SMA est modifiée, avec l'apparition ou la disparition d'agents par exemple.

Nous allons présenté maintenant ces deux aspects.

6.1 Évolution individuelle

L'évolution d'un agent se fait de nombreuses manières : on pourrait dire qu'il évolue dès qu'il fait la connaissance d'autres agents qu'il ajoute à son agenda, ou même dès qu'il obtient une information sur tel ou tel autre agent, dans ce cas, c'est sa connaissance des autres qui est modifiée. Cependant ce qui nous intéresse ici c'est essentiellement l'évolution des capacités de l'agent c'est-à-dire de ses compétences.

6.1.1 Acquisition dynamique de compétence

En fait, nous avons déjà vu comment faire évoluer dynamiquement les compétences d'un agent : nous n'avons même fait que cela ! En effet, lorsque que l'on ajoute des compétences à nos agents, ceci "existe" déjà. Les `addSkill` que nous avons effectués réalisent l'acquisition dynamique d'une compétence. Il suffit donc qu'un agent exécute une instruction `addSkill` pour acquérir une compétence, et ceux à n'importe quel moment de son cycle de vie. La seule restriction est que la nouvelle compétence n'apporte pas une méthode ayant la même signature qu'une méthode déjà connue via une autre compétence. Dans le cas contraire une exception `SkillAlreadyAcquiredExecption` est levée.

6.1.2 Oubli dynamique de compétence

L'oubli se fait de manière tout aussi simple grâce à la méthode `removeSkill`. Celle-ci prend pour paramètre la signature sous forme de chaîne de caractères d'une des méthodes apportée par la compétence à oublier. Attention, toutes les méthodes apportées par la compétence seront oubliées en même temps. Il convient d'être vigilant lors de l'écriture de la signature en respectant exactement la signature, et en mettant des espaces après chaque classe des paramètres.

Examinons un petit exemple. considérons la compétence suivante :

```
package chap6;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
import java.util.*;

public class ToBeForgottenSkill extends MagiqueDefaultSkill{
    public ToBeForgottenSkill (Agent a){
       super(a);
    }

    public void firstMethod(String s) {
       System.out.println("firstMethod "+s);
    }

    public void secondMethod(String s, ArrayList al) {
       System.out.println("secondMethod "+s+" " + al);
    }
}// ToBeForgottenSkil
```

Nous allons construire un petit programme élémentaire qui construit un agent, lui fait acquérir (dynamiquement) cette compétence, exploite une des méthodes de cette compétence et oublie cette compétence. Un nouvel appel à la même méthode va alors permettre de se rendre compte que la compétence n'est plus connue, un niveau de trace de 2 met ceci en évidence avec l'apparition du message `"unknwon request"`. On constate qu'il en est de même pour la seconde méthode de la compétence : c'est bien toute la compétence qui a été oubliée.

```
package chap6;
import fr.lifl.magique.*;

public class IForgetImp extends AbstractMagiqueMain {

   public void theRealMain(String[] args) {
     if (args.length == 1)
         Agent.setVerboseLevel(Integer.parseInt(args[0]));

     Agent a = createAgent("forgotter");
     a.addSkill(new ToBeForgottenSkill(a));

     a.perform("firstMethod", new Object[]{"hello world"});

     a.removeSkill("firstMethod(java.lang.String)");
     //a.removeSkill("secondMethod(java.lang.String,
     //                            java.util.ArrayList)");

     a.perform("firstMethod",
               new Object[]{"encore hello world"});
     a.perform("secondMethod",
               new Object[]{"hello world",
                            new java.util.ArrayList()});
   }
}// IForgetImp
```

L'appel par

```
java fr.lifl.magique.Start chap6.IForgetImp 4444 2
```

permet de visualiser les requêtes à des compétences inconnues.

On peut noter dans l'exemple, qu'il est nécessaire de donner la signature exacte, avec le nom complet des classes des paramètres, ainsi `String` n'est pas correct et doit être écrit `java.lang.String`. De même pour `ArrayList` qui devient

`java.util.ArrayList`. On remarque également que lorsqu'une méthode à plusieurs paramètres, la signature se note en plaçant une virgule immédiatement après chaque classe, sans passer d'espace, par contre cette virgule est elle suivie d'un unique espace.

Dans cet exemple en plaçant la ligne contenant le premier `a.removeSkill` et en enlevant le commentaire devant la suivante, on peut vérifier que le comportement du programme est le même : il suffit d'oublier n'importe quelle méthode de la compétence pour l'oublier dans son ensemble.

Si l'on cherche à faire oublier une compétence que l'on ne possède pas, l'exception `SkillNotKnownException` est levée (ce sera notamment le cas si l'on se trompe dans l'écriture du nom de la méthode à oublier).

6.1.3 Énseignement dynamique de compétence

Nous en arrivons sans doute au point le plus intéressant. Celui-ci est sans doute une particularité et un apport de MAGIQUE encore plus intéressants. Il s'agit de permettre à deux agents, même placés sur des machines différentes de s'enseigner des compétences. Et cela peut être réalisé sans qu'il soit nécessaire d'effectuer des hypothèses sur l'existence de telle ou telle classe dans l'environnement (basiquement le `CLASSPATH`) de l'agent apprenant. Il est bien sûr cependant indispensable que l'agent apprenant connaisse ces compétences !

Ceci se réalise par l'invocation de la requête `learnSkill` par l'apprenant :

- `learnSkill(java.lang.String , java.lang.String)`

`learnSkill` permet à un agent d'apprendre une compétence (dont on donne le nom complet dans le premier paramètre) auprès d'un autre agent (dont on donne le nom en second paramètre). Il existe des variantes avec des paramètres de cette compétence : lorsque le constructeur de la compétence à apprendre requiert des paramètres ceux-ci doivent être précisés, à noter que si l'agent lui-même doit être passé en paramètre, alors ce cas est précisé par un booléen[1] (voir la doc API pour plus de détails).

Nous allons illustrer cet aspect toujours avec notre exemple Ping-Pong : le superviseur sera le seul qui aura la possibilité de connaître les compétences `PingSkill` et `PongSkill`. Il enseignera `PingSkill` à l'agent `ping` (cela se fait en demandant à `ping` d'apprendre) et par contre c'est l'agent `pong` qui demandera de lui-même au superviseur de lui enseigner la compétence `PongSkill`, ceci pour montrer les deux aspects.

```
package chap6;
import fr.lifl.magique.*;

public class SuperImp extends AbstractMagiqueMain {

   // args[0] = verboseLevel **  args[1] = ping Host
   public  void theRealMain(String[] args) {

      Agent.setVerboseLevel(Integer.parseInt(args[0]));

      Agent a = createAgent("super");
      a.perform("ping@"+args[1], "learnSkill",
                "chap6.PingSkill", a.getName(), Boolean.TRUE);
      a.perform("ping");
   }
}
```

[1] Cela est nécessaire entre autre car un agent n'est pas sérialisable et ne peut donc être passé en paramètre d'une requête

```
package chap6;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class PingImp extends AbstractMagiqueMain {

    // args[0] = verboseLevel **  args[1] = super Host
    public void theRealMain(String[] args) {

        Agent.setVerboseLevel(Integer.parseInt(args[0]));
        Agent a = createAgent("ping");
        a.connectToBoss("super@"+args[1]);
    }

}
```

```
package chap6;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class PongImp extends AbstractMagiqueMain {

   // args[0] = verboseLevel **  args[1] = super Host+port
   public void theRealMain(String[] args) {

      Agent.setVerboseLevel(Integer.parseInt(args[0]));

      Agent a = createAgent("pong");
      a.connectToBoss("super@"+args[1]);
      a.perform("learnSkill",
                new Object[]{"chap6.PongSkill",
                             "super@"+args[1], Boolean.TRUE});
   }
}
```

Évidemment l'invocation de `learnSkill` dans `PongImp` ne peut se faire qu'après la connexion au superviseur. On note le `Boolean.TRUE` passé en paramètre de la requête `learnSkill`, celui-ci indique que l'agent apprenant est un argument du constructeur de la compétence apprise.

Le lancement se fait ainsi :

`java fr.lifl.magique.Start chap6.SuperImp 4444 2` ***PingImp host:port***

`java fr.lifl.magique.Start chap6.PingImp 5555 2` ***SuperImp host:port***

`java fr.lifl.magique.Start chap6.PongImp 6666 2` ***SuperImp host:port***

Cependant une expérience plus intéressante doit être menée. Nous avons indiqué que l'apprentissage de compétence pouvait se réaliser dès lors que l'agent enseignant a accès au code de la compétence... Ce point peut être illustré très simplement : il suffit de placer `chap6.PingImp` et `chap6.PongImp` dans des environnements qui n'ont pas accès (par leur `CLASSPATH`) aux compétences `chap6.PingSkill` et `chap6.PongSkill` (tout en laissant accessibles à `chap6.SuperImp`) et de relancer l'exemple. On constate que le comportement est rigoureusement le même : le (byte)code des compétences a donc été "physiquement" transféré de l'environnement de `SuperImp` à celui des deux agents `ping` et `pong` ! L'échange de compétence peut donc se faire sans condition préalable entre tout agent. Cette opération est réalisée par l'intermédiaire des plates-formes.

La démonstration est encore plus parlante si vous disposez de plusieurs machines et pouvez donc placer les agents sur ces machines distinctes (sinon il suffit de créer des répertoires `chap6` distincts et d'y placer les `PongImp.class` et `PingImp.class`). Dans ce cas les compétences sont automatiquement transférées d'un machine à l'autre. Cette fonctionnalité offerte par MAGIQUE offre évidemment une grande souplesse à l'utilisation et facilite la distribution des applications construite avec MAGIQUE.

6.2 Création dynamique d'agents

Il est également possible de créer dynamiquement des agents et de les intégrer à une hiérarchie. On peut distinguer deux cas de figure : le premier, le plus simple, consiste pour un agent à créer des agents au sein de sa propre plate-forme, le second correspond à la création d'agents distants.

6.2.1 Création sur la même plate-forme

Abordons le cas d'une création "locale". Toujours avec notre application de ping-pong, nous allons avoir un agent superviseur qui va créer deux agents `ping` et `pong`, leur attribuer leurs compétences et les relier à lui-même en tant que superviseur.

C'est toujours la plate-forme qui crée concrètement les agents. L'attribution de compétence et la connexion au superviseur peut se faire dans ce cas dans un ordre quelconque. Voici les classes nécessaires :

```
package chap6;
import fr.lifl.magique.*;

public class SuperCreatorImp extends AbstractMagiqueMain {

    // args[0] = verboseLevel **  args[1] = ping Host
    public  void theRealMain(String[] args) {

        Agent a = createAgent("super");
        a.addSkill(new chap6.CreatorSkill(a));
        a.perform("create");
   }

}
```

```
package chap6;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;
import fr.lifl.magique.skill.*;
public class CreatorSkill extends MagiqueDefaultSkill {

    public CreatorSkill(Agent a){ super(a); }

    public void create() {
      Platform myPlatform = getPlatform();

      Agent ping = myPlatform.createAgent("ping");
      ping.addSkill(new chap6.PingSkill(ping));
      myPlatform.addAgent(ping);
      ping.connectToBoss(getName());

      Agent pong = myPlatform.createAgent("pong");
      pong.addSkill(new chap6.PongSkill(pong));
      myPlatform.addAgent(pong);
      pong.connectToBoss(getName());

      perform("ping");
    }
} // CreatorSkill
```

6.2.2 Création distante d'agents

Ce cas est un peu plus complexe que le précédent. Nous allons illustrer avec la même "application", mais cette fois `ping` et `pong` sont créer dans un autre plate-forme que le superviseur.

Ici il va falloir utiliser l'agent plate-forme de la plate-forme d'accueil pour créer les agents voulus. Pour qu'il puisse dialoguer avec cet agent plate-forme, le superviseur doit s'y connecter. Il s'agit ici d'une connexion directe indépendante de toute hiérarchie. Ensuite une requête de création de l'agent voulu est envoyé à cet agent plate-forme (ici la requête est nécessairement nommée car faite en dehors de toute hiérarchie). Vient ensuite la requête de connexion de l'agent créé à son superviseur. Cette fois la connexion se fait absolument avant l'enseignement des compétences, car pour pouvoir avoir un enseignement de compétences distants, avec éventuellement transfert du bytecode, il est évidemment nécessaire que le lien existe entre l'apprenant est l'"enseignant".

On obtient :

```
package chap6;
import fr.lifl.magique.*;

public class DistantCreatorImp extends AbstractMagiqueMain {

  //args[0]=verboseLevel|args[1]=ping Host|args[2]=pong host
  public  void theRealMain(String[] args) {

    Agent.setVerboseLevel(Integer.parseInt(args[0]));

    Agent a = createAgent("distantSuper");
    a.addSkill(new chap6.DistantCreatorSkill(a));
    a.perform("distantCreate", new Object[]{args[1],args[2]});
  }
}
```

```
package chap6;
import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;
import fr.lifl.magique.skill.*;
public class DistantCreatorSkill extends MagiqueDefaultSkill {

    public DistantCreatorSkill(Agent a){ super(a); }

    public void distantCreate(String pingPlatform,
                              String pongPlatform) {

String PMAN = Platform.PLATFORMMAGIQUEAGENTNAME;
     String pingPlatformAgent = PMAN+"@"+pingPlatform;
     connectTo(pingPlatformAgent);
     String pingName = (String) askNow(pingPlatformAgent,
                                       "createAgent","ping");
     perform(pingPlatformAgent,"connectAgentToBoss",
                                pingName,getName());
     perform(pingName,"learnSkill", "chap6.PingSkill",
                                    getName(), Boolean.TRUE);

     String pongPlatformAgent = PMAN+"@"+pongPlatform;
     connectTo(pongPlatformAgent);
     String pongName = (String) askNow(pongPlatformAgent,
                                       "createAgent","pong");
     perform(pongPlatformAgent,"connectAgentToBoss",
                                pongName,getName());
     perform(pongName,"learnSkill", "chap6.PongSkill",
                     getName(), Boolean.TRUE);

     perform("ping");
    }
} // DistantCreatorSkill
```

Pour faire ce teste, il faut lancer les plate-formes sur les machines destinées à accueillir `Ping` et `Pong`, puis de lancer le “créateur” :

- `java fr.lifl.magique.PlatformLauncher` (pour `Ping` et `Pong`)
- `java fr.lifl.magique.Start DistantCreatorImp 4444 2` *PingHost:port PongHost:port*

6.3 Une dynamicité transparente

Dans le même esprit que ce qui vient d’être dit nous allons précisé un point que nous n’avons pas évoqué au moment où nous avons parlé de l’invocation de requêtes : que se passe-t-il lorsqu’un agent passe en paramètre à une requête un objet d’une classe inconnue de l’agent qui reçoit la requête ? Pour être plus précis, nous devrions dire que c’est la plate-forme de l’agent récepteur qui ne connaît pas (n’a pas accès à) la classe de cet objet.

Cela peut se produire dans le cas suivant : considérons la classe `UnObject.java` que voici.

```
package chap6;

public class UnObject implements java.io.Serializable {
    private Object ball;
    public UnObject (Object ball){
       this.ball = ball;
    }
    public String toString() { return ball.toString(); }
}// UnObject
```

Les instances de cette classe agrège un `Object` quelconque. Ces instances sont utilisés comme paramètres de la méthode `ping` (resp. `pong`) de la compétence `PingSkill2` (resp. `PongSkill2`) :

```
package chap6;

import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class PingSkill2 extends MagiqueDefaultSkill {
    public PingSkill2(Agent a){super(a);}

    public void ping(UnObject ball, UnObject autreBall) {
       perform("display",new Object[]{"ping : "+ball.toString()+
                                      " "+autreBall.toString()});
       perform("pong",ball,autreBall);
    }
} // PingSkill2
```

Le problème se pose lorsque l’on passe en paramètre d’une requête des instances de `UnObject` qui agrège des objets dont la classe ne sera pas connue (ou accessible) à l’agent qui recevra cette requête. C’est le cas pour l’invocation à la méthode `ping` par le superviseur que voici :

```
package chap6;

import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class SuperImp2  extends AbstractMagiqueMain {

   public void theRealMain(String[] args) {
     if (args.length==1)
       Agent.setVerboseLevel(new Integer(args[0]).intValue());

     Agent sup = createAgent("super");
     sup.perform("ping",new UnObject(new Ball("la balle")),
                 new UnObject(new AnotherBall("autre balle")));
   }
} // SuperImp2
```

avec pour la classe `Ball` (la classe `AnotherBall` est similaire) :

```
package chap6;

public class Ball implements java.io.Serializable {
    private String myBall;
    public Ball (String myBall){ this.myBall = myBall; }
    public String toString() { return myBall; }
}// Ball
```

Que se passe-t-il pour l'agent `ping` si celui-ci tourne sur une autre plate-forme qui **n'a pas accès** aux classes `Ball` et `AnotherBall` ?

```
package chap6;

import fr.lifl.magique.*;
import fr.lifl.magique.platform.*;

public class PingImp2 extends AbstractMagiqueMain {
    // args[1] = verbose level | args[0]= boss host name
    public void theRealMain(String[] args) {
      if (args.length==2)
          Agent.setVerboseLevel(Integer.parseInt(args[1]));
      Agent a = createAgent("Ping");
      a.addSkill(new chap6.PingSkill2(a));
      a.connectToBoss("super@"+args[0]);
    }
} // PingImp2
```

(Les fichiers `chap6/PongSkill2.java` et `chap6/PongImp2.java` sont similaires)

La réponse est : il ne se passe rien de remarquable pour l'utilisateur ! Ce qui, paradoxalement, est en fait surprenant a priori : la classe n'étant pas connue, il devrait y avoir une exception levée. Mais cela serait préjudiciable à la souplesse d'utilisation de MAGIQUE : il serait nécessaire de faire des présuppositions sur la présence de telle ou telle classe sur les différentes plate-formes.

Pour contourner ce problème, MAGIQUE se charge de transmettre le bytecode des classes des objets nécessaires à la bonne interprétation des paramètres des requêtes entre les plate-formes. Dans notre cas, le bytecode de `Ball` et `AnotherBall` est transmis depuis la plate-forme de `SuperImp2` à la plate-forme de `PingImp2` (et `PongImp2`). La technique sous-jacente est la même que celle qui permet l'échange de compétences.

Cette fonctionnalité soulage d'un problème important qui aurait pénalisé la distribution des applications multi-agent.

Chapter 7

Déconnexion et mort d'un agent

Nous avons vu comment créer, connecter et distribuer des agents. Reste à voir les opérations inverses : déconnecter un agent (et vérifier qu'il peut ensuite être reconnecté), l'arrêter et même arrêter complètement une plate-forme.

Les compétences nécessaires à lé déconnexion et l'arrêt ("mort") d'un agent lui sont connus à la création. Il s'agit principalement des compétences suivantes :

`askForDisconnectionFrom` se déconnecter d'un agent donné

`askForDisconnectionFromMyBoss` se déconnecter de son superviseur

`disconnectFromAll` se déconnecte de tous les agents

`disconnectAndDie` se déconnecter de tous et "mourir"

Après une déconnexion il est évidemment possible à l'agent de se reconnecter en utilisant les services déjà connus.

Le code suivant contient la plate-forme dans laquelle un agent va se déconnecter, se reconnecter, se "suicider", être recréé, puis la plate-forme s'arrête (définitivement). Des appels à une compétences d'un agent sur une autre plate-forme sont faits pour montrer l'effectivité des reconnexions et recréation.

```
package chap7;
import fr.lifl.magique.*;

public class Disconnect2 extends AbstractMagiqueMain{

    // args[0] = verbose level  - args[1] = boss platform
    public void theRealMain(String[] args) {

      Agent.setVerboseLevel(Integer.parseInt(args[0]));

      Agent ag = createAgent("agent2");
      ag.connectToBoss("agent1@"+args[1]);
      ag.perform("tralala");
      pause(1);

      System.out.println("deconnection");
      ag.perform("askForDisconnectionFromMyBoss");
      pause(2);

      System.out.println("reconnection");
      ag.connectToBoss("agent1@"+args[1]);
      ag.perform("tralala");
      pause(3);

      System.out.println("mort de l'agent");
      ag.perform("disconnectAndDie");
      pause(4);

      System.out.println("re``creation'' de l'agent");
      ag = createAgent("agent2");
      ag.connectToBoss("agent1@"+args[1]);

      ag.perform("tralala");
      pause(5);

      System.out.println("arret plate-forme");

      ag.getPlatform().stop();
    }

    private void pause(int index) {
       try {
           System.out.println("dodo "+index);
           Thread.sleep(1000);
           System.out.println("reveil "+index);
       } catch (Exception e) { e.printStackTrace(); }

    }
}// Disconnect2
```

et pour être complet les deux petits fichiers complices :

```
package chap7;
import fr.lifl.magique.*;

public class Disconnect1 extends AbstractMagiqueMain {
    // args[0] = verbose level
    public void theRealMain(String[] args) {

      Agent.setVerboseLevel(Integer.parseInt(args[0]));

      Agent ag = createAgent("agent1");
      ag.addSkill(new chap7.UnSkill());
      System.out.println("\n \n");
    }

}// Disconnect1
```

```
package chap7;
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;

public class UnSkill implements Skill {
    public void tralala() { System.out.println("tralala"); }
}// UnSkill
```

Ces fichiers se lancent par :

- `java fr.lifl.magique.Start chap7.Disconnect1 4444` *verbose*
- `java fr.lifl.magique.Start chap7.Disconnect2 5555` *verbose* *Disconnect1-Host*`:4444`

Une fois que la plate-forme avec `Disconnect2` s'est arrêtée, il peut être intéressant de la relancer pour remarquer que la reconnexion entre plate-forme est possible.

Une autre expérience intéressante est tuer violemment la JVM avec `Disconnect2` par un superbe CTRL-C en cours d'exécution et ensuite de la relancer. On peut remarquer que la encore la reconnexion est possible (bien sûr puisqu'on relance le programme, toutes les requêtes sont réexécutées, il n'y a pas de reprise à l'endroit du "crash" provoqué).

Chapter 8

L'environnement graphique d'exécution

Un environnement graphique permettant :

- la construction d'agents par enrichissement de compétences
- l'organisation de ces agents en une forêt de hiérarchies
- la distribution/déploiement du SMA sur le réseau
- l'interaction avec les agents

Cet environnement ce lance par la commande

```
java fr.lifl.magique.gui.LanceurAgents
```

Il est éventuellement nécessaire de préciser le `classpath` par l'option `-classpath`, notamment, il va falloir s'assurer que toutes les compétences que vous allez donner à vos agents sont accessibles. Vous obtenez alors à l'écran quelque chose comme ce qui est présenté à la figure 8.1.

Vous pouvez alors en quelques clicks construire vos agents et votre SMA. L'environnement et son fonctionnement sont présentés plus en détail à l'adresse : `http://www.lifl.fr/MAGIQUE/gui`

Pour pouvoir déployer vos agents sur le réseau, la seule condition et qu'une plate-forme soit prête à les accueillir. Nous avons vu au chapitre 6 que grâce aux mécanismes d'échange dynamique contenus dans MAGIQUE, il n'était pas nécessaire de faire des hypothèses sur le code accessible par les plateformes. Il convient donc uniquement d'avoir lancer sur les machines hôtes une plate-forme d'accueil par :

```
java fr.lifl.magique.PlatformLauncher [port]
```

ou

```
java -jar magique.jar [port]
```

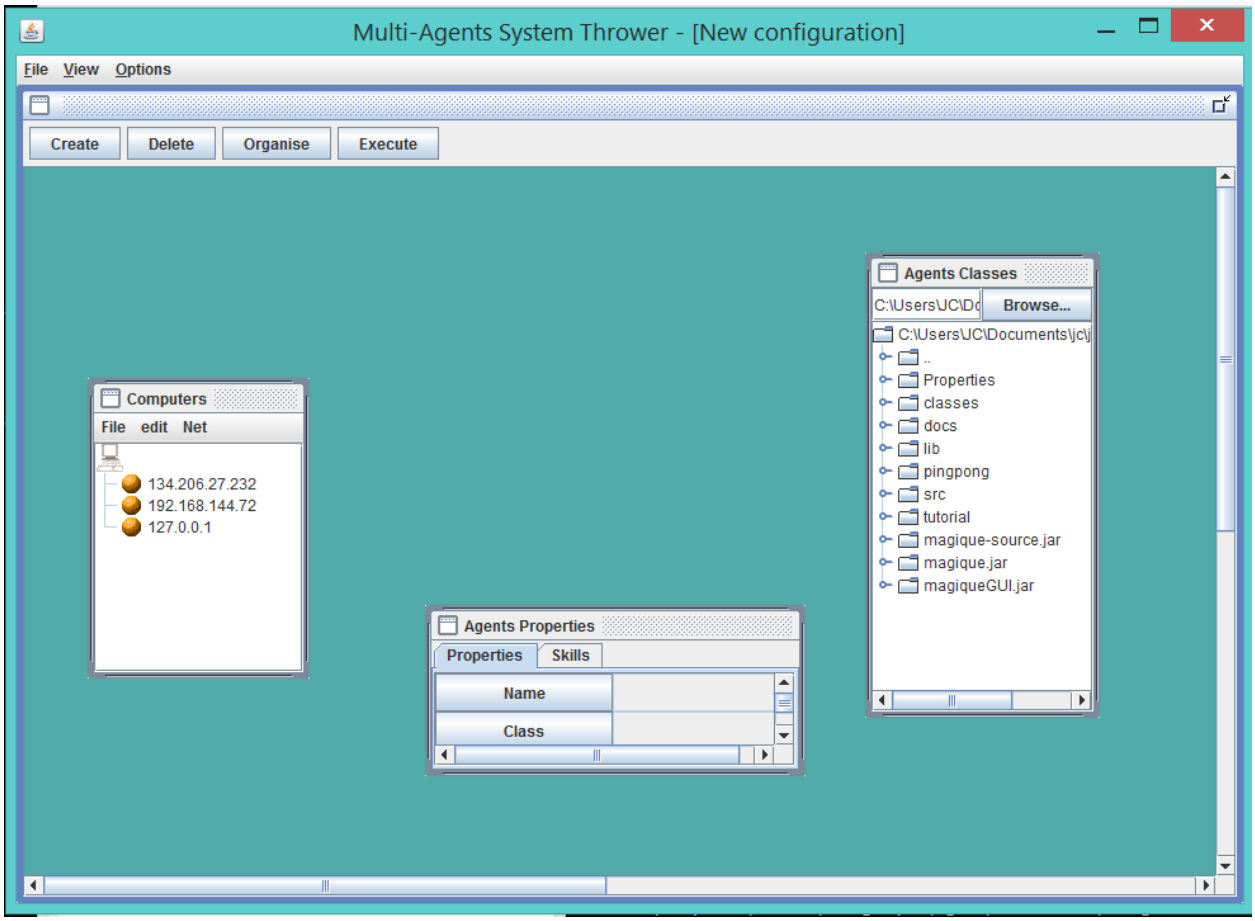

Figure 8.1: L'environnement graphique