Toy Problem test (le tout premier) : Factorielle
(Attention ce problème est plus complexe qu'il n'en a l'air au premier abord)
Il s'agit de créer une population d'agents composée de la façon suivante :

- ▷ *un agent factorielle F_1 qui sait comment on calcule une factorielle, mais qui ne sait pas multiplier,*
- ▷ *3 agents multiplicateurs M_1, M_2, M_3 qui connaissent les quatre opérations (on supposera pour simplifier qu'ils savent réaliser des opérations à deux arguments).*

Le but du jeu est de leur faire calculer 10! (factorielle 10).

Il faudra présenter tout le code effectivement écrit et être prêt à justifier les choix de conception (synchronisme, asynchronisme,...) et à répondre aux critiques sur des alternatives possibles. Il n'y a pas d'autres contraintes.

1 Solution avec MAGIQUE

Jean-Christophe Routier et Philippe Mathieu. Equipe SMAC - LIFL. Lille.

MAGIQUE est la proposition de plate-forme multi-agent de l'équipe SMAC (Systèmes multi-agents et coopération) du LIFL (Laboratoire d'Informatique Fondamentale de Lille). Il s'agit à la fois de la proposition de modèles d'agent, d'organisation de systèmes multi-agent et d'un environnement de développement de systèmes multi-agents.
Nous décrirons rapidement ici ces modèles et l'API les mettant en œuvre avant de présenter notre solution au problème *factorielle*.

1.1 Présentation de MAGIQUE

MAGIQUE vient de "Multi-AGent hiérarchIQUE" et propose à la fois un modèle organisationnel [BM97], basé sur une organisation hiérarchique par défaut, et un modèle d'agent [RMS01], qui est basé sur une construction incrémentale des agents.

1.1.1 Le modèle d'agent : construction par enrichissement de compétences.

Le modèle d'agent est basé sur une construction incrémentale des agents à partir d'un agent élémentaire (ou atomique) par l'acquisition dynamique de compétences. En se plaçant plus du point de vue du développeur, une compétence peut être vue comme un composant logiciel qui regroupe un ensemble cohérent de fonctionnalités. Ces compétences peuvent être construites indépendamment de tout agent et réutilisées dans différents contextes. Nous utilisons "compétence" plutôt que "service" [1].
Nous affirmons que deux compétences initiales sont nécessaires et suffisantes à un *agent atomique* pour lui permettre d'évoluer vers tout agent désiré : une pour interagir et une pour acquérir de nouvelles compétences (voir détails dans [RMS01]).
Ainsi nous pouvons considérer que tous les agents sont à leur naissance (ou création) similaires (du point de vue des compétences) : une coquille ne possédant les deux compétences précédemment mentionnées.

[1] Nous réservons *service* pour "le résultat de l'exploitation d'une compétence".

Il en résulte que les différences entre agents proviennent de leur “éducation”, c-à-d. des compétences qu’ils ont acquis durant leur “existence”. Ces compétences peuvent, soit avoir été données lors de la création de l’agent, soit avoir été apprises dynamiquement suite à des interactions avec d’autres agents (si l’on considère que le développeur est un agent, le premier cas est inclus dans le second). Cette approche n’entraîne pas de limitations sur les capacités d’un agent. Enseigner une compétence à un agent lui donne la possibilité de jouer un rôle particulier dans le système multi-agent auquel il appartient.

1.1.2 Le modèle organisationnel.

Dans MAGIQUE, il existe une structure organisationnelle par défaut : la hiérarchie. L’existence d’une organisation a priori offre la possibilité de proposer un mécanisme automatique permettant la recherche d’un prestataire de service.
La hiérarchie caractérise la structure basique des accointances dans le SMA et fournit un support par défaut pour le routage des messages entre agents. Un lien hiérarchique indique un canal de communication entre les agents concernés. Quand deux agents d’une même structure échangent un message, par défaut celui-ci passe par la structure arborescente.
Avec seulement ces communications hiérarchiques, l’organisation serait trop rigide ainsi MAGIQUE offre la possibilité de créer des liens directs (c-à-d. en dehors de la structure hiérarchique) entre agents. Nous les appelons “*liens d’accointance*” (par opposition au “*liens hiérarchiques*”). La décision de créer de tels liens dépend d’une stratégie de l’agent. Cependant, le but visé est le suivant : après un certain temps, si il se trouve qu’une requête à une compétence se produit fréquemment avec un autre agent, l’agent peut décider de créer dynamiquement un lien d’accointance pour cette compétence. Le but est bien évidemment de promouvoir les interactions “naturelles” entre agents aux dépens des interactions hiérarchiques.
Si le modèle MAGIQUE propose une organisation par défaut hiérarchique, cette structure est destinée à évoluer en accord avec le comportement dynamique du SMA en favorisant les relations les plus fréquentes. Il en résulte, qu’après un certain temps, le SMA devrait plus ressembler à un graphe. L’intérêt est que, la structuration étant dynamique et adaptative, le concepteur d’un SMA peut se reposer en partie sur ce mécanisme lors de la conception de l’organisation de son SMA.
La dynamicité est un point clef dans MAGIQUE, il est également possible d’avoir une évolution dynamique de la structure de la hiérarchie (par création d’agents assistants) ou de la répartition des compétences dans une hiérarchie (afin d’éviter les problèmes liés au caractère critique d’une compétence ressource). Ces points sont présentés dans [MRS02a].

Parce qu’elle offre un réseau d’accointances par défaut, la structure hiérarchique permet de disposer d’un mécanisme automatique de délégation de requêtes.
La réalisation d’une tâche par un agent requiert l’exploitation d’un certain nombre de compétences auxquelles il devra faire appel, qu’il les possède ou pas. Dans les deux cas, le mode d’invocation de ces compétences se réalise de manière identique. La délégation éventuelle de la réalisation à un autre agent est transparente pour lui

grâce à la structure hiérarchique. Le mécanisme est le suivant : si l'agent "possède" la compétence, il l'utilise directement, sinon plusieurs cas de figure sont possibles : d'abord il a une accointance particulière pour cette compétence, il demande alors à cette accointance de la réaliser pour lui; sinon s'il est superviseur d'un agent compétent, il lui transmet la réalisation de la compétence; enfin, dans le dernier cas, il demande à son superviseur de trouver quelqu'un de compétent pour lui et celui-ci applique ce même mécanisme récursivement.
Un des avantages de cette délégation transparente est d'accroître la fiabilité du système : l'agent qui exécutera la compétence étant indifférent pour l'invocateur, il peut changer entre deux "appels" à la même compétence (qu'il ait disparu du système suite à une panne ou qu'il soit surchargé ou...)

Le mécanisme de délégation automatique couplé à une évolution dynamique et automatique de la hiérarchie au profit des relations privilégiées permet au concepteur de se débarrasser au moins partiellement de l'un des problèmes majeurs qui se pose lors de la conception d'un SMA : la création du réseau d'accointances des agents, c-à-d. "Avec qui un agent est il en relation et pour quels services ?".

1.1.3 L'API MAGIQUE

Ces modèles ont été concrétisés au travers d'une API JAVA. Elle permet le développement de SMA distribués sur un réseau hétérogène [RM02, MRS02b]. Les agents sont développés de manière incrémentale (et dynamiquement si besoin) par enrichissement de compétences et sont organisés hiérarchiquement au sein des SMA. L'API peut être téléchargée à `http://www.lifl.fr/SMAC` rubrique MAGIQUE.

Cette API permet la création d'applications basées sur des systèmes multi-agent à la MAGIQUE. Les agents peuvent être naturellement (et simplement) distribués sur un réseau hétérogène de machine, l'API masque les problèmes liés au réseau en fournissant les primitives de connexion/déconnexion et de communication. Elle se base sur un système de plate-formes d'accueil des agents au sein d'une machine (cette plate-forme étant gérée par un agent). Il peut y avoir plusieurs plate-formes par machine si nécessaires. La présence de plate-formes permettent notamment d'assurer les échanges nécessaires de bytecode lorsque deux agents distants s'enseignent des compétences. Les noms des agents sont normalisés à leur création sous la forme `nom@num_ip:port` où `num_ip` est le numéro IP de la plate-forme hébergeant l'agent et `port` le port utilisé par cette plate-forme pour la communication.

Nous allons, très brièvement et succinctement, présenter les principales fonctionnalités de cette API[2].
Les agents que l'on peut créer correspondent à notre modèle d'agents construit incrémentalement à partir d'un agent atomique ne possédant que deux compétences : la communication et l'acquisition de compétences. Ainsi l'API fournit une classe `AtomicAgent` correspondant à ce modèle. Cette classe dispose essentiellement des mé-

[2]Un tutoriel téléchargeable sur le site fournit une présentation plus détaillée et plus complète de l'API.

thodes `connectTo` et `send` pour la communication et de la méthode `addSkill(Skill s)` pour l'acquisition dynamique de compétences.
Une compétence est simplement une classe implémentant l'interface `fr.lifl.magique.Skill`. Les méthodes publiques de cette classe correspondent aux services offerts par la compétence. En passant une instance d'une telle classe comme paramètre de l'invocation par un agent de la méthode `addSkill`, on enrichit celui-ci de cette compétence. Il peut dès lors répondre aux requêtes de tout autre agent d'exploitation des services (méthodes) de la compétence.
Il y a trois manières pour un agent a d'envoyer à un agent b une requête d'exploitation d'une compétence *skill* (en fait méthode d'une compétence acquise par b) :

1. une requête sans attente de réponse :
   a`.perform(`b`,"skill",params)`
2. une requête avec résultat asynchrone :
   `Request r =` a`.ask(`b`,"skill",params)`
   Le flux d'exécution de a n'est pas bloqué par cet appel et la réponse peut alors être récupérée ultérieurement, quand nécessaire, par :
   `Object result = r.returnValue()`
   Il est possible en utilisant la commande `isAnswerReceived(r)` re savoir si la réponse à une requête à été reçue.
3. une requête avec résultat asynchrone (blocage du flux d'exécution) :
   `Object result =` a`.askNow(`b`,"skill",params)`

Pour que a utilise l'une de ses propres compétences, on peut évidemment remplacer b par a comme premier paramètre ou, plus simplement, ne rien mettre.
Nous obtenons avec ce mécanisme une sémantique assez faible au niveau des envois de messages puisqu'ils correspondent en fait à des invocations (de signatures) de méthodes. Mais il a l'énorme avantage d'être simple à appréhender et à mettre en œuvre par le concepteur d'une application SMA.

Cependant pour pouvoir intervenir dans une organisation à la MAGIQUE, nos agents doivent posséder un certain nombre de fonctionnalités de base. La classe `Agent` (que l'on aurait pu appeler `MagiqueAgent`) correspond alors à ces agents, elle est obtenue pas enrichissement de la classe `AtomicAgent`, et on y trouve (tel quel) la méthode suivante réalisant l'enrichissement dynamique (puisqu'invoquée après création) de l'agent atomique afin qu'il devienne un agent MAGIQUE :

```
protected void initBasicSkills() throws SkillAlreadyAcquiredException {
   this.addSkill(new fr.lifl.magique.skill.magique.BossTeamSkill(this));
   this.addSkill(new fr.lifl.magique.skill.system.ConnectionSkill(this));
   this.addSkill(new fr.lifl.magique.skill.magique.ConnectionToBossSkill(this));
   this.addSkill(new fr.lifl.magique.skill.magique.KillSkill(this));
   this.addSkill(new fr.lifl.magique.skill.system.DisplaySkill());
   this.addSkill(new fr.lifl.magique.skill.system.AddSkillSkill(this));
   this.addSkill(new fr.lifl.magique.skill.system.LearnSkill(this));
}
```

Les compétences qui sont ici enseignées à notre agent lui permettent de jouer son rôle dans les organisation "à la MAGIQUE", notamment avec la gestion de la hiérarchie prise en compte par les compétences du paquetage `fr.···.skill.magique`. Ainsi on peut construire une hiérarchie en connectant des agents à leur supérieur hiérarchique par la méthode `connectToBoss`. L'agent a se connectera à l'agent de nom *myBoss* en tant que membre de son équipe ainsi[3] :

```
a.connectToBoss("myBoss")
```

Mais les connections "directes", c'est-à-dire en dehors de toute hiérarchie, existent également. Elles sont créées grâce à la méthode `connectTo`.
Avec la mise en place de la hiérarchie, nos agents MAGIQUE récupèrent la possibilité d'utiliser le mécanisme automatique de délégation de compétences. Il suffit de ne pas indiquer de destinataire lors d'une requête par `perform`, `ask` ou `askNow` et le mécanisme de délégation est alors mis en œuvre, et donc la recherche automatique d'un agent compétent dans la hiérarchie à laquelle appartient l'agent.

Nous avons présenté, bien que brièvement, suffisamment l'API MAGIQUE pour permettre à chacun de comprendre la solution proposée au problème *factorielle*. Nous allons maintenant étudier cette solution.

1.2 Le problème *factorielle* avec MAGIQUE

1.2.1 Analyse

Le sujet du toy problem imposait les agents, l'analyse sera donc rapide, mais il est vrai qu'il n'y avait de toute façon ici pas de mystère (rappel : un agent ne sait que faire la factorielle et trois autres ne savent que multiplier).

Compétences. Cependant, à la différence du sujet du problème, plutôt que de parler d'agents, nous devons dans le cas de la résolution avec MAGIQUE plutôt réfléchir en terme de compétences. Car ce sont elles qui sont primordiales. Pour résoudre notre problème, nous identifions donc naturellement deux compétences essentielles : la compétence de calcul de factorielle et la compétence de multiplication (les compétences permettant la réalisation des trois autres opérations de base mentionnées dans le sujet ne sont pas utiles, ici mais pourrait être bien sûr facilement définies). Nous appelons ces deux compétences `FactorialSkill` et `MultiplierSkill`.
Maintenant identifiées, ces compétences peuvent être développées, indépendamment de tout agent et de toute organisation de SMA.
La compétence `MultiplierSkill` fournit le seul service de multiplication de deux entiers :

```
public BigInteger mult(BigInteger x, BigInteger y)
```

Le code complet de cette compétence est fourni à la figure 2. Il est on ne peut plus simple à comprendre ou à écrire pour peu que l'on dispose du minimum de connaissances JAVA. La requête `perform("display",...)` permet juste d'utiliser la

[3] en fait cette méthode masque l'exploitation de la compétence, son code contient l'unique ligne :"`askNow("connectionToBoss", new Object[] bossName);`'' qui est bien une demande d'exploitation de compétence comme nous l'avons présentée précédemment

compétence d'affichage de l'agent, cette compétence pouvant être changée en fonction de l'application et des souhaits du concepteur. Par défaut il s'agit d'un affichage sur la sortie standard.
Quant à la compétence `FactorialSkill`, elle offrira un seul service (une méthode du point de vue du code) `factorielle` dont la signature[4] sera :

```
public BigInteger factorielle(BigInteger n)
```

Son code[5] est donné à la figure 3. Il nécessite sans doute un peu plus d'explication afin d'accélérer la compréhension du lecteur.
La variable `team` permet de gérer l'équipe des multiplicateurs. Dans une première phase, l'ensemble des calculs des multiplications $i \times (i+1)$ pour i variant de 2 en 2 et plus petit que n sont distribués aux agents multiplicateurs. Les requêtes sont envoyées à l'aide de la commande `ask` qui permet un envoi *asynchrone* et donc toutes les requêtes peuvent être envoyées, sans attendre les résultats. Toutes les requêtes envoyées sont rangées dans la variable `theQuestions`. La variable `team` est remise à jour dès que l'on a soumis une requête à chacun des multiplicateurs. Le ré-examen régulier de "l'équipe" des multiplicateurs, permet de prendre en compte tout nouvel agent multiplicateur rejoignant la hiérarchie en cours de calcul.
Dans une seconde phase, il faut récupérer les résultats des requêtes émises. Cela est effectuée par les appels à la commande `returnValue(...)` qui utilise les résultats de la méthode `firstAnsweredQuestion()`. Celle-ci recherche parmi les requêtes de `theQuestions` émises, la première dont on a reçu la réponse (grâce à la commande `isAnsweredReceived()`). Dès que l'on dispose de deux résultats, il faut demander le calcul de leur produit. Le principe d'envoi des requêtes est ici le même que dans la première phase. On continue jusqu'à ce qu'il n'y ait plus qu'une seule requête dont on attende la réponse, c'est-à-dire plus qu'un seul produit à calculer. C'est alors cette dernière requête qui contient le résultat du calcule de $n!$.

Les agents. Nous pouvons maintenant nous intéresser aux agents. Il s'agit de partir de nos agents MAGIQUE de base (qui correspondent donc aux agents atomiques du modèle déjà enrichis des compétences propres à MAGIQUE) et de leur enseigner les compétences applicatives requises, qui sont ici les `FactorialSkill` et `MultiplierSkill`. D'une manière général, la distribution de ces compétences est à la discrétion du concepteur et fonction de l'application, on peut pour une même résolution de problème avoir des distributions de compétences différentes. Cependant, ici, pour satisfaire le cahier des charges, nous allons équiper un seul agent de la compétence `FactorialSkill` et autant d'agents que voulus (3 si l'on souhaite se conformer exactement au sujet du toy problem) de la compétence `MultiplierSkill`
Il faut ensuite créer le SMA et son organisation. Nous choisissons une hiérarchie dont la racine est occupée par l'agent calculant la fatorielle auquel sont connectés par des liens hiérarchiques les agents multiplicateurs (cf Figure 1).
La figure 4 donne les codes complets des agents, on note les appels à `addSkill` pour les acquisitions de compétences (cette acquisition est bien dynamique car les

[4] Il n'y a aucun argument raisonnable pour limiter le calcul de notre application à la factorielle de 10, afin de permettre de calucler la factorielle de n'importe quel entier, nous travaillons avec des `BigInteger`.

[5] Ce code, ainsi que celui de MAGIQUE, commence à dater un peu, on y trouve donc par exemple les "vieux" `Vector` et `Enumeration` de JAVA et non pas les `Collection` et `Iterator`

agents sont déjà créés lors de cette invocation) et à `connectToBoss` (pour créer la hiérarchie).
Nous avons ajouté un agent supplémentaire, appelé `starter`, qui permet de démarrer le calcul. Celui-ci n'a aucune compétence particulière et n'a pas besoin d'appartenir à la hiérarchie. Il suffit qu'il soit en relation (connecté) à l'agent `fact`. Il peut alors lui demander de manière synchrone (avec `askNow`) d'effectuer le calcul de $n!$ pour un n donné. Cette fois l'agent destinataire de la requête (ici `fact@...`) doit impérativement être précisé car l'agent `starter` est en dehors d'une hiérarchie et ne peut donc pas profiter du mécanisme automatique de délégation de requêtes.

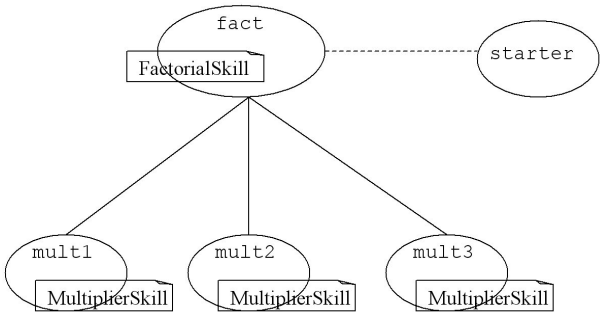

Figure 1: L'organisation du SMA proposée pour le problème factorielle.

```
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
import java.math.BigInteger;
public class MultiplierSkill extends DefaultSkill {
  public MultiplierSkill(Agent ag) { super(ag); }
  public BigInteger mult(BigInteger x, BigInteger y) {
    perform("display",new Object[]{getMyAgent().getName()+" multiplie "+x+" par "+y});
    return x.multiply(y);
  }
} // MultiplierSkill
```

Figure 2: Code de la compétence de multiplication

1.2.2 Lancement de l'application

Il suffit d'exécuter les programmes de la figure 4 sur les différentes machines utilisées pour l'application. L'ordre de lancement des agents n'a a priori aucune importance pour MAGIQUE, dans ce cas-ci il est cependant préférable de lancer l'agent `starter` en dernier, une fois que les autres agents sont créés, afin qu'il joue pleinement son rôle. Il faut donc exécuter (`magique.jar` étant supposé présent dans le `CLASSPATH`) :

▷ `java fr.lifl.magique.Start FactorialImp 4444`
ceci lance l'agent `fact`. L'utilisation de `fr.lifl.magique.Start`, et des

```
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
import java.util.*;
import java.math.BigInteger;
public class FactorialSkill extends MagiqueDefaultSkill {
  private static BigInteger DEUX = new BigInteger("2");
  private Vector theQuestions = new Vector();
  public FactorialSkill(Agent ag) { super(ag); }

  private Request firstAnsweredQuestion() {
    boolean found = false;
    Request question = new Request();
    while (!found) {
      for (Enumeration quest = theQuestions.elements();
        quest.hasMoreElements() && !found;) {
        question = (Request) quest.nextElement();
        found = isAnswerReceived(question);
      }
    }
    theQuestions.removeElement(question);
    return question;
  }

  public BigInteger factorielle(BigInteger n) {
    Enumeration team;
    BigInteger i;
    if (n.mod(DEUX).intValue()==0) i = BigInteger.ONE; else i = DEUX;
      team = getMyTeam().getMembers();
    // première phase :  distribution aux agents multiplieur de
    // mon équipe des multiplications de base
    for(BigInteger j=i;j.compareTo(n)==-1;j=j.add(DEUX)) {
      if (!team.hasMoreElements()) {
        team = getMyTeam().getMembers();
      }
      theQuestions.addElement(ask((String) team.nextElement(),"mult",j,
                                  j.add(BigInteger.ONE)));
    }
    // seconde phase :  distribution des multiplications
    // des résultats intermédiaires reçus
    while (theQuestions.size()>1) {
      if (!team.hasMoreElements()) {
                                  team = getMyTeam().getMembers();
      }
      theQuestions.addElement(ask((String) team.nextElement(),"mult",
                                  (BigInteger) returnValue(firstAnsweredQuestion()),
                                  (BigInteger) returnValue(firstAnsweredQuestion())));
    }
    Request quest = firstAnsweredQuestion();
    return (BigInteger) returnValue(quest);
  }
} // FactorialSkill
```

Figure 3: Code de la compétence de calcul de la factorielle de n.

```
import fr.lifl.magique.*;
public class MultiplierImp extends AbstractMagiqueMain {
  public void theRealMain (String[] args) {
    Agent agent = createAgent(args[0]); // args[0] = nom de l'agent
    agent.addSkill(new MultiplierSkill(agent));
    agent.connectToBoss("fact@"+args[1]); // args[1] = IP de hôte de "fact"
  }
} // MultiplierAgent

import fr.lifl.magique.*;
public class FactorialImp extends AbstractMagiqueMain {
  public void theRealMain (String[] args) {
    Agent agent = createAgent("fact");
    agent.addSkill(new FactorialSkill(agent));
  }
} // FactorialAgent

import fr.lifl.magique.*;
import java.math.BigInteger;
public class StarterImp extends AbstractMagiqueMain {
  public void theRealMain (String[] args) {
    Agent agent = createAgent("starter");
    agent.connectTo("fact@"+args[0]); // args[0] = IP de hôte de "fact"
    // args[1] = valeur de n pour calcul de n!
    S.o.p(args[1]+"!  = "
          +agent.askNow("fact@"+args[0],"factorielle",new BigInteger(args[1])));
  }
} // GoAgent
```

Figure 4: Les agents impliqués dans l'application *factorielle*

méthodes `theRealMain` des fichiers `*Imp`, permet de masquer le lancement de la plate-forme MAGIQUE, le paramètre `4444` indique le port utilisé par la plate-forme.

▷ `java fr.lifl.magique.Start MultiplierImp PORT nom fact_host:4444`
où

PORT est le numéro de port de la plate-forme de l'agent multiplicateur, si vous voulez tester toute l'application sur une seule machine, il faut bien sûr mettre des numéros de port tous différents,

nom est le nom de l'agent multiplicateur

fact_host est le numéro IP (ou le nom DNS) de la machine hôte de la plate-forme de `fact`.

▷ `java fr.lifl.magique.Start StarterImp PORT fact_host:4444 n`
où n est le nombre dont vous calculez la factorielle. Mettre donc 10 pour répondre exactement au cachier des charges (mais pourquoi s'arrêter là ?).

Remarques.

▷ En prenant une valeur de n assez grande afin que le calcul soit assez long, 100 par exemple, il est possible de vérifier la prise en compte de l'arrivée de nouveaux agents dans le calcul en cours. Il suffit de lancer les agents, y compris

`starter`, comme mentionné ci-dessus, puis une fois le calcul commencé (et sans trop tarder) de lancer un **nouvel** agent multiplicateur, on constate alors que celui-ci est immédiatement pris en compte et sollicité pour participer au calcul.

▷ Vous pouvez aussi tuer un des agents multiplicateurs par un CTRL-C des plus brutal (même si il serait préférable d'utiliser les compétences prévues dans MAGIQUE pour la déconnection ou la mort d'un agent), ceci a pour effet de stopper les calculs puisqu'il manque des résultats... cependant si vous relancer ce même agent (avec le même nom), vous voyez le calcul reprendre (ne vous occupez pas des messages d'exception dans la fenêtre de `fact`).

1.3 Conclusion

Nous avons présenté notre solution au "toy problem" *factorielle* en utilisant MAGIQUE ses modèles et son API. Plus largement que le sujet, la solution proposée permet de calculer $n!$ pour tout n et avec un nombre d'agent multiplicateur quelconque.
Cet exemple, tout en étant simple, permet de présenter l'essentiel des caractéristiques de base de la programmation avec MAGIQUE : l'acquisition de compétences, les différents types de requêtes (`perform`, `ask` asynchrone et `askNow` synchrone) ainsi que les relations hiérarchiques (`connectToBoss`) ou non (`connectTo`). Comme on peut le constater, la mise en œuvre est simple et naturelle si l'on connaît déjà le langage JAVA.

References

[BM97] N.E. Bensaid and P. Mathieu. A hybrid and hierarchical multi-agent architecture model. In *Proceedings of PAAM'97*, pages 145–155, 1997.

[MRS02a] P. Mathieu, JC. Routier, and Y. Secq. Principles for dynamic multi-agent organizations. In *Proceedings of the PRIMA2002 Conference*, LNCS, 2002.

[MRS02b] P. Mathieu, JC. Routier, and Y. Secq. Using agents to build a distributed calculus framework. 1(2):197–208, 2002.

[RM02] JC. Routier and P. Mathieu. A multi-agent approach to co-operative work. In *Proceedings of the CADUI'02 Conference*, pages 367–80, 2002.

[RMS01] JC. Routier, P. Mathieu, and Y. Secq. Dynamic skill learning: A support to agent evolution. In *Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems*, pages 25–32, 2001.