KLAVA : un Package Java pour Programmer des Applications Distribuées avec Mobilité de Code

Pierre Colot

Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur

Résumé KLAVA est un middleware conçu pour programmer des applications distribuées qui permettent la mobilité du code. Il se base sur le modèle Linda, un modèle de communication entre processus concurrents. Cet article présente le modèle Linda et ses adaptations pour l'environnement distribué, le fonctionnement de KLAVA ainsi que ses différentes classes Java et leurs méthodes, et une comparaison (sous un angle particulier : le load balancing) à un autre middleware, LIME, conçu dans le même but, basé aussi sur le modèle Linda, et dont une implémentation a été également réalisée en Java.

1 Introduction

Les applications distribuées utilisent des ressources et des services distants via un réseau. Les *mobiles hosts*, comme les PDA et laptops, sont des hôtes qui se déplacent et modifient la topologie du réseau. En conséquence, les applications distribuées doivent être capables de réagir dynamiquement aux changements de leur environnement et de se situer par rapport aux autres composantes du réseau. Des applications répondant à de telles exigences sont appellées des *network-aware applications*.

Les chercheurs se sont penchés également sur un nouveau modèle d'interaction entre client et serveur : la mobilité de code. Dans ce modèle, des programmes, appelés *mobile agents*, sont envoyés vers des sites distants et exécutés à leur arrivée. La mobilité de code avait des avantages que le modèle client-serveur traditionnel n'offrait pas : une meilleure utilisation de la capacité du réseau (les machines ayants des charges égales), et une diminution du trafic par une minimisation des interations à travers le réseau.

Nous avons donc d'une part les *mobile hosts* qui sont une forme de *physical mobility*, et d'autre part les *mobile agents* qui sont une forme de *logical mobility*. Les *mobile agents* peuvent être définis comme des processus capables de se déplacer d'un hôte à un autre en préservant leur code et leur état d'exécution.

Le Modèle Linda

KLAVA et LIME, qui seront détaillés plus tard, sont deux middlewares conçus pour programmer des applications distribuées qui permettent la mobilité du code. Ils se basent tous les deux sur le modèle Linda. La suite de cette section sera consacrée aux principes de base qui articulent ce modèle.

Linda est un modèle de communication entre processus concurrents. Dans Linda, les processus concurrents partagent un *tuple space*, c'est-à-dire un ensemble d'éléments (les *tuples*), qui contiennent des données élémentaires. Des primitives (*in(p)*, *rd(p)*, *out(t)*) permettent aux processus d'accéder au tuple space. Les *tuples* sont anonymes : ils sont choisis par un processus grâce à un mécanisme de *pattern matching* qui associe la structure d'un tuple (son *template* ou son *pattern*) à un tuple du tuple space.

- in(p) cherche un tuple t dans le tuple space qui correspond par pattern matching au tuple p. Si t est trouvé, il est retiré du tuple space et les valeurs correspondantes de t sont assignées au tuple p. Sinon, l'opération est suspendue (blocage) jusqu'à ce que t soit trouvé.
- rd(p) : même comportement que in(t) sauf que t n'est pas retiré du tuple space (read).
- out(t) ajoute le tuple t au tuple space.

C'est le tuple p qui est souvent appelé *pattern*. Les champs (*fields*) d'un tuple sont soit *actual* soit *formal*. Par exemple, le premier field du tuple p = <coucou, ?entier, ?caractere> est actual et les deux autres sont formals. Un tuple *pattern* peut contenir des champs actuals ou formals mais un tuple d'un tuple space ne contient que des actuals. Le tuple t = <coucou, 2, x> correspond par pattern matching au tuple p. Après le pattern matching, p prend les valeurs de t.

Initialement conçu pour fonctionner sur une machine isolée, le modèle Linda a été adapté et adopté pour les systèmes distribués.

Linda a été adopté car le caractère asynchrone des communications entre processus était une condition *sine qua non* pour un environnement distribué dans lequel il peut y avoir des problèmes de bande passante et des déconnexions fréquentes. Prenons un exemple : un utilisateur dépose un tuple dans le tuple space, puis se déconnecte du réseau. Le tuple reste dans le tuple space. Un autre utilisateur se connecte et récupère le tuple. En examinant cela, on voit que Linda permet donc trois choses :

- *time uncoupling* : pour s'échanger des informations, l'expéditeur et le destinataire n'ont pas besoin d'être connectés au réseau en même temps.
- *destination uncoupling* : l'expéditeur ne connaît pas le destinataire et le destinataire ne connaît pas l'expéditeur.
- *space uncoupling* : les processus ont accès à tous les tuples via *une seule et même* interface (celle du tuple space), et non plusieurs comme dans d'autres cadres de communication client-serveur.

Lors de l'adaptation de Linda à un environnement distribué, les tuple spaces sont devenus des *first class objects* (ils peuvent être obtenus comme résultat d'une expression et utilisés comme field d'un tuple), ils peuvent être créés dynamiquement et peuvent être structurés de façon hiérarchique. La structure hiérarchique des tuple spaces a été considérée de différentes manières selon les différentes implémentations de Linda, comme nous pourrons le voir en comparant KLAVA à LIME.

2 Klava

Cette section présente d'abord une introduction sur le fonctionnement de KLAVA, ensuite ses différentes classes Java et leurs méthodes, et enfin une remarque sur sa mobilité de code.

2.1 Introduction

KLAVA est basé sur le concept de *tuple space*. KLAVA manipule plusieurs tuple spaces, chacun de ces tuple spaces est placé dans un *node*, les nodes sont regroupés dans un *net*. Chaque node peut être adressé grâce à sa *locality*, il contient un et un seul tuple space ainsi que des *processes* qui travaillent sur ce tuple space. Les processes s'exécutent concurrentiellement, comme des threads en java. Ce sont des first class objects, donc ils peuvent être contenus dans des tuples et déplacés d'un node à l'autre. Ils peuvent exécuter des opérations (les primitives Linda) sur les tuple spaces et les nodes (par exemple envoyer un process s'exécuter sur un autre node). Un mécanisme de *pattern matching* est utilisé vu que les tuples sont anonymes.

2.2 Mécanismes et Classes Java

Tuples Un tuple est une séquence de fields, qui peut être soit *actual* soit *formal*. Un field actual contient une valeur (expression, process, locality) tandis qu'un field formal est une variable d'un certain type, mais de valeur indéterminée.

Voici une illustration de la classe `Tuple` :

```
Vector v = new Vector();
v.addelement(o1);
v.addelement(o2);
v.addelement(o3);
Tuple t = new Tuple(v);
```

Pour créer le même tuple, on aurait pu faire :

```
Tuple t = new Tuple(o1,o2,o3);
```

L'interface `TupleItem` est utilisée pour manipuler les fields.

```
public interface TupleItem extends java.io.Serialisable {
public boolean isFormal();
public void setValue(Object o);
public boolean equals(Object o);
}
```

Le package KLAVA fournit des classes pour les types standards :
`KString`, `KInteger`, `KBoolean`, `KVector`.

Pattern Matching Quand un process veut sélectionner un tuple dans le tuple space, un mécanisme de pattern matching entre en jeu. La sélection se fait suivant les règles suivantes :

1. les deux tuples doivent avoir le même nombre de fields.
2. les fields doivent correspondre entre eux : un actual du *pattern* correspond à un actual qui a une valeur identique, un formal du *pattern* d'un certain type doit correspondre à un actual qui a une valeur de ce type.

Les méthode de `TupleItem` sont employées ici : `isFormal()` parle "de lui-même", `equals()` vérifie si deux fields actual ont la même valeur. Après le pattern matching, le tuple prend les valeurs du tuple auquel il a correspondu grâce à la méthode `setValue(Object o)`.

```
KString s = new KString();
KInteger i = new KInteger();
Tuple t1 = new Tuple(s,i);
Tuple t2 = new Tuple(new KString("HELLO"),new KInteger("10"));
t2.match(t1);
System.out.println("s is now : " + s);
System.out.println("i is now : " + i);
```

Tuple Spaces La classe `TupleSpace` inclut des méthodes pour placer et retirer des tuples dans un tuple space.

```
public boolean in(Tuple p)
public boolean read(Tuple p)
public void out(Tuple t)
```

`public boolean read_nb(Tuple p)` et `public boolean in_nb(Tuple p)` se comportent comme read(p) et in(p) mais si un tuple correspondant à p n'est pas trouvé, l'opération renvoie `false`.

Les communications réseau peuvent être lentes et, de plus, l'absence d'un tuple correspondant dans le tuple space peut bloquer un processus. KLAVA propose donc deux méthodes supplémentaires :

```
public boolean read(Tuple t, long TimeOut)
throws KlavaOutException

public boolean in(Tuple t, long TimeOut)
throws KlavaOutException
```

Elles se comportent comme read(t) et in(t) mais si le timeout est écoulé avant que l'opération trouve un tuple, une exception `KlavaOutException` se déclenche.

Voici un exemple d'utilisation de l'objet `TupleSpace` :

```
TupleSpace TS = new TupleSpace();
KString s = new KString();
KInteger i = new KInterger();
```

```
Tuple t = new Tuple(s, new KInterger(10));
TS.out(new Tuple(new KString("hello"), new KInteger(10)));
TS.out(new Tuple(new KInteger(10));
TS.in(t); \\ retire le premier tuple
TS.read(new Tuple(i)); \\lit le second
```

Localities La *localité* est le moyen par lequel un process peut accéder aux nodes (et donc aux tuple spaces) distants. Il y a deux types de localités :

- les localités physiques : l'identifiant d'un node sur le net. Il ne s'agit pas de l'adresse IP, mais d'un nom que le node utilise pour s'enregistrer au net. Par ailleurs, il peut y avoir deux nodes sur la même machine.
- les localités logiques : un nom symbolique, un alias pour chaque node. Ces noms ont un portée limitée au node sur lequel ils sont considérés.

Quant aux classes java, nous avons une classe abstraite `Locality` qui implémente l'interface `TupleItem`. Les classe `PhysicalLocality` et `LogicalLocality` sont dérivées de cette classe.

Prenons un exemple où une localité est insérée dans un tuple : un processus veut se référer à un tuple space distant et cherche dans le tuple space local la référence (localité) du node de ce tuple space distant.

```
TupleSpace TS = new TupleSpace();
PhysicalLocality pl = new PhysicalLocality();
Tuple p = new Tuple("next node", pl);
TS.in(p);
TupleSpace TSdistant = new TupleSpace(pl);
```

Si le tuple space contient un tuple contenant une logical locality, un mécanisme appelé l'*allocation environment* associe les localités logiques aux localités physiques. Ce mécanisme joue également un rôle si c'est un processus qu'on insère dans un tuple.

Nodes Le node contient un et un seul tuple space. Il est l'endroit où peuvent s'exécuter les processus. Un objet `Node` contient les méthodes qui permettent d'accéder au tuple space :

```
public void out(Tuple t, Locality l)
public boolean in(Tuple t, Locality l);
public void read(Tuple t, Locality l);
```

La méthode out(t,l) place le tuple t dans le tuple space localisé au node ayant la localité l. Pour plus de facilité, KLAVA prévoit des versions surchargées pour qu'un tuple ne doive pas être explicitement construit :

```
public void out(TupleItem t1, TupleItem t2, Locality l)
```

La classe Node founit aussi les méthodes :

```
public void eval(KlavaProcess p, Locality loc)
throws KlavaException

public Locality newloc()
throws KlavaException
```

- *eval(pr, l)* envoie un processus *pr* s'exécuter sur le node *l*
- *newloc()* crée un nouveau node et renvoie sa physical locality.

Chaque node a deux champs : `self` (de la classe `LogicalLocality`) et `here` (de la classe `PhysicalLocality`) et dispose , comme nous l'avons déjà vu, d'une fonction partielle appellée *allocation environment* qui associe les localités physiques à des localités logiques. En particulier, `self` est associé à `here`.

Les nodes communiquent entre eux par des messages (à travers des flux connectés à des sockets), implémentés par la classe `NodeMessage`.

Processes Les processus KLAVA sont implémentés dans une classe héritant de la classe abstraite `KlavaProcess` qui définit une méthode `execute()` (similaire à la méthode `run` des threads Java). KLAVA offre aussi les méthodes d'accès aux tuples, ce sont les mêmes méthodes que celles présentées pour le node.

Considérons le cas spécial où un processus est inséré dans un tuple et placé dans un tuple space à l'aide de *out(pr, loc)* pour mettre en évidence la différence avec l'opération *eval(pr, loc)*, mis à part que *eval* déclenche automatiquement l'exécution de *p* sans le placer dans le tuple space distant. L'opération *eval* a une portée dynamique pour les localités : *p* va s'exécuter en tenant compte de l'*allocation environnement* du site de destination. Par exemple, `self`, qui est, rappelons-le, une logical locality, se référera donc bien au site sur lequel il est en train de s'exécuter (donc le site distant). Par contre, *out* a une portée statique : si on déclenche *p* sur le site distant, il va s'exécuter en tenant compte de l'*allocation environnement* d'origine. `self` se référera donc au site d'origine...

Nets L'objet `Net` garde les physical localities de tous les nodes qui composent le net KLAVA. Il coordonne les nodes entre eux et leur permet de communiquer. Il n'y a qu'un seul object `Net` pour un net.

Quand un node veut s'inscrire sur le net, l'objet `Net` crée un `NodeHandler`, un thread qui va enregistrer le node au net, garder une connexion constante avec ce node et s'occuper du transfert de messages entre ce node et d'autres.

2.3 Mobilité de Code avec KLAVA

Un processus peut être envoyé dans un message et exécuté sur un site distant, sur lequel la classe de ce processus (son code) peut être inconnu. Pour ce problème, il existe deux solutions possibles :

1. *automatic approach* : les classes nécessaires sont rassemblées et envoyées d'un seul coup avec le processus.

2. *on demand approach* : au moment où il se rend compte qu'il manque une classe sur le site distant, elle est récupérée sur le site d'origine.

KLAVA prend la première solution car elle correspond mieux au paradigme du *mobile agent* : un agent est autonome, il a avec lui tout ce dont il a besoin pour s'exécuter sur un site distant. Par ailleurs, avec la deuxième solution, la connexion entre les deux noeuds doit être maintenue tant que le processus n'a pas terminé son exécution, au cas où le site distant se rend compte qu'il ne dispose pas de toutes les classes du processus. Bien sûr, les classes communes à tous les nodes ne doivent pas être envoyées (par exemple, les classes java et KLAVA).

Quand un processus arrive avec ses classes, le node ajoute ces classes à la table du `NodeClassLoader`. Les noms des classes utilisées par ce processus sont retrouvés grâce à la *java reflection API*. Une fois que les noms de ces classes sont connus, leur code est rapatrié depuis le node d'où le processus provient dans la table du `NodeClassLoader`.

Il y a deux sortes de mobilités :

1. *strong mobility* : un *mobile agent* peut commencer à s'exécuter sur un site, migrer vers un autre emplacement, puis reprendre son exécution là où il était avant la migration.
2. *weak mobility* : après migration, les *mobile agents* reprennent leur exécution depuis le début.

Java ne permet pas de mémoriser l'état d'exécution du processus (le stack), donc ne permet que la weak mobility.

3 Lime

Cette section n'a pas la prétention de présenter LIME en long et en large, elle consiste en une courte introduction sur LIME et une description concise de son fonctionnement, sans entrer dans le détail de son code.

3.1 Introduction

LIME, tout comme KLAVA, se base sur le modèle Linda et a été implémenté en Java. Le tuple space de Linda se retrouve ici divisé en différents *interface tuple spaces*, associés de façon permanente avec un *mobile agent*. Ces tuple spaces forment, quand ils sont tous fusionnés, un *transiently shared tuple space* ("transient" car le contenu varie selon la *migration* des agents) auquel les agents ont accès de façon transparente depuis leur ITS.

3.2 Principes de Fonctionnement

Interface Tuple Space L'*interface tuple space* (ITS) est à la fois une interface qui fournit les primitives (*in*, *rd* et *out*) de Linda et un ensemble qui contient les tuples que le mobile agent veut rendre accessibles aux autres agents connectés.

Des agents sont connectés s'ils s'exécutent sur le même hôte (les tuple spaces de ces agents forment un *host level tuple space*) ou sur deux hôtes différents qui sont connectés entre eux. La fusion des host-level tuple spaces de plusieurs hôtes forme un *federated tuple space*. Chaque agent a accès à l'ensemble des tuples du réseau via son ITS qui accède au host level tuple space (qui peut être vu comme l'ITS de l'hôte) et au federated tuple space (qui peut être vu comme l'ITS du réseau).

Exemple : soient deux agents A et B. A effectue un *out(t)* sur son ITS. Après cela, t est accessible à B grâce à la primitive *in(p)* (p étant un pattern corresonpondant à t) de l'ITS de B.

LIME permet aux agents de disposer de plusieurs ITS, dont certains seront privés. Pour accéder à ces ITS, on préfixe les primitives par le nom (T) de cet ITS : *T.out(t)*.

Engagement et Disengagement l'engagement et le disengagement sont les noms donnés aux opérations d'arrivée et de départ d'un mobile agent ou d'un mobile host. Chacune de ces deux opérations, qui regroupent en fait plusieurs opérations, est effectuée comme une seule opération atomique. Elles se déroulent en trois étapes :

1. la décomposition du transiently shared tuple space : chaque mobile agent n'accède plus qu'à son (ou ses) propres ITS. Le contenu des ITS n'est plus partagé.
2. l'ajout, le déplacement ou la suppession du mobile agent à un mobile host, ou la connexion ou la déconnexion d'un mobile host (et des mobile agents qu'il contient).
3. la recomposition du transiently shared tuple space : chaque mobile agent a de nouveau accès aux tuple spaces contenu dans les ITS des autres agents connectés.

Fine-grained Control Modifions légèrement notre exemple à deux agents : A effectue un *out(t)* sur son ITS, puis se déconnecte. Il emporte donc le contenu de son ITS avec lui. B n'y a donc pas accès, son *in(p)* se bloquera. Le tuple space de LIME ne peut donc pas, comme celui de Linda, être considéré comme un répertoire global de tous les tuples émis par les agents.

Pour résoudre ce problème, A devrait pouvoir spécifier que le tuple qu'il ajoute au tuple space devrait être placé dans l'ITS de B (*fine-grained control*). Nous revenons donc fatalement à la notion de *location* : un tuple peut être ajouté à l'ITS du mobile agent A grâce à l'opération *out(t)[A]*. Cette opération, qui effectue une double action (une création et un déplacement de tuple vers un autre ITS), est effectuée comme une seule opération atomique.

Cette solution engendre un autre problème : que fait *out(t)[A]* si A n'est pas connecté ? Le tuple reste momentanément dans l'ITS de l'émetteur jusqu'à ce que le destinataire A se connecte. Pour permettre à LIME de détecter les tuple en "séjour illégal" dans un ITS, ceux-ci contiennent des champs indiquant leur

ITS courant et celui où ils devraient être. Un *garbage collector* est prévu pour enlever les tuples dont on sait que le destinataire ne se connectera plus.

Reactive Programming LIME permet aussi à un code de s'exécuter en réaction à un événement, par exemple l'arrivée ou le départ d'un mobile agent, la connection ou la déconnection d'un mobile host, la disponibilité de données suite à l'apparition d'un agent, des changements dans la *quality of service*. Les opérations *in* et *read* de Linda permettaient déjà de réagir à certains de ces événements (mais pas tous). Mais comme ces opérations bloquent un thread par événement, mettre l'arrivée de beaucoup d'événements en écoute serait coûteux.

Dans LIME, les événements précités sont détectés par le *run time support* qui a été modélisé comme un agent omniprésent dans le réseau. Celui-ci transforme en tuples ces événements et les met dans son ITS.

En pratique, les agents utilisent le *reactive statement* : *reactsTo(s, p)* où s est le fragment de code à exécuter et p le template "événement" du tuple à trouver par pattern matching.

4 Comparaison entre Klava et Lime : le Load Balancing

Cette section comporte d'abord des remarques d'ordre général sur certaines différences entre KLAVA et LIME. Elles sont déterminantes dans la discussion qui va suivre sur le load balancing.

4.1 Remarques Préliminaires

Mobilité de code La mobilité de code est mise en oeuvre par la primitive *eval(pr, loc)* dans KLAVA et par l'*engagement* et le *disengagement* de mobile agents dans LIME. Tandis que LIME laisse la migration proprement dite de processus en dehors de son modèle, KLAVA l'incorpore, en fait une fonctionnalité intégrée au modèle.

Tuple spaces KLAVA ne propose pas comme LIME une interface d'accès à tous les tuples de façon transparente. Cela a l'avantage, d'après les concepteurs de KLAVA, de diminuer l'overhead. De plus, les processus peuvent, en connaissant la localité du node distant, soit accéder directement à ces tuples, soit aller s'exécuter sur le site distant. Le fait de déposer un tuple dans un tuple space déterminé n'affecte pas le *destination uncoupling* car ce n'est pas le destinataire que l'on doit préciser dans l'opération *out(t, loc)*.

Mobile agents Dans KLAVA, les processus ne sont pas associés à des tuples. En contrepartie, ils ont un accès immédiat aux tuples du tuple space du node sur lequel ils s'exécutent et peuvent, au besoin, changer de node. Avec LIME, les tuples se déplacent en même temps que l'agent car ils sont en permanence associés à cet agent. Donc, l'agent doit connaître le destinataire de ses tuples si il

veut laisser ses tuples sur le réseau avant de se déconnecter (on n'a plus de *destination uncoupling*). Par ailleurs, plus d'opérations (overhead) sont nécessaires pour déplacer un agent dans LIME que dans KLAVA. Avec KLAVA, le programmeur devrait faire "à la main" cette association entre l'agent et ses tuples si il la juge vraiment utile, car les tuples sur le site d'origine sont toujours accessibles aux agents s'exécutant sur un site distant à condition de connaître la localité d'origine, et au prix d'une augmentation de la tranmission des données sur le réseau.

Gestion des Événements La gestion des événements est plus avancée dans LIME que dans KLAVA. LIME réagit plus efficacement et à plus d'événements que KLAVA.

4.2 Discussion

La mobilité de code est plus facile dans KLAVA et constitue donc un avantage certain pour ce modèle. Aucun autre mécanisme de LIME ne permet de procurer la flexibilité procurée par *eval(pr, loc)*.

Le transiently shared tuple space est le concept de base de LIME. Les tuple spaces non partagés de KLAVA ne permettent pas une programmation aussi intuitive et souple. Par exemple, pour rechercher un tuple, un processus regarde d'abord dans le tuple space local puis cherche la localité d'un autre tuple et fait une recherche dans un autre tuple space... toutes ces actions doivent être écrites par le programmeur. Avec LIME, un simple *in(p)* suffit. Toutefois, si on regarde uniquement les modèles sous l'optique du load balancing, cela constitue plutôt un désavantage pour LIME à cause de l'overhead causé par la maintenance du transiently shared tuple space et à cause du fait que le tuple space n'est pas en lui-même un outil utile pour se renseigner sur la charge d'un CPU...

Les mobile agents sont abordés différemment selon qu'on utilise KLAVA ou LIME, mais cette différence de conception entre les mobile agents de KLAVA et LIME ne donne pas clairement un avantage à l'un ou l'autre pour la mise en oeuvre du load balancing.

Le reactive programming proposé par LIME est un avantage : il pourrait permettre de repérer de façon efficace des événements comme la surcharge d'un CPU. Pour compenser ce manque avec KLAVA, un programmeur devra lui-même évaluer la charge CPU d'une machine distante, par exemple avec la technique du *leaky bucket of credits* : soient plusieurs agents "travailleurs" (qui supporte la charge de travail), et un agent "régulateur" (qui gère l'allocation de la charge aux agents travailleurs). Plus le processeur utilisé par un agent travailleur est occupé, moins il envoie de crédits (messages) à l'agent régulateur. Donc, quand on aura besoin de lancer un nouveau processus, l'agent régulateur demandera à l'agent travailleur qui a envoyé le plus de credits de faire tourner ce nouveau processus.

5 Conclusion

KLAVA, tout comme LIME, est un middleware mûr pour la création d'applications mobiles dans un environnement distribué. Il offre un modèle simple et facile à comprendre, il épargne au programmeur la connaissance de tous les mécanismes sous-jacents des communications entre client et serveur, du transport de code d'un hôte à l'autre et de l'arrivée et du départ d'hôtes sur le réseau.

Il est basé sur le modèle Linda. Ce modèle sous-jacent fait de lui un modèle asynchrone, aux communications anonymes : il permet le space uncoupling, le time uncoupling et le destination uncoupling.

La notion de transiently shared tuple space de LIME apporte un niveau d'abstraction plus élevé au programmeur, mais elle est compensée dans KLAVA par l'intégration du déplacement de code dans le modèle et la diminution de l'overhead qu'aurait occasionné le partage des tuple spaces. Le seul avantage significatif de LIME est la gestion des événements par *reactive programming*, qui n'a pas été intégrée par les développeurs dans KLAVA, et qui évite au programmeur de devoir lui même trouver des moyens "intermédiaires" pour repérer ces événements et de programmer lui-même un agent pour repérer et réagir à chacun de ces événements.

Références

1. Lorenzo Bettini, Rocco De Nicola, Rosario Purgliese. *KLAVA : a Java Package for Distributed and Mobile Applications*, SP&E.
2. Lorenzo Bettini, Rocco De Nicola, GianLuigi Ferrari, Rosario Purgliese. *Mobile Appications in X-Klaim.*
3. Gian Pietro Picco, Amy L. Murphy, Gruia-Catalin Roman. *LIME : Linda Meets Mobility.*
4. Gian Pietro Picco, Amy L. Murphy, Gruia-Catalin Roman. *LIME : a Middleware for Physical and Logical Mobility.*