Université Joseph Fourier

Rapport de Master 2R
Mathématiques et Informatique

Spécialité Systèmes et Logiciels

Réseaux de Services Avancés

BEKHIEKH MOHAMMED

Encadré par : Stavros TRIPAKIS & Sergio YOVINE

Laboratoire d'accueil : Verimag

Table des Matières

Table de Figure

Chapitre 1

INTRODUCTION

1.1 Introduction :

Jini est une technologie orientée *service*, basée sur l'idée de réseau spontané et des fédération de services. Un service est l'unité fondamentale dans une application Jini, c'est une implémentation d'une interface que l'on peut considérer comme un 'contrat', le contrat défini strictement la façon par laquelle on peut utiliser le service. Un service n'est pas restreint aux entités logicielles, mais il peut être une entité matérielle et il peut inclure la capacité du réseau à étendre de matériel comme : imprimante, camera digitale, PDA, … En Jini, un service de recherche (LUS[1]) joue un rôle primordial dans son fonctionnement. C'est un répositoire qui sert à enregistrer des services, il agit comme un intermédiaire entre le programme Client et le programme Serveur. Un serveur qui a besoin de publier un service doit localiser un LUS en procédant à une étape de recherche. Une fois la localisation du LUS est connue, le Serveur envoie l'objet représentant le service. Puis le client, qui connaît l'interface peut effectuer une recherche de service pour récupérer cet objet, à ce moment le client peut utiliser le service, et selon le type de service publié, il va appeler des méthodes locales ou éloignées. Donc, une architecture Jini nous offre principalement trois opérations : découverte, adhésion et recherche.

La première opération nous permet d'entamer un processus de découverte pour localiser un LUS, il y a deux types de découverte : une découverte unicast, utilisée si la localisation du LUS est connue d'avance, l'autre est multicast utilisée dans le cas contraire.

La seconde opération nous permet de procéder à l'enregistrement de service auprès du LUS, un identificateur unique est ajouté au service qui permet de le distinguer des autres, le serveur doit récupérer le bail (lease). Un bail est la durée de vie du service, il nous permettra par la suite de renouveler ou d'annuler le service.

La troisième opération permet aux clients de lancer un processus de recherche de service, le client envoie au LUS une requête précisant le type et les attributs de service demandé. Selon la demande du client, le LUS pourrait répondre au client en lui adressant un copie de service demandé.

[1] LUS : en anglais Louukup Service , au long de ce rapport on utilisera le terme service de recherche pour sa traduction

1.2 Problématique :

Dans cette architecture, nous avons vu que on a trois opérations, elles permettent aux applications d'utiliser les services fournis. De plus, les services sont à enregistrer dans un répositoire. Une application ou un client doit s'adresser au répositoire pour avoir une copie du service – s'il existe un service répondant à sa demande -. Pour répondre aux exigences d'applications et leurs complexités qui demandent un grand assemblage de services, nous souhaitons étendre ces capacités par d'autre moyens efficaces et féconds. La ***composition*** de services est un pas naturel dans ce champ, elle permette de répondre aux besoins de clients. En négociant un processus d'assemblage, nous voulons répondre aux applications dans la cas où le service demandé n'existe pas. L'objectif est de rendre cette composition la plus automatique, correcte et optimale.

Nous voulons présenter le Serveur comme un *module* qui fourni un certain service, et le Client est un autre *module* qui utilise ce service. Nous souhaitons également et particulièrement voir et à la fois un module comme un producteur et un consommateur de service, la production de service est assurée par une opération d'*exportation* et la consommation est assurée par une opération d'*importation*. L'opération de composition consiste à prendre en entrée au moins deux modules et nous retourner, s'ils sont composables, un seul module composite. Ainsi, même si un service n'est pas offert par un seul module, il peut être réalisé par composition des services de plusieurs modules. Le module composite peut avoir toutes ou une partie des fonctionnalités de tous les modules de départ, comme il peut engendrer de nouveaux fonctionnalités. Ces fonctionnalités représentent son métier, c'est ce qu'il est visible de son extérieur, et nous voulons cacher les détails de réalisation.

Deux modules sont dit 'composables' si l'un des services exportés par l'un est importé par l'autre. L'objective est de tester parmi les modules disponibles ceux qui sont composables, et d'une manière automatique engendrer un nouveau module composite fournissant de nouveaux services, ou il fourni l'ensemble des services de déparrts. Les services exportés par le module composite sont des services distribués, c'est-à-dire la composition doit se faire sans perte de distribution. La composition doit être implicite, automatique, intelligente.

1.3 Travaux en rapport :

Beaucoup de travaux ont abordé le sujet de composition, que ce soit en orienté objet, orienté composant, ou en orienté service, on cite entre autres : [14] Abstract Semantics for module composition, [15] The Sahara model for service composition, [20] Separation of Concerns for Dependable Embdded Systems, [28] Component Composition Validation, [29] Composition for Component-Based Modeling , [30] The Ninja Architecture.

Les travaux les plus récents qui visent notre domaine sont fournis dans [16, 17, 18]. Dans ces travaux on peut remarquer les points suivants : la définition des contrats nécessitent un langage spécial basé sur un schéma XML, ce langage permet la précision des attributs du service, les méthodes et les évènements. De plus, il utilise des machines abstraites et pour chaque méthode , il essaye de tirer beaucoup d'informations: paramètres, invocation, pré-condition, post-condition, invariant, sécurité, dépendance…. un grand processus est à déclenché pour l'extraction du contrat à partir d'une classe et la restitutions de toutes les information citées ci-dessus.

Dans [35] il y a un travail qui emploi les transactions pour coordonner l'interaction fiable des services de Jini. C'est une sorte de composition de services Jini qui nécessite une bonne connaissance de Jini, elle manque d'automatisation et de génération automatique de l'implémentation.

1.4 Contribution :

Notre objectif est de faire un lien théorie pratique entre une technologie orientée service Jini [1,2,3] et la composition de modules présentée dans [8], dans lequel il y a une bonne formalisation du problème de composition. Le travail à réaliser consiste à mettre en œuvre une chaîne automatique pour composer de modules, cette chaîne a en entrée une spécification de modules disponibles et nous retourne en sortie une implémentation Jini du module composite. En termine ce rapport par un exemple illustratif.

1.5 Plan :

Pour arriver à notre objectif cité au-dessus, on va subdiviser ce rapport on trois chapitres. Dans ce paragraphe, on les citera brièvement:

Dans le chapitre deux, on va aborder la technologie utilisée : Jini. Ce chapitre commence par une petite introduction, qui va nous permettre d'avoir une idée très simple sur cette technologie. Puis dans le deuxième point on parlera de la plate forme de cette technologie et au troisième

point on discutera de son modèle de programmation : bail, évènements et transactions. Dans le quatrième point, on aborde en détail l'utilisation de cette technologie, en particulier, les trois processus nécessaires pour utiliser correctement cette architecture : processus de découverte, processus d'adhésion et processus de recherche. On finira ce chapitre, par le mode d'emploi : structure de client et de serveur ainsi qu'un exemple d'utilisation est donné pour mettre en évidence le mode de fonctionnement de la technologie.

Le deuxième chapitre, aborde le problème de composition, on va y entamer notre sujet. D'abord une introduction est donnée pour se préparer au sujet: illustration des modules qu'on veut utiliser dans nos programmes, les opérations qui nous permettent d'utiliser ces modules, la conception des modules et son implémentation, et son interaction avec le mode extérieur. Après, une définition du problème de composition de modules est donnée, une formalisation du problème est jugée importante et nécessaire, elle représente notre théorie utilisée. L'algorithme qu'on va utiliser et qui est divisé en trois étapes est illustré dans le paragraphe suivant. L'architecture et son implémentation sont mentionnées : un diagramme de cette implémentation est fourni. Une grammaire du langage d'entrée est donnée, et qui permet surtout de préciser comment cette solution sera utilisée. Un exemple d'utilisation permet de mettre en évidence chaque étape de développement et de clarifier le sujet. Ce chapitre sera fermé par la génération de code du module composite.

Dans le troisième chapitre, on va donner un exemple d'utilisation concret, c'est l'exemple d'une composition de deux modules: un module représentant une camera digitale, qui exporte des images, l'autre c'est un module représentant une imprimante qui va les imprimer.

En fin, la conclusion et les perspectives de ce rapport seront citées dans le dernier chapitre.

Chapitre 2

La Technologie JINI

2.1 Introduction [2]:

Jini est un système orienté service [1,2,3], les services sont y organisés dans des fédérations. Une fédération est définie comme un ensemble de services associés. Le but de Jini est de transformer le réseau en un outil flexible et facilement administré sur lequel les ressources peuvent être facilement trouvées. Le service est l'entité de base, qui peut prendre n'importe qu'elle forme (logiciel, matériel, …). Un système Jini est articulé autour de trois axes :

- Les clients ;
- Le (s) service de recherche (Lookup Service :LUS) ;
- Les services.

Le point central est service de recherche (LUS). C'est lui qui crée le lien entre les services et les clients. Nous aurons donc une structure en triangle avec pour le sommet le LUS et comme base un ensemble de clients et de services. Le schéma type est représenté par la figure suivante.

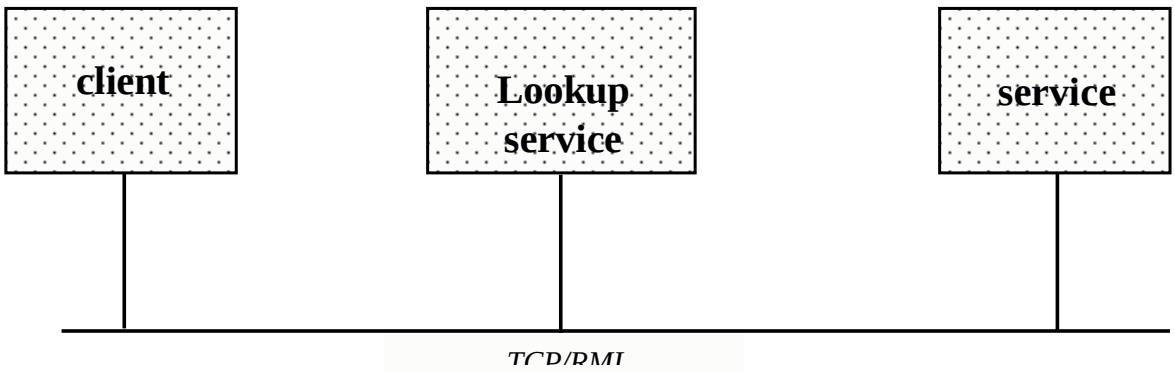


***Figure2.1** : Le triangle du Jini*

[2] La partie importante de ce chapitre est étudie dans le tutorial 10, et pour de plus amples informations voir [1,2,3,5,6,7]

2.2 Infrastructure :

Le système fonctionne autour de trois opérations :

- L'enregistrement d'un service auprès du LUS ;
- La recherche d'un service par le client ;
- La connexion du client et d'un service .

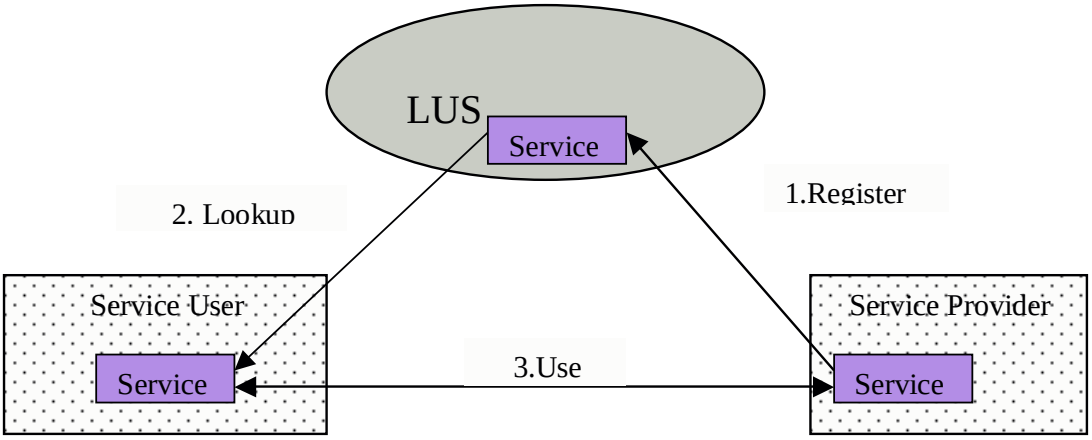


Figure 2.2 : *Les trois opérations fondamentales*

L'entité centrale d'un système Jini est le service de recherche LUS qui enregistre des services disponibles sur le réseau. C'est également le point principal de contact entre le système et les utilisateurs du système, on peut considérer un LUS comme un répositoire qui permet l'enregistrement et la localisation de services. Le service est un objet qui implémente une interface, le client ne peut appeler qui les méthodes définies dans l'interface, que l'on peut considérer comme un contrat entre eux. Dans la prochaine section on parlera, en détaille, des trois opérations nécessaires.

Le schéma suivant nous indique la structure de la technologie Jini, qui est fondée sur Java et RMI[3]. Elle fournit un modèle de programmation qui comporte : bail *(Laising)*, Evènements distribués*(Distributed Events) et* Transactions. Une infrastructure composée de processus de découverte (Discovery) et d'un processus de recherche (Lookup).

[3] RMI : Remote Methode Invocation, c'est la version Java de RPC

Services Jini	JavaSpace Transaction Manager …
Infrastructure	Discovery Security Lookup
Modèle de programmation	Leasing Distributed Events Transactions
Java	Java RMI Java VM

Table 2.1 : *Structure du Jini*

2.3 Modèle de programmation :

- **Bail:**

Il permet de gérer la durée de vie des objets distribués en étendant la notion de ramasse miettes aux environnements distribués. Chaque bail est négocié entre l'utilisateur et le fournisseur de service. Les baux sont exclusifs ou non-exclusifs. Les baux exclusifs assurent qu'un seul utilisateur à la fois peut utiliser la ressource, tandis que les baux non exclusifs permettent aux utilisateurs multiples de partager une ressource. Un système de bail a les caractéristiques suivantes :

- Il fourni une simple méthode basée sur le temps pour l'allocation et la réservation de ressources ;
- Il fourni d'une manière uniforme une méthode pour renouveler ou annuler le bail ;
- Il permet d'utiliser ce bail d'une manière claire et évidente.

- **Les transactions** *:*

Il s'agit d'une manière de regrouper un ensemble d'opérations impliquant plusieurs entités (clients ou services) afin d'assurer un certain degré d'atomicité dans l'exécution de ces opérations selon le principe du tout ou rien. Une transaction enveloppe une série d'opérations

d'un simple service ou de plusieurs services. Une transaction doit respecter les trois propriétés ,dites ACID, suivantes :

- Atomicité : toutes les opérations groupées dans une transaction terminent toutes avec succès ou toutes échouées.
- Consistance : une transaction doit préserver le système dans un état consistant.
- Isolation : un participant dans une transaction ne peut voir que les états intermédiaires entre les transactions.
- Durabilité : les résultats d'une transaction doivent être persistants de même que l'entité sur laquelle la transaction a été commitée.

- **Les événements distribués***:*

Etendent la notion d'événement Java aux environnements distribués, l'objectif est de permettre à un objet dans une JVM d'inscrire pour recevoir de notification sur l'occurrence d'un événement sur un objet dans autre JVM. Dans un tel système il y a deux parties :

- Un écouteur éloigné : qui a la possibilité d'inscrire son intérêt pour un certain type d'évènements auprès de générateur d'événement ;
- Un générateur éloigné : qui envoie un événement éloigné pour indiquer l'apparition de l'événement.

2.4 Processus de découverte :

Un fournisseur de service (resp client) doit localiser un LUS avant d'enregistrer (resp de récupérer) un service. Il existe deux types de découverte : le premier est unicast[4], on peut l'utiliser quand la localisation du LUS est connue, le Registrar[5] est récupéré directement. Alors, le second est multicast[6], on l'utilise dans le cas où la localisation du LUS est inconnue, nous devons diffuser une requête de recherche, en maintenant en état d'écoute des réponses. On a trois protocoles de découvertes :

- Le protocole multicast: utilisé par les entités qui souhaitent découvrir un LUS ;
- Le protocole d'annoncement : permet au LUS d'annoncer leurs présence ;
- Le protocole unicast : permet aux entités de communiquer avec un LUS spécifié.

[4] Unicast : une seule requête est envoyée au LUS
[5] Un Registrar est le représentant du LUS, récupéré par le Client et le Serveur
[6] Multicast : une requête est diffusée à tous les LUS

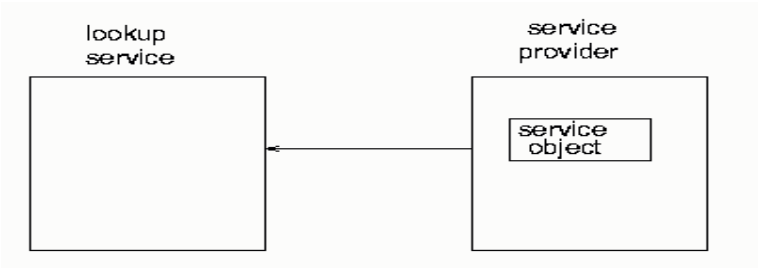
Dans un LUS, les services sont organisés dans des groupes, un groupe est connue par une chaîne de caractères, le groupe publique prend une chaîne de caractères vide, tous les LUS sont dans les groupes publiques.

2.5 Processus d'adhésion :

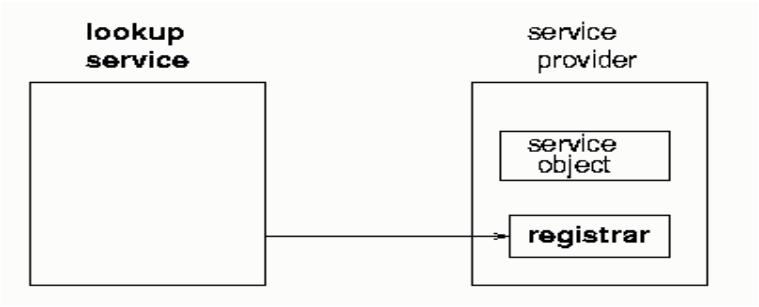
Un service qui veut s'enregistrer auprès du LUS doit suivre les étapes ci-dessous:

1. Le fournisseur de services envoie une requête pour localiser un ou plusieurs LUS.
2. Le LUS répond par l'envoi d'un registrar.
3. Le fournisseur de services faite une copie de l'objet service, et l'envoie au LUS.

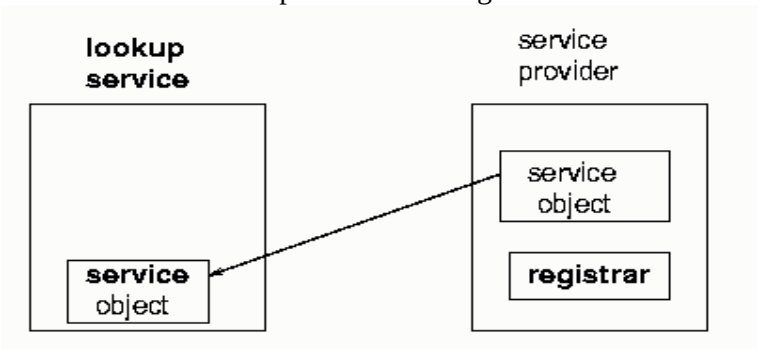
Les figures suivantes illustrent ce processus :



a: Localisation d'un LUS



b: Récupération du ''Registrar''



c: Envoie du Service

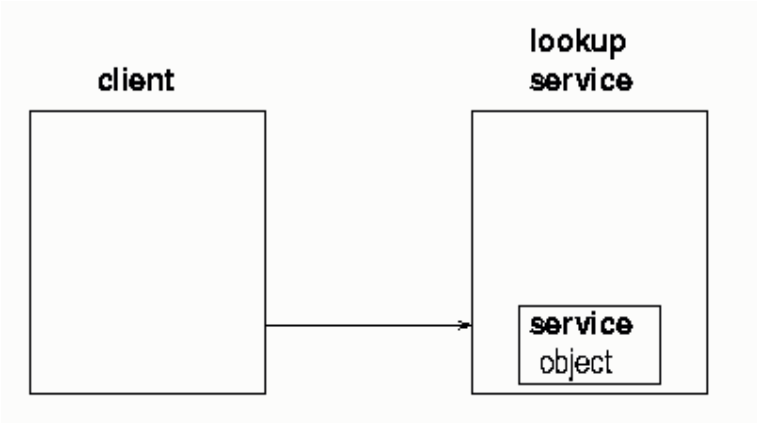
Figure 2.3 :*Processus d'adhésion*

2.6 Processus de Recherche :

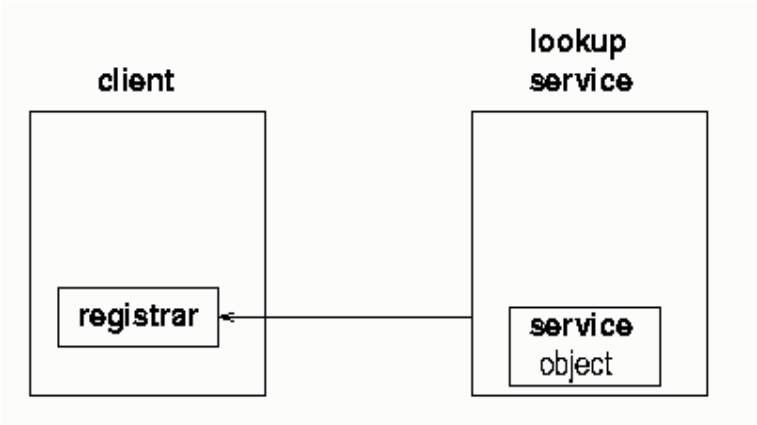
Les clients peuvent localiser les services en suivant les étapes suivantes:

1. Un client qui veut accéder à un service, envoie une ou plusieurs requêtes au LUS.
2. Les LUS qui sont en attente de requêtes en écoutant un port défini, peuvent envoyer leur représentant au client.
3. Le client utilise ce représentant pour demander le service.
4. Le LUS envoie une copie du service au client.

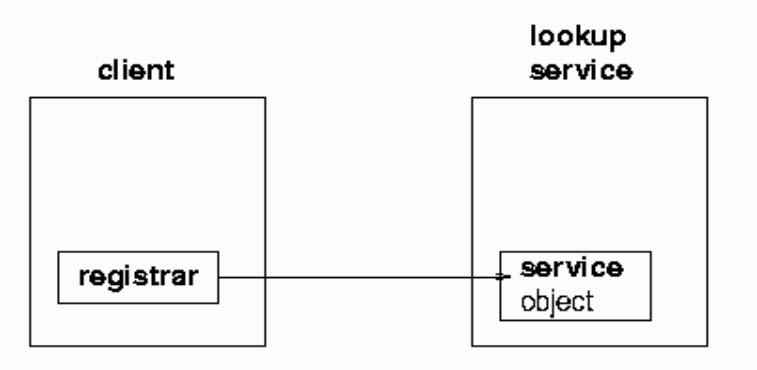
Les figures suivantes illustrent ce processus (processus de recherche):



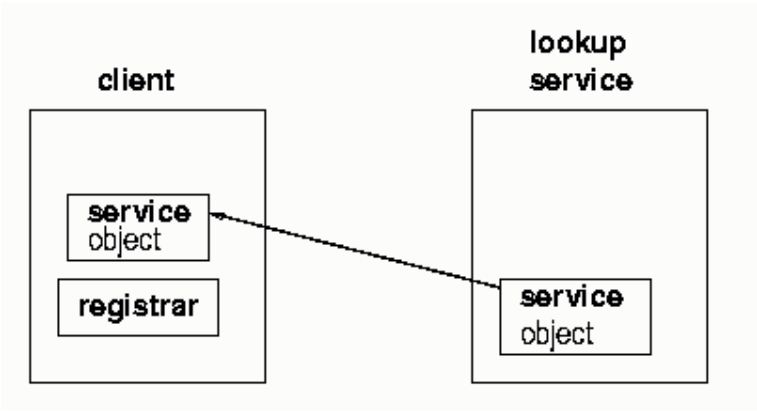
a: Localisation d'un LUS



b: Récupération du Registrar



c: Appel pour le Service



d : récupération de service

Figure 2.4 :*Processus de Recherche*

2.7 Structures du Client &Service :

On présente dans cette section les pseudo codes du client et du serveur, la prochaine section illustrera par un exemple la structure du client et du service. Les codes peuvent être variés, mais d'une manière générale sont similaires aux structures suivantes (données comme pseudo code) :

Un client avant de récupérer un représentant du LUS, et comme il est cité plus haut, doit effectuer une première étape, appelée étape de découverte, permettant au client d'avoir le moyen de communiquer avec le service de recherche et de récupérer son représentant. Puis une fois que le client possède le représentant, il peut lancer la recherche en fournissant une template, le résultat de cette étape est la réception d'une copie du service, et il ne reste au client qu'appeler ses méthodes. Le pseudo code suivant illustre ces étapes :

pseudo code: Client

Début

Préparation pour la découverte ;

Découvrir un LUS;

Préparation d'une template pour la recherche;

La recherche d'un service;

Appel du service;

Fin

Figure 2.5 : *Pseudo code client*

De sa part, le serveur ne peut pas enregistrer ses services qu'à après la récupération du représentant du LUS, il aura ce représentant à la fin de l'étape de découverte. Puis il créer l'item de son service et utilise le représentant du LUS pour enregistrer le service. Il reste après cette étapes de renouveler périodiquement le bail sinon il peut l'annuler. Le code suivant illustre ces étapes :

Pseudo code: Serveur

Début

Préparation pour la découverte;

Découvrir un LUS;

Créer un item pour le service;

Enregistrer le service;

Renouveler périodiquement le bail;

Fin

Figure 2.6 : *Pseudo code Serveur*

2.8 Exemple [7]:

Afin d'illustrer les structures du Client et du Serveur, nous présentons dans cette section un exemple d'utilisation. Il s'agit d'un programme Jini qui fournit un service « Claculator » défini par une interface Java, qui limite les opérations que le client peut les appeler. Une classe proxy[8] qui est l'implémentation cette interface doit être serializable ; le serveur localise un LUS

[7] Le code complet de cet exemple est donné dans l'annexe A, ici c'est seulement juste pour donner une idée

[8] Proxy est un représentant de service à envoyer aux clients

et y installe une copie de cette proxy, en définissant une Item. Une fois la proxy est présente dans le LUS, le client peut la récupérer, en définissant une requête Template.

- ***l'interface***

```
public interface Calculator{

        public double add(double x,double y);
        public int add(int x,int y);
        public double subtract(double x,double y);
        public int subtract(int x,int y);
        public double multiply(double x,double y);
        public int multiply(int x,int y);
        public double divide(double x,double y);
}
```

- ***La proxy:***

```
public class CalculatorImpl implements Calculator, Serializable{
      public CalculatorImpl(String name){

      }
}
```

- ***Le serveur:***

```
public class CalculatorServer implements ServiceIDListener[9] {
    public static void main(String argv[]) {
            new CalculatorServer();
            Object keepAlive = new Object();
            synchronized(keepAlive) {
            try {
                        keepAlive.wait();
            } catch(InterruptedException e) {}
            }
    }
    public CalculatorServer() {
      //Enrengistrement du service
    }
    public void serviceIDNotify(ServiceID serviceID) {
      System.out.println("Reception du service ID :" + serviceID.toString());
    }}
```

- ***Le Client:***

```
public class CalculatorClient {
    public static void main(String argv[]) {
      new CalculatorClient();
      try {
            Thread.currentThread().sleep(2*WAITFOR);
      } catch(java.lang.InterruptedException e) {}
    }
    public CalculatorClient() {
      //Récupération du service et affichage de l'interface
      }
}
```

[9] ServiceIDListener est une interface sert à attendre la notification du LUS

Chapitre 3

Les Modules

3.1 Introduction [10]:

On a vu dans le précédent chapitre que Jini est une architecture orientée service et comment les services sont créés et comment ces services sont enregistrés auprès d'un LUS,et comment un client peut les accéder et les utiliser tout en connaissant l'interface. Le client est le service sont d'accord sur cette interface, autrement dit cette interface est connue. D'ailleurs le client et le service doivent définir d'une manière très explicite la façon dont ils communiquent avec le service de recherche et sans écrire beaucoup de code pour arriver à cet objectif. Dans ce chapitre en libérera le programmeur de cette fastidieuse tache en fournissant des opérations simples et efficaces. Et on essaye d'éliminer la séparation entre ce qui est client de ce qui est serveur. En effet, notre idée est de faire des clients et des services des entités similaires, qui peuvent utiliser les mêmes opérations. Ces entités sont les ***modules***[11] : un module est un programme qui peut importer des services, il est dans ce cas client, comme il peut exporter des services, et il est dans ce cas serveur, l'avantage est qu'une entité peut à la fois être client et serveur, et elle peut utiliser des services pour fournir d'autres. En bref, un module est une entité capable d'utiliser deux opérations : importer et exporter.

3.2 Les Opérations :

On a motivé dans le précédent paragraphe la nécessité d'avoir les deux opérations importation et d'exportation. Dans ce paragraphe, on va illustrer comment ces opérations vont être implémentées. En effet, ces deux opérations sont simplement définies par deux interfaces. Le code qui nous permet de définir ces interfaces est donné dans les lignes suivantes :

```
package jini.module ;
 public interface Exportation{
        public void exports(ServiceItem item) throw java.rmi.RemoteException ;

 }

package module.jini ;
 public interface Importation{

        public ServiceItem imports(ServiceTemplate template)
                                throw java.rmi.RemoteException ;
}
```

[10] La grande partie de ce chapitre est tirée de [8]
[11] Inspiré de [8] Automated Module Composition

Ces deux opérations nous permettent d'envoyer et de récupérer de service de et vers le service de recherche, et ce quelque soit sa localisation. Un module qui appelle l'opération d'exportation, doit fournir en paramètre un service item qui sera enregistré auprès de service de recherche LUS, cette opération ne lui retournera rien. Le module appelant l'opération d'importation doit fournir comme paramètre une 'template' qui définit sa requête, et récupère l'item du service demandé.

3.3 Les modules :

Un *module*[12] est l'implémentation des interfaces *Importation* et *Exportation, qui sont* citées ci-dessus. Par conséquent, toute classe qui hérite de cette classe peut utiliser directement ces les opérations '*importe'* et '*exporte'*. Elle permettent ,sans avoir recours à écrire une dizaine de lignes de code, à envoyer et à récupérer le service. Ile suffit au module d'implémenter son interface métier avec Serializable et utilise les opérations d'importation et d'exportation s'il a besoin. En bref, tout module est une classe qui a la capacité d'utiliser les opération d'importe et d'exporte. La figure suivante illustre ce qui précède :

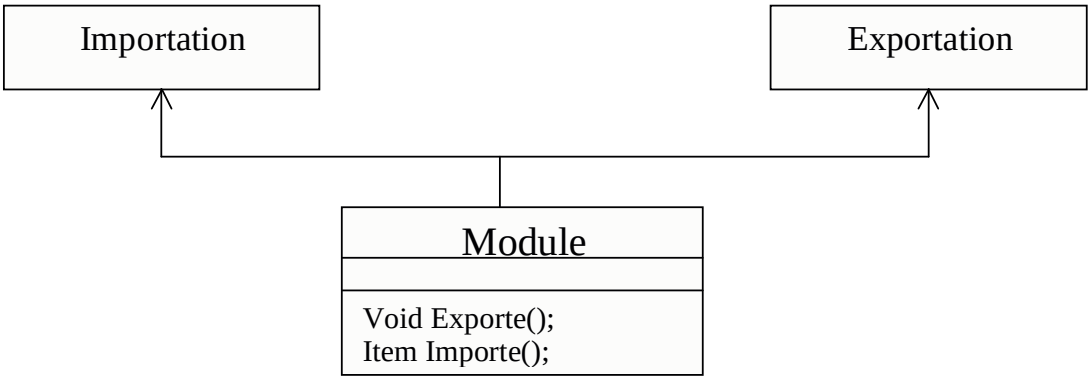


Figure 3.1 : *Diagramme de classe*

3.4 L'interaction avec LUS :

On a vu dans le chapitre précédent qu'un LUS est un répositoire de services, les clients et les serveurs doivent être capables de communiquer avec lui. Dans notre travail, on veut utiliser des modules définies dans la précédente section, nous souhaitons que l'interaction avec le LUS soit plus claire et évidente et ne nécessite pas beaucoup de code. Un module est un objet qui

[12] Le Code complet de la classe Module est donné en annexe B, il comporte l'implémentation des interfaces *Importation* et *Exportation*

consomme et produit de services, il a deux ports[13]: un port d'entrée qui lui permet l'importation de services et port de sortie lui permet l'exportation de services, le module M présenté dans la figure suivante, importe le service S1 par son port d'entrée et fournit un service S2 par son port de sortie. Il n'y a aucune restriction sur le nombre de services importés ou exportés, en effet un module peut appeler les opérations '*importe'* et '*exporte'* autant qu'il veut. Un service importé ou exporté est logé auprès du LUS :

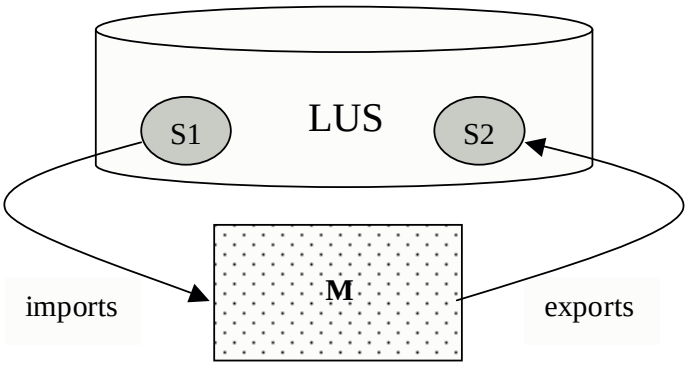


Figure 3.2 : *Interaction du module avec LUS*

Nos classes héritent de la classe Module, peuvent appeler les méthodes *exports* et *imports*[14]. Pour illustrer, nous donnons un exemple: nous voulons créer un service qui défini par l'interface *Calculator* (voir l'exemple suivant).

3.5 Exemple d'utilisation :

Nous donnons cet exemple pour mettre en évidence comment cette conception de module peut être utilisée. Il s'agit d'un service défini par une interface Calculator, le *module*[2] doit implémenter cette interface, Un module a les deux opérations '*imports'* et '*exports'*. Reste à noter qu'il doit de plus implémenter serializable.

```
//packages nécessaires

public class CalculatorImpl extends Module implements Calculator, Serializable{
//l'implémentation est à définir ici

public CalculatorImpl(){

exports(new ServiceItem(null, this, new Entry[]{new Name("Calculator")}));
}
```

[13] Un port dans ce contexte consiste à deux façades logiques qui signifient l'entrée est la sortie
[14] On utilise imports au lieu import, pour éviter la confusion avec import, qui est un mot réservé en Java
[9] Dorénavant, on appel module toute classe qui hérite de la classe Module

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

System.setSecurityManager(new java.rmi.RMISecurityManager());

new CalculatorImpl();

try{
        Thread.currentThread().sleep(10000);
}catch(java.lang.InterruptedException exc){

        exc.printStackTrace(System.err);

}
}
}
```

Pour exporter le service, il suffit pour tout programme d'appeler la méthode *exporte*, c'est la méthode héritée de la classe *Module*, elle accepte comme paramètre un ServiceItem.

3.6 Le problème MC :

Après avoir construit le modèle, qui définit clairement les modules et les opérations *import* et *export*. On va dans ce paragraphe entrer dans le cœur du sujet qui est la **composition** de modules, nous tenterons de donner un moyen simple et facile pour composer de modules disponibles. Le schéma suivant illustre bien notre problématique (figure 3.3). Dans ce schéma, on a deux modules et nous voulons avoir automatiquement le module composite:

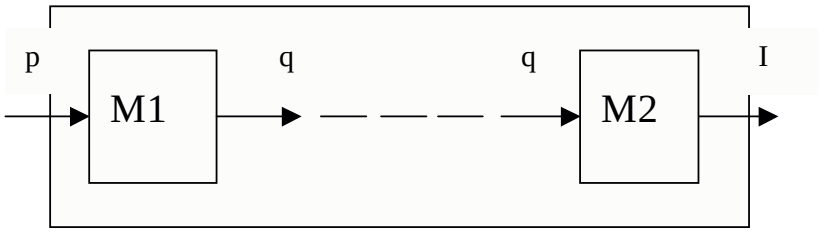


Figure 3.3 : *Un problème de composition*

Ce schéma représente un problème de composition de deux modules M1 et M2 : M1 est un module qui importe *p* et exporte *q*, et M2 est un module qui importe *q* et exporte *i*. Nous souhaitons obtenir un seul module M qui importe *p* et exporte *i*. On remarque que le service exporté par M1 est celui importé par M2. Cette composition est possible seulement si *q* exporté

par M1 est *compatible*[15] avec *q* importé par M2. L'opération de composition doit nous retourner un module qui ressemble au schéma suivant :

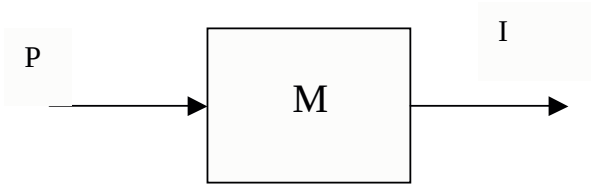


Figure 3.4 : *Module Cible*

3.7 Formalisation du Problème :

La formulation que on est en train de citer ici est tirer de [8]. Dans cette section on se contente de donner les grandes lignes de la formulation mathématique au problème *MC*[16], qui consiste à formuler le problème de composition de modules comme un programme linéaire.

Soit $\mathcal{M} = \{\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots\}$ l'ensemble de modules types. Le programme linéaire contient les variables et les contraintes suivantes :

- x_i $i = 0, \ldots n$, représente le nombre d'instances pour chacun de modules $\mathcal{A}_i$, soit $N_{inst}(\mathcal{A}_i) = [\ l_i, u_i\ ]$. Pour toute x_i on a la contrainte suivante :

  $$l_i \leq x_i \leq u_i \qquad (1)$$

  Si $u_i = \infty$ alors la contrainte précédente devient : $l_i \leq x_i$

- y_p π dans P_{in} représente le nombre total de copies d'un port d'entrée *p* d'un module $\mathcal{A}_i$, soit $f_{in}(\mathcal{A}_i)(p) = [\ a_p, b_p\ ]$. Chaque instance de $\mathcal{A}_i$ peut contribuer par a_p jusqu'à b_p copies de p. Nous avons pour chaque p la contrainte suivante :

  $$x_i.a_p \bullet y_p \bullet x_i.b_p \qquad (2)$$

  Si $b_p = \infty$ alors la contrainte précédente devient : $a_p \leq y_p$

[15] Une relation de compatibilité de services est nécessaire pour la composition de deux modules
[16] MC :Module Composition voir [8] pour de plus amples informations

- z_θ q dans Q_{out} représente le nombre total de copies d'un port de sortie q d'un module $\mathcal{A}_i$. soit f_{out} ($\mathcal{A}_i$) (q) = [a_q, b_q]. Toute instance de $\mathcal{A}_i$ peut contribuer a_q jusqu'à b_q copies de q. Nous avons pour chaque q la contrainte suivante :

$$\chi_i . a_\theta \bullet z_\theta \bullet \chi_i . b_\theta \qquad (3)$$

Si $b_q = \infty$ alors la contrainte précédente devient : $a_q \leq z_q$

- $w_{\pi,\theta}$,(p,q) dans C (La relation de compatibilité) représente le nombre de connections entre une copie *p* et une copie *q*. Nous accompagnons $w_{\pi,\theta}$ avec les variables y_p et z_θ les contraintes suivantes :

$$y_p = \sum_{(p,q)\in C} Wp,q \qquad z_q = \sum_{(p,q)\in C} Wp,q \qquad (4)$$

Cette contrainte assure l'existence d'une composition close.

En prenant en compte l'optimisation, nous pouvons ajouter une fonction objectif à optimiser sous les contraintes précédentes. Soient $c_i \geq 0$ le coût d'une instance de module $\mathcal{A}_i$, $c_p \geq 0$ le coût d'une copie d'un port *p*, $c_q \geq 0$ le coût d'une copie d'un port *q* , $c_{p,q} \geq 0$ le coût d'une connexion d'une copie *p* à une copie *q*. Ainsi la fonction objectif sera de la forme suivante :

$$f(\vec{x},\vec{y},\vec{z},\vec{w}) = \sum_{i=0,\ldots,n} CiXi + \sum_{p\in Pin} CpYp + \sum_{q\in Pout} CqZq + \sum_{q\in C} Cp,qWp,q$$

telles que $\vec{x},\vec{y},\vec{z},\vec{w}$ sont des vecteurs associés aux variables x_i, y_i, z_i, $w_{p,q}$
Le problème linéaire obtenue est le suivant :

$$\begin{cases} \textit{min } f(\vec{x},\vec{y},\vec{z},\vec{w}) \ ; \\ \textit{les contraintes : 1, 2, 3, 4 ;} \\ \textit{int } x_i, y_i, z_i, w_{p,q} \ ; \end{cases}$$

3.8 Algorithme du problème MC :

L'algorithme pour résoudre le problème MC comporte trois étapes :

1. étape une :

Ecrire dans un fichier qui décrit le problème MC: les modules disponibles qu'on souhaite composer et ainsi que le module cible qu'on veut obtenir. Créer ou définir le programme linéaire approprié.

On essaye de trouver la solution au problème linéaire défini précédemment. S'il n y a pas de solution au problème linéaire alors le problème MC ne peut pas être résolu. Sinon passer à l'étape suivante.

2. étape deux :

Possédant une solution au problème d'optimisation précédent, récupérer les valeurs de variables x_i, y_p, z_q, $w_{p,q}$, procéder à créer des instances de modules de ports et de connections en suivant la procédure qui suit :

créer x_i instances de *Ai* , pout tout port d'entrée de *Ai* il y aura en total y_p copies de *p* pour toutes les instances de *Ai*, pout tout port de sortie de *Ai* il y aura en total z_q copies de *q* pour toutes les instances de *Ai*. Assigner à chaque instance ses copies d'entrée et de sortie.

3. étape trois :

On construit le module cible comme suit:

Initialement tous les ports sont déconnectés, à partir d'une copie quelconque d'un port d'entrée p et faire une connexion avec un n'importe quelle copie libre d'un port de sortie q, répéter ça jusqu'à ce que tous les ports sont connectés.

3.9 Architecture des programmes:

L'algorithme précédent nous fournit la solution du problème MC. Dans cette section on va décrire une implémentation de cet algorithme qui nous permettra de créer un nouveau modules à partir d'autres qui sont déjà disponibles. Cette implémentation est décrite par le schéma suivant , il présente les différents composants que nous devons utiliser et les données échangées entre eux. En tête du diagramme , on trouve un fichier qui décrit le problème MC à réaliser. Un programme

prend en entrée ce fichier et qui envoie à un autre programme deux fichiers représentent les deux ensembles Pin et Pout : Pin est l'ensemble des ports d'entrée et Pout est l'ensemble des ports de sortie. Ce programme, à partir d'une définition claire de la relation de compatibilité, lui retournerait un fichier qui décrit les ports que l'on peut connecter. Une fois ce composant a l'information complète, il procédera à créer le programme linéaire. Un autre composant lit le fichier d'entrée pour donner une description claire du module cible. En utilisant un outil extérieur, le Solver a comme donnée le problème linéaire et nous retourne la solution au problème d'optimisation, en particulier des valeurs assignées aux variables x_i, y_p, z_q, $w_{p,q}$.

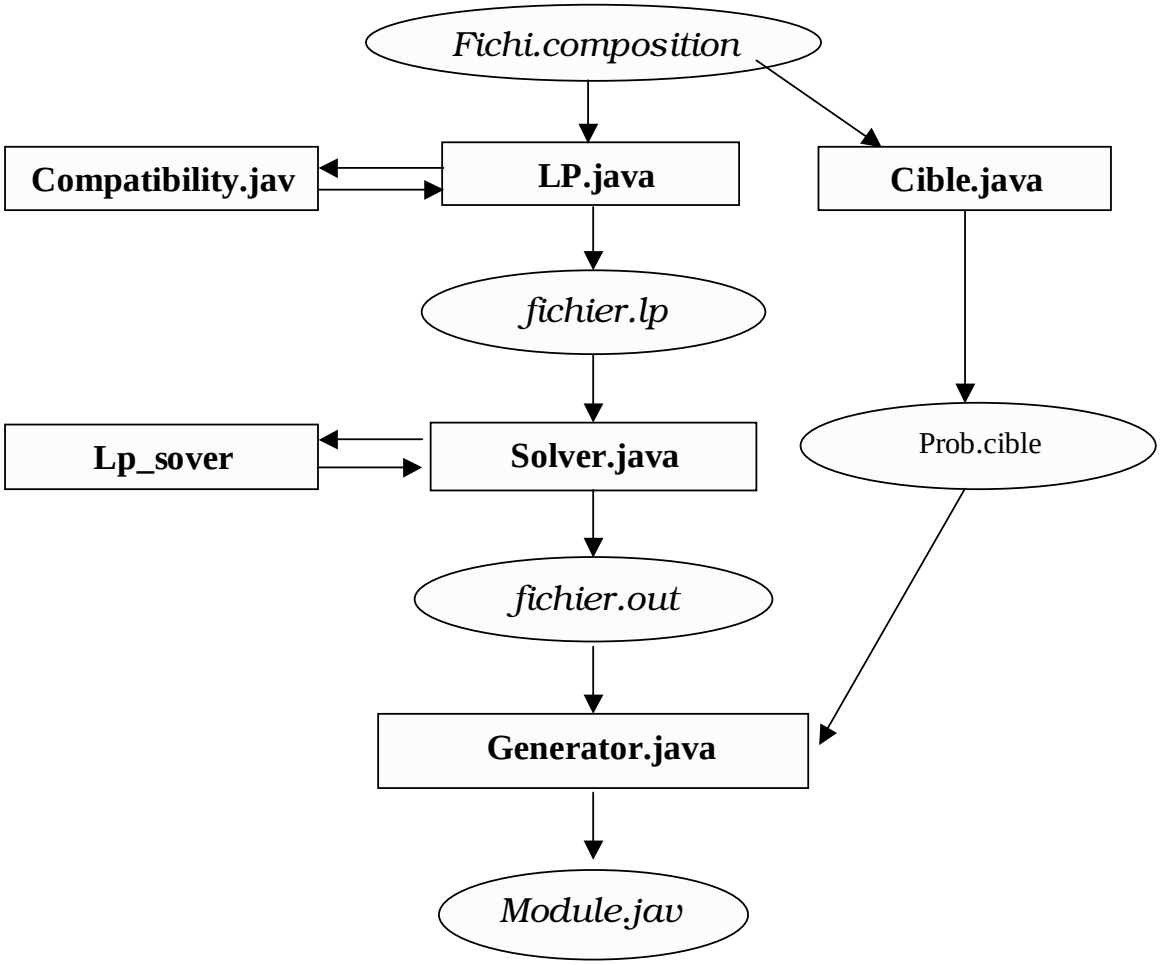


Figure 3.5 : *Architecture des programmes*

Une fois la solution au problème linéaire disponible, un composant n'a qu'à lire le fichier contenant cette solution et procéder à l'instanciation de modules et les ports importés et exportés

et connecter ces instances pour construire la structure interne invisible par l'extérieure. Pour qu'il puisse construire la partie extérieure, il doit appeler un autre composant qui lui donnera la définition de cette partie. Enfin, un programme sera imprimé dans un fichier de sortie qui représente le module demandé. Cette architecture est décrite dans le diagramme ci-dessus (voir la figure 3.5).

3.10 Implémentation :

Dans la précédente section on a présenté l'implémentation de l' algorithme de composition. On parlera dans cette section et d'une manière détaillée de l'implémentation de l'architecture précédente, cette implémentation suit les étapes citées ci-dessous : une grammaire sert à définir le fichier de départ, dans lequel on précise les modules disponibles à composer et le module cible. Un fichier de composition doit prendre avec précision la forme d'une grammaire, dont la structure est donnée au paragraphe suivant (voir figure 3.6), formulée avant d'entamer le traitement ; un programme prend ce fichier de composition et le transforme en un programme linéaire, lui aussi et à son tour appelle un programme pour définir la relation de compatibilité entre modules de départs. Le Solver résout le problème linéaire, le résultat est stocké dans un autre fichier. Le générateur va nous générer le code du module cible, qui doit appeler un programme qui lui fournira des informations précises du module cible. Dans le schéma les rectangle sont des programmes, les ellipses sont des fichiers et les flèches représentent le flux d'informations échangées entres ces programmes. L'implémentation va suivre cette logique, en va commencer cette chaîne, qui sert à nous définir un langage spécifique à nos besoins .

3.10.1 La Grammaire :

Le fichier d'entrée sert à décrire le problème de composition à réaliser. Sa structure nécessite la définition d'une grammaire qui doit être respectée pour chaque problème MC. Cette grammaire est la suivante constitue pour nous un langage, elle définie des règles qu'on doit respecter au moment de l'écriture du problème de composition et les mots en majuscule pour exprimer : Composition, Module, Input, Output, …. Ce langage peut répondre à nos problèmes de composition, l'objectif crucial de ce langage est d'éviter toute ambiguïté et d'exprimer d'une manière claire tout problème de composition, et elle permet un meilleur traitement par la suite, dans les prochaine étapes.

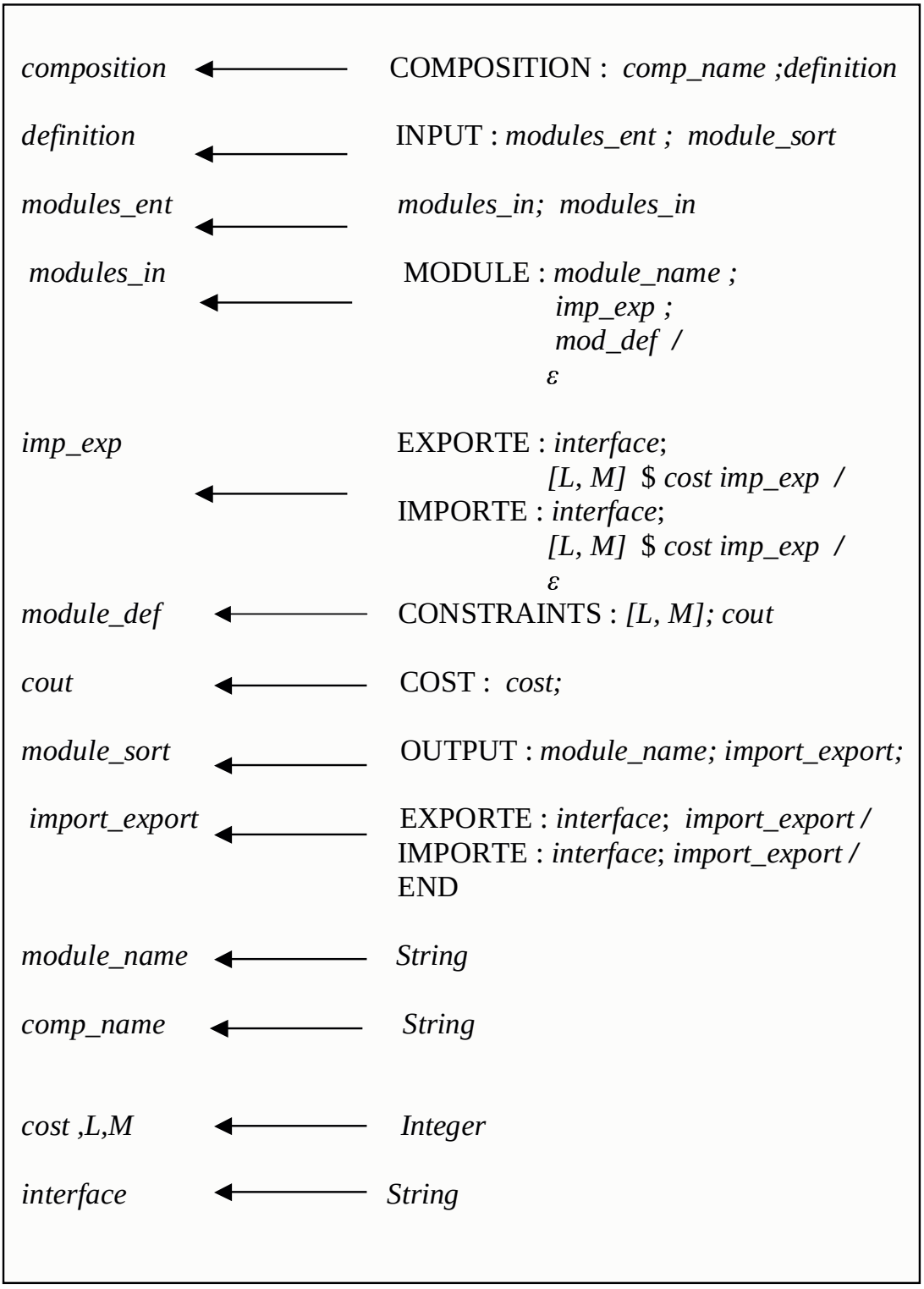


Figure 3.6 : *Grammaire du fichier d'entrée*

3.10.2 Exemple :

Dans cet exemple, nous volons montrer la structure du fichier de composition. Supposons qu'on a trois modules disponibles comme indiqué dans la figure suivante (figure 3.7 A) :

Le module M1 est un module qu'exporte deux services définis par leurs interfaces I et J, mais n'en importe aucun, le module M2 importe un service défini par I et exporte un autre service défini par P, le module M3 importe le service défini par J et exporte celui défini par Q.

A partir de ces module de départ, nous souhaitons avoir et d'une manière automatique un autre module comme il est défini dans la figure 3.7.B

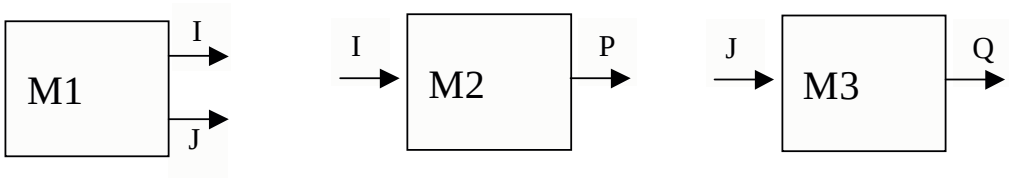


Figure 3.7 A : *Modules de départs*

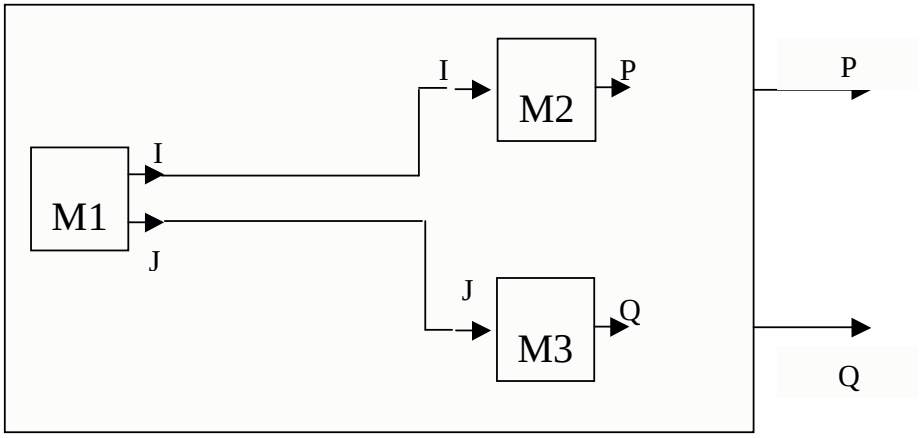


Figure 3.7 B : *Module Cible Possible*

Le module cible qui doit fournir deux services à la fois, a une structure interne similaire indiquée à la figure 3.7 B, mais cette dernière n'est pas connue à l'avance par le client. Cette structure en générale n'est pas unique et peut prendre plusieurs compositions différentes. Elle donné à titre indicatif.

3.10.3 Fichier d'entrée :

Le fichier suivant illustre une structure possible de fichier de composition, il respecte exactement la grammaire précédente. Il est divisé en deux parties, une dite d'entrée comporte la définition de modules de départ, et qui sont disponibles avant l'opération de composition. Chaque module de départ a un nom qui permet de le distinguer des autres, un certain nombre de ports d'entrée et de sortie, chaque port représente une interface, l'utilisateur peut imposer ces conditions, en donnant le nombre de copies qu'on peut utiliser pour chaque port qui est limité par une intervalle, à chaque copie on associe un cout. Comme pour les ports, le nombre de copies pour un module de départ est défini par une contrainte et un cout est ajouté pour chaque copie.

```
COMPOSITION: comp_M5
INPUT:

MODULE : m1;
EXPORTE: i; [2,5] $9
EXPORTE: j; [3,8] $2
CONSTRAINTS: [2,6]
COST: 5;

MODULE : m2;
IMPORTE: i;  [4,7] $4
EXPORTE: p; [2,5] $2
CONSTRAINTS: [4,5]
COST: 7;

MODULE : m3;
IMPORTE: j; [2,3] $1
EXPORTE: q;  [5,6] $6
CONSTRAINTS: [1,3]
COST: 9;

OUTPUT:
MODULE: M4;
EXPORTE: p;
EXPORTE: q;

END
```

Figure 3.8 : *Fichier de composition*

En fin, le module cible est défini tout simplement par ces ports d'entrée et de sortie, il est également à distinguer par un nom qui lui sera ajouté.

3.10.4 Traduction du fichier d'entrée :

Le fichier défini dans le paragraphe précédent et qui est strictement conforme à la grammaire, sera traduit en un programme linéaire comme il est indiqué dans la formulation du problème de composition. Cette traduction est assuré par un programme qui a comme entrée le fichier précédent, il lit ligne par ligne le fichier et génère les variables, la fonction objectif, et les contraintes :

Pour chaque module de départ, il va généré une variable *x* qui présente le nombre de copies de ce module et qui doit être utilisées dans la composition. Cette variable est multipliée par le coût du module est ajoutée à la fonction objectif, une contrainte est ajoutée à l'ensemble suivant sa définition dans le fichier de composition.

Un module de départ a un certain nombre de ports d'entrée et de sortie qui sont définis par les clauses importe et exporte : à chaque port le programme doit générer une variable *y* si c'est un port d'entrée et *z* si c'est un port de sortie. Les variable *y* et z vont être multipliées par leurs coûts respectifs et seront ajoutées à la fonction objectif. Une contrainte est générée pour exprimer le nombre de copies à utiliser pour chaque port.

Enfin, car l'utilisateur ne connaît pas la structure interne, le programme doit être capable de créer une autre variable *w* présentant le nombre de connexions utilisées. On associe à cette variable un coût et on l'ajoute à la fonction objectif. On prendra en considération la formulation du problème, on ajoute les deux contraintes en relation avec la variable *w.*

Il reste à déclarer dans le programme que toutes les variables sont entières et la fonction objectif est à minimiser.

En utilisant le problème de composition indiqué dans la figure 3.8, on aura le programme linéaire indiqué dans la figure 3.9. Ce programme linéaire sera résolue et utilisé dans les prochaines étapes. Ce programme linéaire comporte les variables formulées précédemment ainsi que la fonction objective et les contrainte selon la formulation du problème de composition MC.

Toutes les variables sont entières, selon toujours le même formulation.

```
min:6 zJ + 9 zI +  5 x_m1 + 4 yI + 2 zP +  7 x_m2 + 8 yJ + 2 zQ + 9 x_m3 + 8 WI_I + 0
WJ_J;
zJ <=  9.x_m1; zJ >= 4.x_m1;
zI <= 5.x_m1; zI >= 2.x_m1;
x_m1 <= 6; x_m1 >= 2;
yI <= 7.x_m2;yI >= 4.x_m2;
zP <= 5.x_m2; zP >= 2.x_m2;
x_m2 <= 5; x_m2 >= 4;
yJ <=  7.x_m3; yJ >=  5.x_m3;
zQ <=  4.x_m3; zQ >= 3.x_m3;
x_m3 <=  3; x_m3 >= 1;
yI = WI_I;yJ = WJ_J;
zI = WI_I; zJ = WJ_J;
int x_m1, zJ, zI, x_m2, yI, zP, x_m3, yJ, zQ, WI_I, WJ_J;
```

Figure 3.9 : *Programme linéaire*

3.10.5 Relation de compatibilité:

Dans le programme linéaire précédent on peut remarquer facilement deux variables *w,* qui présentent le nombre et le type de connexions. Ces deux variables sont le résultat d'un programme, qui va lire le fichier d'entrée, et constitue des ensembles : un pour les ports d'entrée et l'autre pour les ports de sortie. En définissant la relation de compatibilité, ce programme peut indiquer quel port d'entrée peut être connecté à quel port de sortie. Et pour chaque connexion possible en aura une variable *w* indice le port entrée et port de sortie. Dans notre exemple, deux ports sont compatibles[17] s'ils sont du même type, c'est-à-dire s'ils sont identiques (ils ont les même noms, de mêmes types …). Dans ce cas, la relation de compatibilité représente l'intersection des deux ensembles. Et à chaque élément de cet ensemble on doit générer une variable *w.*

3.10.6 Résolution du programme linéaire :

Un programme qui va prendre en entrée le résultat de l'opération précédente, doit appeler l'outil extérieur pour résoudre le problème linéaire. La solution sera stockée dans un autre fichier qui a l'extension .out. ce fichier ressemble au suivant (figure 3.10):

[17] deux modules sont compatibles s'ils s'échangent de services qui se matchent

Cette solution sert par la suite pour créer les instances de modules les ports, et les connexions.

```
Value of objective function: 669
yI              16
yJ              16
zI              16
zJ              16
zP               8
zQ               9
WI_I            16
WJ_J            16
x_m1             4
x_m2             4
x_m3             3
```

Figure 3.10 : *Solution du programme linéaire*

3.10.7 Module composite :

En se basant sur la solution du problème linéaire et selon l'étape trois de l'algorithme défini ci-dessus, un programme doit lire le fichier issu de la précédente phase, et il effectue les opérations suivantes :

pour chaque variable x_i générer un nombre d'instances égale à la valeur indiquée dans le fichier (figure 3.10). Pour chaque *variable* y_p instancier un nombre de ports importés comme indiqué dans le fichier lu, de la même façon créer les ports de sortie exportés par chaque instance, en fin relie les ports exportés avec les ports importés.

En fin, ce composant doit fournir les interfaces importées et exportées et qui sont définies par un second fichier en entrée et fournies par le composant cité précédemment. Dans l'exemple précédent (figure 3.7), le module composé exporte P et Q ; donc, ce programme fait appel à un autre pour récupérer l'information nécessaire et générer le code adéquat. En se basant sur cette solution en aura une structure similaire à celle présentée dans la figure 3.11.

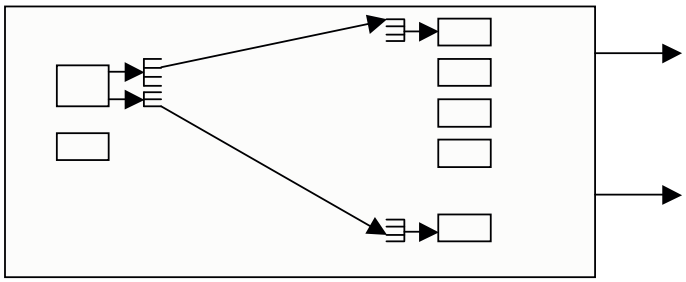


Figure 3.11 : *Module composite*

3.10.8 Génération de code :

Le module composite illustré par le schéma ci-dessus peut être générér comme suit :

- **Instance de module** : une instance est crée par le fragment de code suivant :
```
ServiceItem m2_3 = imports(new ServiceTemplate(null, null,new Entry[]
{new Name("m2")}));
Module mod_m2_3 = (Module) m2_3.service;
```

- **Port d'entrée** : un port est crée par un fragment de code qui est similaire au suivant, le nombre de copies de ce port égale à yI comme dans la solution du problème linéaire.
yI = 16
```
ServiceTemplate[] yI = new ServiceTemplate[16];

yI[0] = new ServiceTemplate(null, new Class[]{I.class}, null);
yI[1] = new ServiceTemplate(null, new Class[]{I.class}, null);
yI[2] = new ServiceTemplate(null, new Class[]{I.class}, null);
yI[3] = new ServiceTemplate(null, new Class[]{I.class}, null);
yI[4] = new ServiceTemplate(null, new Class[]{I.class}, null);
yI[5] = new ServiceTemplate(null, new Class[]{I.class}, null);
yI[6] = new ServiceTemplate(null, new Class[]{I.class}, null);
```

- **Port de sortie** : prendra la forme du code suivant, le nombre de copie de ports d'entrée est à lire dans la solution (ici yi = 3) :
```
ServiceItem[] zI = new ServiceItem[16];

zI[0] = imports(new ServiceTemplate(null, new Class[]{I.class}, null));
zI[1] = imports(new ServiceTemplate(null, new Class[]{I.class}, null));
zI[2] = imports(new ServiceTemplate(null, new Class[]{I.class}, null));
zI[3] = imports(new ServiceTemplate(null, new Class[]{I.class}, null));
zI[4] = imports(new ServiceTemplate(null, new Class[]{I.class}, null));
zI[5] = imports(new ServiceTemplate(null, new Class[]{I.class}, null));
```

- Un module qui importe un certain service, et exporte un autre utilise ce fragment de code:
```
mod_m2_3.items[mm] = imports(yI[mm]);
mod_m2_3.exports(zP[ll]);
```

- **Connexion** :Une connexion aura la forme suivante :

```
//Le Connections:
Random ranI = new Random();
```

```
      zI[ranI.nextInt(zI.length - 1)] = imports(yI[ranI.nextInt(yI.length -
1)]);
      zI[ranI.nextInt(zI.length - 1)] = imports(yI[ranI.nextInt(yI.length -
1)]);
```

- **Le module composite**: indiqué plus haut exporte p et q comme suit :

```
      ServiceItem item_1 = imports(new ServiceTemplate(null, new
Class[]{P.class}, null));
      this.exports(item_1);
      ServiceItem item_2 = imports(new ServiceTemplate(null, new
Class[]{Q.class}, null));
      this.exports(item_2);
```

Ces lignes constituent la structure de code qui est dynamique, c'est-à-dire qui dépend de la solution du problème MC, l'autre code dit figé, est le code qui est commun, est généré indépendamment de la solution du problème linéaire. En fin, le code généré doit être compilable, autrement dit ne contenir pas d'erreurs. La totalité de code est une classe qui hérite de la classe « module » qui implémente Serializable. Reste à noter qu'un ''SecurityManager'' est nécessaire dans des programmes similaires ou il y a d'importation et d'exportation de code.

Chapitre 4

Un exemple d'utilisation

4.1 Introduction[18] :

Dans ce chapitre, nous voulons montrer par un exemple concret comment on peut utiliser notre méthodologie pour construire un module composite à partir des modules existants. Les modules de départ dans cet exemple présentent deux entités matérielles : un appareil numérique et une imprimante, chacune des entités est représentée par un module. L'appareil numérique représenté par un module qui exporte des images et l'imprimante est représentée par un module qui importe des fichiers en entrée est les imprime en sortie.

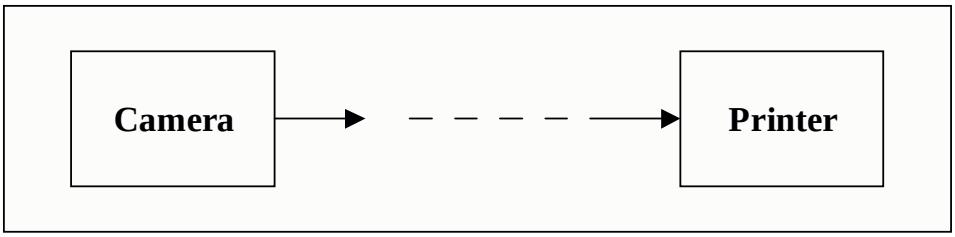


Figure 4.1 : *Module composite*

```
COMPOSITION: printer
INPUT:

MODULE : Camera;
EXPORTE: Camera; [2,5] $9
CONSTRAINTS: [2,6]
COST: 5;

MODULE : Printer;
IMPORTE: Camera;  [4,7] $4
EXPORTE: Printer; [2,5] $2
CONSTRAINTS: [4,5]
COST: 7;

OUTPUT:
MODULE: Print_image;
EXPORTE: Printer;
END
```

```
min:9 zCamera +  5 x_Camera + 4
yCamera + 2 zPrinter +  7 x_Printer + 7
WCamera_Camera;

zCamera <= 5.x_Camera;
zCamera >= 2.x_Camera;
x_Camera <= 6;
x_Camera >= 2;
yCamera <= 7.x_Printer;
yCamera >= 4.x_Printer;
zPrinter <= 5.x_Printer;
zPrinter >= 2.x_Printer;
x_Printer <= 5;
x_Printer >= 4;
yCamera = WCamera_Camera;
zCamera = WCamera_Camera;
int x_Camera, zCamera, x_Printer,
yCamera, zPrinter, WCamera_Camera;
```

```
Value of objective
function: 384

yCamera              16
WCamera_Camera       16
zPrinter              8
x_Camera              4
zCamera              16
x_Printer             4
```

Figure 4.2 A: *Problème de compositio* ***B:*** *Programme linéaire* ***C:****Solution du P.linéaire*

[18] Cet exemple est inspiré de [15, 16].

4.2 appareil numérique :

Le module qui représente l'appareil numérique est une classe qui hérite de la classe 'Module'. Cette classe implémente l'interface publique Camera, cette interface définit le service à exporter par le module; elle hérite l'interface Remote ; elle comporte une seule méthode déclarée publique qui permet de récupérer l'image. Car cette méthode est appelée d'une manière éloignée une RemoteException est nécessaire, voici le code de son code :

```
import java.io.File;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface Camera extends Remote{

    public File GetImage() throws RemoteException;

}
```

Cette interface sera implémentée par une classe Camera_Impl qui implémente aussi Serializable, elle hérite deux opérations de la classe Module : importe et exporte. Ces deux opération permettent au module d'échanger des services avec le monde extérieur, il envoie des services par l'opération d'exportation, et récupère des services par l'opération d'importation. Voici le code complet de cette ce module :

```
import jini.module.*;
import common.entry.Cost;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.entry.Entry;
import java.io.Serializable;
import net.jini.lookup.entry.Name;
import java.io.File;
public class Camera_Impl extends Module implements Camera, Serializable{
public File GetImage(){
    File f = new File("cavalo.jpg");
    return f;
}
public Camera_Impl(){
    exports(new ServiceItem(null, this, new Entry[]{new Name("Camera")}));
}
public static void main(String args[]){
    new Camera_Impl();
    try{
            Thread.currentThread().sleep(10 * 1000L);
    } catch(java.lang.InterruptedException exc){
    }
}
}
```

4.3 Imprimante :

Comme avec l'appareil numérique, le module qui représente l'imprimante est très similaire. Une interface publique qui définie le service à exporter par le module est donnée ci-dessous; elle comporte essentiellement une méthode publique qui permet d'imprimer un document ou une image. Comme indiqué, cette méthode reçoit comme argument un fichier et elle n'a pas d'argument de retour.

```
import java.io.File;
import java.rmi.RemoteException;
public interface Printer{
        public void print(File f) throws RemoteException;
}
```

L'implémentation de cette interface est réalisée par la classe suivante qui hérite aussi de la classe module, dont elle peut utiliser nécessairement les opérations d'importation. A l'inverse du module Camera, ce module effectue une opération d'importation puis une opération d'exportation.

```
import jini.module.*;
import common.entry.Cost;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.entry.Entry;
import java.io.Serializable;
import net.jini.lookup.entry.Name;
import java.io.*;
import javax.print.*;
public class Printer_Impl extends Module implements Printer, Serializable{
public void print(File f){
     System.out.println("Impression d'image :");
     //impléméntation à continuer
}
public Printer_Impl(){
ServiceItem item = imports(new ServiceTemplate(null, new Class[]{Camera.class}, null));
Camera camera = (Camera) item.service;
File f  =  camera.GetImge();
Print(f);
exports(new ServiceItem(null, this, new Entry[]{new Name("Printer")}));
}
public static void main(String args[]){
new Printer_Impl();
try{
     Thread.currentThread().sleep(10 * 1000L);
} catch(java.lang.InterruptedException exc){}
}
}
```

4.4 Module composite:

Le code du module composite, qui est la composition du module Camera et Printer , ressemble aux lignes suivantes :

```
import java.io.*;
import jini.module.Module;
import service.common.Camera;
import service.common.Printer;
import service.common.Camera;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;
import java.util.Random;
public class Module_Result extends Module implements Serializable {
public Module_Result(){
      System.setSecurityManager(new java.rmi.RMISecurityManager());
      //Les Ports d'entrée
      ServiceTemplate[] yCamera = new ServiceTemplate[16];
      yCamera[0] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[1] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[2] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[3] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[4] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[5] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[6] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[7] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[8] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[9] = new ServiceTemplate(null, new Class[]{Camera.class}, null);
      yCamera[10] = new ServiceTemplate(null, new Class[]{Camera.class},
null);
      yCamera[11] = new ServiceTemplate(null, new Class[]{Camera.class},
null);
      yCamera[12] = new ServiceTemplate(null, new Class[]{Camera.class},
null);
      yCamera[13] = new ServiceTemplate(null, new Class[]{Camera.class},
null);
      yCamera[14] = new ServiceTemplate(null, new Class[]{Camera.class},
null);
      yCamera[15] = new ServiceTemplate(null, new Class[]{Camera.class},
null);
      //Les Ports de srotie
      ServiceItem[] zPrinter = new ServiceItem[8];
      zPrinter[0] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[1] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[2] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[3] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[4] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[5] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[6] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      zPrinter[7] = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
      //Les Ports de srotie
      ServiceItem[] zCamera = new ServiceItem[16];
```

```
      zCamera[0] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[1] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[2] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[3] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[4] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[5] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[6] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[7] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[8] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[9] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[10] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[11] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[12] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[13] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[14] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      zCamera[15] = imports(new ServiceTemplate(null, new
Class[]{Camera.class}, null));
      //Les instances de Modules
      ServiceItem Camera_1 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Camera")}));
      Module mod_Camera_1 = (Module) Camera_1.service;
      ServiceItem Camera_2 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Camera")}));
      Module mod_Camera_2 = (Module) Camera_2.service;
      ServiceItem Camera_3 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Camera")}));
      Module mod_Camera_3 = (Module) Camera_3.service;
      ServiceItem Camera_4 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Camera")}));
      Module mod_Camera_4 = (Module) Camera_4.service;
      //Les instances de Modules
      ServiceItem Printer_1 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Printer")}));
      Module mod_Printer_1 = (Module) Printer_1.service;
      ServiceItem Printer_2 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Printer")}));
      Module mod_Printer_2 = (Module) Printer_2.service;
      ServiceItem Printer_3 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Printer")}));
      Module mod_Printer_3 = (Module) Printer_3.service;
      ServiceItem Printer_4 = imports(new ServiceTemplate(null, null,new
Entry[] {new Name("Printer")}));
      Module mod_Printer_4 = (Module) Printer_4.service;
      //Le Connections:
      Random ranCamera = new Random();
      zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
      zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
```

```
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		zCamera[ranCamera.nextInt(zCamera.length - 1)] =
imports(yCamera[ranCamera.nextInt(yCamera.length - 1)]);
		//Exportations & Importations
		ServiceItem item_1 = imports(new ServiceTemplate(null, new
Class[]{Printer.class}, null));
		this.exports(item_1);
	}

	public static void main(String[] args){
		new Module_Result();
		try{
			Thread.currentThread().sleep(10000L);
		} catch(java.lang.InterruptedException exc){
		}
	}
}
```

Chapitre 5

CONCLUSION & PERSPECTIVES

5.1 Conclusion :

Le travail réalisé dans ce mémoire suit les étapes logiques qu'on a citées dans les chapitres précédents : la première étape est la définition d'un problème de composition qui est divisé en deux parties : une partie consacrée aux modules disponibles ou de départ, dans l'autre partie on défini le module cible, le problème est à définir dans un fichier dont la structure est définie par une grammaire. On essaye à partir de ce problème d'en déduire un programme linéaire comme il est indiqué dans la formalisation du problème. Ce problème linéaire est à stocker dans un fichier, ce problème est à résoudre par un outil extérieur qui va nous retourner la solution optimale du programme linéaire. Un autre programme va prendre cette solution pour créer le module composite qui est demandé en entrée, ce module composite est défini par son code complet. Toutes ces étapes constituent une chaîne complète, elles permettent d'arriver à une composition optimale et répond aux besoins du client.

D'après ce qu'on a dit, on peut en tirer comme conclusion les points suivants : la composition est automatique, du fait que le code du module composite est généré automatiquement sans intervention de l'utilisateur. La solution est optimale, comme on a vu que le problème est transformé à un programme linéaire. La compatibilité est assurée par un programme dans lequel on peut ajouter d'autres propriétés fonctionnelles ou non fonctionnelles, ou le modifier sans affecter l'ensemble. La définition des ports est claire, en effet un port est un service que ce soit importé ou exporté. Les opérations d'importation et d'exportation sont définies explicitement, elles permettent aux modules d'importer ou d'exporter des services sans aucun souci de son implémentation.

5.2 Perspectives :

Nous souhaiterons au futur et en se basant sur les résultats du travail effectué, compléter notre effort par d'autres moyens plus efficaces et plus économiques, dont on cite :

- Notre travail est basé sur un fichier d'entrée textuel, nous souhaitons au futur avoir un moyen graphique pour définir un problème de composition pour une meilleure lisibilité.

- Les modules utilisent les opérations importe et exporte pour échanger de services Jini avec le monde extérieur. Nous souhaiterons étendre ce moyen à d'autres technologies, comme EJB et Web service, et ce pour une meilleure exploitation des services disponibles.

- En visant un meilleur pouvoir d'expression, nous souhaitons avoir un langage de définition de services et de composition de modules.

Annexe A

Cet annexe est une présentation détaillée du code présenté dans le chapitre deux, il comporte quatre package : common, ui, service et client.

- Le package *common* contient l'interface Calculator :

```
ackage common;
import java.awt.Frame;
import java.rmi.*;
public interface Calculator{
        public double add(double x,double y);
        public int add(int x,int y);
        public double subtract(double x,double y);
        public int subtract(int x,int y);
        public double multiply(double x,double y);
        public int multiply(int x,int y);
        public double divide(double x,double y);
}
```

- Le package *ui* contient l'implémentation de l'interface graphique :

```
import common.Calculator;
import net.jini.lookup.ui.MainUI;
import net.jini.core.lookup.ServiceItem;
import java.awt.event.*;
import java.awt.*;


public class CalcFrame extends Frame implements
Calculator, ActionListener, KeyListener, MainUI  {
//les bouttons de l'intaerface
public CalcFrame(ServiceItem item, String name){
                super(name);
                this.item = item;
                initializeUI();
                initializeEventListeners();
                setVisible(false);
}
public CalcFrame(){}
private void initializeUI(){ //inisialisation des buttons de l'interface}
private void initializeEventListeners(){//ajoute de l'écouteur d'évènements}
//implémentation de l'interface Calculator
public static void main(String args[]){
                ServiceItem item =null;
                CalcFrame cFrame = new CalcFrame(item, "Calculator");
                cFrame.setBounds(20,20,300,300);
                cFrame.setTitle("Calculator ");
                cFrame.setVisible(true);
}
}
```

- Package *service* :

```
package service;
import ui.CalcFrame;
import ui.frameFactory;
import java.rmi.RMISecurityManager;
import net.jini.lookup.JoinManager;
```

```
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RemoteException;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import net.jini.core.entry.Entry;
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;
import java.rmi.MarshalledObject;
import java.io.IOException;
import java.util.Set;
import java.util.HashSet;

public class CalculatorServer implements ServiceIDListener {
 public static void main(String argv[]) {
        new CalculatorServer();
        Object keepAlive = new Object();
        synchronized(keepAlive) {
         try {
                keepAlive.wait();
        } catch(InterruptedException e) {}
}
public CalculatorServer() {
        System.setSecurityManager(new RMISecurityManager());
        JoinManager joinMgr = null;
        Set typeNames = new HashSet();
        typeNames.add(FrameFactory.TYPE_NAME);
        Set attribs = new HashSet();
        attribs.add(new UIFactoryTypes(typeNames));
        MarshalledObject factory = null;
        try {
           factory = new MarshalledObject(new frameFactory());
        } catch(Exception e) {
          e.printStackTrace();
         System.exit(2);
        }
        UIDescriptor desc = new UIDescriptor(MainUI.ROLE, frameFactory.TOOLKIT, attribs, factory);
        Entry[] entries = {desc};
        LookupDiscoveryManager mgr = null;
        leaseRenewalManager leaseMgr = new LeaseRenewalManager();
        try {
        mgr = new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,  null , null);
        joinMgr = new JoinManager(new CalcFrame(), entries, this, mgr, leaseMgr);
        } catch(Exception e) {}
         }
         public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("Reception du service ID :" + serviceID.toString());
 }
}}
```

- Package *client*

```
package client;
import common.Calculator;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
```

```
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.core.entry.Entry;
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;
import java.awt.*;
import javax.swing.*;
import java.util.Set;
import java.util.Iterator;
import java.net.URL;
public class CalculatorClient {
private static final long WAITFOR = 15*60*1000L;
 public static void main(String argv[]) {
        new CalculatorClient();
        try {
        Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {}
}
public CalculatorClient() {
        ServiceDiscoveryManager sdmgr = null;
        System.setSecurityManager(new RMISecurityManager());
        LookupDiscoveryManager ldmgr = null;
        try {
        ldmgr = new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS, null, null);
        sdmgr = new ServiceDiscoveryManager(ldmgr, new LeaseRenewalManager());
        } catch(Exception e) {}
        Class [] classes = new Class[] {common.Calculator.class};
        Entry [] entries = null;
        ServiceTemplate template = new ServiceTemplate(null, classes, entries);
        ServiceItem item = null;
        try {
           item = sdmgr.lookup(template, null, WAITFOR);
        } catch(Exception e) {}
        if (item == null) {
           System.out.println("aucun service ...");
           System.exit(1);
        }
        if (item.service == null) {
           System.out.println("service null");
           System.exit(1);
        }
        checkUI(item);
   }
   private void checkUI(ServiceItem item) {
        Entry[] attributes = item.attributeSets;
        for (int m = 0; m < attributes.length; m++) {
           Entry attr = attributes[m];
           if (attr instanceof UIDescriptor) {
                checkForAWTFrame(item, (UIDescriptor) attr);
           }
        }
   }
   private void checkForAWTFrame(ServiceItem item, UIDescriptor desc) {
        Set attributes = desc.attributes;
        Iterator iter = attributes.iterator();
        while (iter.hasNext()) {
           Object obj = iter.next();
           if (obj instanceof UIFactoryTypes) {
                UIFactoryTypes types = (UIFactoryTypes) obj;
        if (types.isAssignableTo(FrameFactory.class)) {
```

```
                    FrameFactory factory = null;
                    try {
                            factory = (FrameFactory) desc.getUIFactory
                            (this.getClass().getClassLoader());
                       } catch(Exception e) {}
                       System.out.println("Téléchargement de la Frame: " + item);
                  Frame frame = factory.getFrame(item);
                  frame.setLocation(500, 400);
                  frame.setSize(300, 300);
                  frame.setTitle("Calculator : Veriamg");
                  frame.setResizable(false);
                    frame.setVisible(true);
                    }
              }
          }
    }
}
```

Annexe B

Dans cet annexe on donne le code complet de la classe Module, discutée au cours du chapitre 3. Elle implémente essentiellement les interfaces importation et exportation, ces deux interfaces permettent de définir deux opérations importe et exporte :

```
package jini.module;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceID;
import net.jini.lookup.JoinManager;
import net.jini.lookup.ServiceIDListener;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.lookup.ServiceDiscoveryEvent;
import net.jini.lookup.ServiceItemFilter;
import net.jini.lookup.LookupCache;
import net.jini.lease.LeaseRenewalManager;
public class Module implements Importation, Exportation,  ServiceIDListener, ServiceDiscoveryListener,
ServiceItemFilter{
public static ServiceItem[] items;
public Module(){
}
public void exports(ServiceItem item){
System.setSecurityManager(new java.rmi.RMISecurityManager());
LookupDiscoveryManager ldmgr = null;
JoinManager jmgr = null;
LeaseRenewalManager leaseMgr = LeaseRenewalManager();
try{
ldmgr = new LookupDiscoveryManager( LookupDiscovery.ALL_GROUPS, null, null);
jmgr = new JoinManager( item.service, item.attributeSets, this, ldmgr, leaseMgr);
}catch(Exception exc){}
}
public ServiceItem imports(ServiceTemplate template){
System.setSecurityManager(new java.rmi.RMISecurityManager());
ServiceDiscoveryManager sdmgr = null;
LookupDiscoveryManager ldmgr = null;
LookupCache cache = null;
try{
ldmgr = new LookupDiscoveryManager( LookupDiscovery.ALL_GROUPS, null, null);
sdmgr = new ServiceDiscoveryManager(ldmgr,
new LeaseRenewalManager());
cache = sdmgr.createLookupCache(template, this, this);
 } catch(Exception exc){}
items = cache.lookup(this, 10);
return cache.lookup(this);
}
public void serviceIDNotify(ServiceID id){
System.out.println("Un service exporté :" + id.toString());}
public void serviceAdded(ServiceDiscoveryEvent evt){
System.out.println("Un nouveau service :" + evt.getPostEventServiceItem().service.toString());    }
public void serviceRemoved(ServiceDiscoveryEvent evt){
System.out.println("Un service suprimé :" + evt.getPreEventServiceItem().service.toString());}
public void serviceChanged(ServiceDiscoveryEvent evt){
System.out.println("Un service modifié :" + evt.getPostEventServiceItem().service.toString());}
public boolean check(ServiceItem item){return true;}
}
```

BIBLIOGRAPHIE

[1] Scott Oaks & Henry Wong *Jini in a Nutshell , 1st edition march 2000*
[2] W.Keith Edwards *Core Jini , second edition : 2000*
[3] Jan Newmarch *A programmer's guide to Jini*
[4] Dipanjan Chakraborty *Service Composition in Ad-hoc Environments* March 9, 2004
[5] Ilango Kumaran;*Jini Technology: An Overview, Nov 2001, Barnes & noble*
[6] W.Keith Edwards & Tom Rodden *Jini example by example, Juin 2001 Barnes & noble*
[7] Ken Arnold *The Jini Specifications: Second edtion, Dec 2000 Addison-Wesley*
[8] Stavros Tripakis *Automated Module Composition at Verimag*
http://www-verimag.imag.fr/~tripakis/papers/tacas03-web.pdf
[9] Sing Li With Ronald Ashri & Jerome Scheuring *Professional Jini, aout 2000*
[10] Fredrik Andersson and Magnus Karlsson *Secure Jini Services in Ad Hoc Networks:, 2000*
[11] Dirk Gorissen *H2O Metacomputing - Jini Lookup and Discovery*
University of Antwerp Faculty of Sciences 2003-2004
[12] Jason I. Hong *An Overview of the Jini Coordination Framework*
University of California, Berkeley
[13] Michael Fahrmair, Chris Salzmann, Maurice Schoenmakers *Managing Dynamic Distributed Jini Systems* http://www.research.ibm.com/Middleware2000/WiP_Papers/carpat.pdf
[14] Grigore Rosu *Abstract Semantics for Module Composition University of California*
http://gureni.cs.uiuc.edu/~grosu/download/mod.pdf
[14] Bhaskaran Raman, Sharad Agarwal, Yan Chen .. *The SAHARA Model for Service Composition Across Multiple Providers*
http://www.cs.berkeley.edu/~jshih/Publications/Pervasive02.pdf
[15] Nikola Milanovic, Vladimir Stantchev, Jan Richling, Miroslaw Malek
Towards adaptive & Composable Services
http://www.informatik.hu-berlin.de/~milanovi/services.pdf
[16] Nikola Milanovic, Vladimir Stantchev and Miroslaw Malek
Definition of contracts for Service Composition
http://www.informatik.hu-berlin.de/~milanovi/cdl.pdf
[17] Matthias Werner, Jan Richlin, Nikola Milanovic Vladimir Stantchevz
Composability Concepts for Dependable Embedded Systems
http://www.informatik.hu-berlin.de/~milanovi/composability_concept.pdf
[18] Patrick Thomas Eugster *Dynamic proxies for classes*
http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200317.pdf
[19] Emre K c man and Armando Fox *Separation of Concerns in Networked Service Composition 15th March 2001* www.stanford.edu/~emrek/pubs/wascse01.ps
[20] Bhaskaran Raman, Randy H. Katz
An architecture for Highly Available Wide-Area Service Composition
http://www.cse.iitk.ac.in/~bhaskar/berkeley/ccj/ccj.pdf
[21] Toyotaro Suzumura, Satoshi Matsuoka, Hidemoto Nakada *A Jini-based Computing Portal System* http://www.sc2001.org/papers/pap.pap200.pdf
[22] Ken Arnold *The Jini™ Architecture: Dynamic Services in a Flexible Network*
http://www.sigda.org/Archives/ProceedingArchives/Dac/Dac99/papers/1999/dac99/pdffiles/09_3.pdf
[23] Kevin Bowers Kevin Mills and Scott Rose *Self-Adaptive Leasing for Jini*
http://www-2.cs.cmu.edu/~kbowers/bowers03adaptive.pdf
[24] Jan Newmarch and Robin Kirk *A Service Architecture for Scalable Distributed Audio*
http://jan.netcomp.monash.edu.au/publications/scalable_audio.pdf
[25] Jim Waldo *The Jini™ Architecture*
http://www.cse.ttu.edu.tw/~cheng/courses/Java/e717.pdf
[26] ANDREAS SPECK, ELKE PULVERMÜLLER *Component Composition Validation*

http://matwbn.icm.edu.pl/ksiazki/amc/amc12/amc1253.pdf
[27] Gregor G¨ossler and Joseph Sifakis *Composition for Component-Based Modeling* http://www-verimag.imag.fr/~sifakis/fmco02.pd
[28] Larry Rudolph *Review of the Ninja Architecture* http://www.org.lcs.mit.edu/6.964/ninja.pdf
[29] Marco Zennaro, Jeff Ko, Raja Sengupta and Stavros Tripakis *A service network architecture for a multivehicle search mission* http://www-verimag.imag.fr/~tripakis/papers/svnets-cdc01.pdf
[30] Vladimir Tosic, David Mennie, Bernard Pagurek *Dynamic Service Composition and Its Applicability toE-Business Software Systems – The ICARIS Experience* http://www.sce.carleton.ca/netmanage/papers/TosicEtAlWOOBS01SV.pdf
[31] Chakraborty, D. Chen, H. *Service Discovery in the Future for Mobile Commerce. ACM Crossroads*, Vol. 7, Issue 2 (Winter 2000) 18-24. On-line at: http://www.acm.org/crossroads/xrds7-2/service.html
[32] Rekesh, John. *UPnP, Jini and Salutation : A look at some popular coordination framework for future network devices* Technical Report, California Software Lab, 1999. http://www.cswl.com/whiteppr/tech/upnp.html.
[33] Mennie David, Pagurek Bernard.: *An Architecture to Support Dynamic Composition of Service Components*. Presented at WCOP 2000 (Sophia Antipolis, France, June 2000). On-line at: http://www.doc.ic.ac.uk/~hf1/phd/papers/toread/mennie00architecture.pdf
[34] John McKim Dynamic Composition Using Jini and JavaSpaces October 19, 1999 http://www.jini.org/meetings/second/jcm2_dynamic_softw_mitre.pdf
[35] Frank Sommers Survival of the fittest Jini services, Part 1 09/21/2001 http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-jiniology.html

WEBOGRAPHIE

1) http://www.sun.com/jini
2) http://www.jini.org/
3) http://java.sun.com/products/jdk/rmi/
4) http://java.sun.com/products/jdk/rmi
5) http://www.cswl.com/whiteppr/tutorials/jini.html
6) http://www.artima.com/jini/index.html
7) http://www.kedwards.com/jini/
8) http://www.cdegroot.com/cgi-bin/jini
9) http://www.javaworld.com/javaworld/topicalindex/jw-ti-jiniology.html
10) http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml
11) http://www.cs.auc.dk/~illum/jini/
12) http://members.tripod.com/gsraj/jini/chapter/
13) http://www.cs.wustl.edu/mobilab/Publications/Papers/2004/
14) http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-jiniology.html