UNIVERSITE LIBRE DE BRUXELLES
Faculté des Sciences appliquées

Année académique 2000-2001

Adaptation du protocole Jini à la parallélisation d'algorithmes

DIRECTEUR DE MEMOIRE : Prof. H.Bersini

TRAVAIL DE FIN D'ETUDES
PRESENTE PAR OLIVIER SAMYN
EN VUE DE L'OBTENTION DU GRADE
D'INGENIEUR CIVIL INFORMATICIEN

Remerciements

Je voudrais remercier toutes les personnes qui de près ou de loin m'ont aidé dans l'accomplissement de mon travail de fin d'études.

Tout d'abord, mon promoteur, Mr Hugues Bersini et ses asistants pour le suivi du travail et les divers conseils quant aux orientations à prendre.

Ensuite, tous ceux qui ont pris le temps de relire et d'apporter leurs commentaires à mon travail : Natacha Vanpée, Joël Cannau, et tous les autres.

Merci aussi à Jean-Michel Dricot pour nos grandes discussions autour de JINI, RMI, CORBA, ...

Merci à tous mes camarades de promotion pour l'ambiance de travail surtout durant les longues journées passées à rédiger ce travail dans la salle Palto.

Merci à ma petite soeur Cindy pour ses conseils de présentation et ses trucs et astuce de TFE-iste.

Merci à mes parents pour leurs allez-retour Bruxelles-Arlon durant la rédaction de ce travail, pour leur soutien moral, et pour m'avoir encouragé durant tout mon cursus.

Merci enfin à Julie qui non seulement a eu la patience de relire tout le travail, mais qui m'a soutenue durant toute cette année et qui a supporté mes incompréhensibles explications sur JINI et sur le calcul parallèle.

Chapitre 1

Introduction

Le marché actuel de l'informatique est en constante évolution. L'utilisation de réseaux d'ordinateurs est de plus en plus courante. Par contre, le nombre de technique permettant de faciliter la programmation et la gestion de services distribués n'est pas important.

JINI est l'une de ces techniques. JINI est plus qu'un protocole. C'est une série de protocole définissant une architecture construite autour d'un service appelé "lookup".

Mes motivations pour aborder JINI furent tout d'abord la plate-forme dont il dépend : Java. Plus qu'un langage, Java définit avec sa Java Virtual Machine, tout un environnement, indépendant de l'architecture d'ordinateur utilisé. Dans l'informatique actuelle, on ne peut pas être sûr du type d'ordinateur que l'on aura à utiliser dans l'avenir. Mais une chose est sûre : quelque soit l'architecture utilisée, il sera possible d'y faire tourner une Java Virtual Machine et de bénéficier de toutes les applications écrites pour cette plate-forme.

Ensuite, lorsqu'est venu l'idée d'étudier JINI, il manquait une application concrète à tester. Cette application, je l'ai trouvée au sein même de notre université, elle est utilisée par de nombreux chercheurs : le calcul scientifique.

Qui dit calcul scientifique dans le cadre d'une application faisant appel à un réseau, dit aussi calcul parallèle. L'application à tester avec JINI était donc trouvée : utiliser JINI comme gestionnaire pour un réseau d'ordinateurs devant effectuer un calcul.

Le calcul tel qu'envisagé ici peut trouver des applications dans bien des domaines :

- en recherche, avec des calculs demandant des plus en plus de puissance ;
- en génération d'images de synthèse de plus en plus utilisées par les cinéastes ;
- en entreprises, avec des besoins de forte puissance dans l'extraction d'informations à partir d'une grande quantité de données.

Une seconde motivation pour l'approche du calcul parallèle avec JINI vient d'une observation assez simple. Dans de nombreux endroits, il est possible de trouver des ordinateurs fonctionnant jour et nuit et qui ne sont utilisés qu'une petite partie de la journée. On peut trouver ce type de situation dans de nombreux endroits :

- Des écoles avec des salles informatiques destinées aux étudiants ;

- Des entreprises où chaque bureau est équipé d'un ordinateur qui ne sert à rien lorsque son utilisateur est en réunion ou en déplacement ;
- Des ordinateurs personnels, qui parcequ'ils sont connectés à internet par cable restent allumés toute la journée sans servir ;
- Des laboratoires où de puissantes machines de calcul restent parfois inutilisées ;
- ...

Il existe dans le monde une puissance de calcul latente prête à être utilisée.

C'est dans cette perspective que JINI va apporter de nouvelles idées au calcul parallèle et permettre d'utiliser ces ressources tout en gardant une extrême simplicité de gestion du réseau d'ordinateurs.

La suite de ce document se découpera en quatre parties :
- une introduction à JINI : son but et son fonctionnement ;
- une recherche d'architecture de calcul parallèle à implémenter en JINI ;
- l'implémentation de cette architecture ainsi que quelques résultats de tests ;
- une analyse des apports que peut avoir JINI dans le monde du calcul parallèle.

On pourra trouver en annexe : un exemple de programme illustrant l'utilisation de JINI, ainsi que la documentation de la plate-forme de test générée à partir du code Java.

Chapitre 2

Architecture Jini

2.1 Introduction

JINI est une architecture créée par SUN, les premières versions publiques datent de 1999, et toujours en cours de développement. Pour ce projet, la version “Jini 1.1” (octobre 2000) a été utilisée.

Le but principal de JINI est d’offrir à l’utilisateur un réseau flexible et facilement configurable. Les ressources doivent pouvoir être découvertes par un utilisateur ou une machine. Ces ressources sont soit des logiciels soit du matériel.

Dans JINI,l’accent est mis sur la dynamique. Une ressource peut apparaître, disparaître ou devenir inaccessible, le tout devant rester transparent pour l’utilisateur. JINI est donc un ensemble constitué de :

- Composants permettant la mise en place du partage de ressources entre les utilisateurs ;
- Une API permettant de créer facilement des nouveaux services JINI.

Le texte qui suit est fortement inspiré de la description de l’architecture JINI qui est faite dans le document [13].

2.2 Concepts de base

Afin de rendre l’explication de JINI plus claire, il est utile d’introduire quelques concepts de base :

- Les services ;
- Le “Lookup” ;
- Le “Leasing” ou “bail” ;
- Les événements.

2.2.1 Services

Les services sont des entités qui peuvent être utilisées par des personnes, des programmes ou d'autres services. Un service peut être un ordinateur, un système de stockage, une imprimante réseau,. . . Un service peut être matériel ou logiciel.

La méthode de communication entre service et utilisateur n'est pas imposée par JINI. Par contre, un service offre à l'utilisateur une Interface au sens du langage Java. Cette interface est un ensemble de prototypes de méthodes, l'implémentation étant laissée au programmeur du service. Ainsi, une même interface peut-être implémentée de plusieurs manières différentes selon les caractéristiques propres du service.

Prenons un exemple simple (exemple classique dans le monde JINI inspiré par celui donné dans [13]) : Soit deux imprimantes, l'une couleur, l'autre noir et blanc. L'imprimante couleur est directement connectée à un réseau et est capable d'imprimer directement un document Postscript. L'imprimante noir et blanc est une imprimante PCL connectée à un ordinateur. Cet ordinateur est connecté au réseau et se charge de faire la conversion PostScript→PCL.

Ces deux imprimantes partagent une même interface (représentée ici à la manière Java) :

```
public interface PrinterInterface {
    public boolean printDocument(String postscriptDoc );
    public void reset ();
}
```

Ce qui donne deux fonctions possibles :
- L'impression d'un document postscript transmis sous forme d'une chaîne de caractères ;
- Une remise à zéro.

Ces fonctions sont partagées par de nombreuses imprimantes, mais l'implémentation différera selon le type d'imprimante. Nous aurons dans ce cas-ci deux implémentations différentes pour un même service d'impression, l'une logicielle et l'autre matérielle :

- Une implémentation matérielle pour l'imprimante couleur où un appel à la fonction `printDocument` transmet directement le document postscript au système d'impression de l'imprimante ;
- Une implémentation logicielle pour l'imprimante noir et blanc où un appel à la fonction `printDocument` transmet la chaîne de caractères à un logiciel de transformation, Postscript vers PCL, qui transmet le résultat à l'imprimante.

2.2.2 Lookup

Le “Lookup” est le service de base de JINI. Ce service fonctionne comme une grosse table associant les interfaces offertes avec les objets offrant ces interfaces. Une notion d'attributs peut être ajoutée à chaque association. Le mécanisme permettant la création et l'accès à cette table sera expliqué plus longuement avec le fonctionnement du lookup, Section 2.3.

Outre cette table et sa propagation, le lookup gère l'apparition et la disparition de services, ainsi que la propagation de ces événements aux différents clients. Les différents Lookup pouvant communiquer entre eux, il est aisé de créer un réseau de lookup et donc de propager les tables de correspondances.

Malheureusement, dans la version de lookup fournie avec JINI, il manque certaines fonctions de gestion. La découverte d'autres lookup ne peut se faire que par multicast IP, ce qui réduit les chances de découverte sur un réseau tel celui de l'ULB qui limite les paquets multicast à un sous-réseau.

De même, par défaut, toutes les communications entre les services et le lookup, ainsi qu'entre les clients et le lookup se font via le protocole RMI inclus avec le langage Java. Selon les développeurs de JINI, il serait possible d'implémenter une version du lookup n'utilisant pas RMI. D'un autre côté, puisque les objets envoyés sont des objets Java, utiliser un autre protocole devient inutile.

Enfin, il faut savoir que Java n'est pas indispensable pour faire fonctionner des services JINI. Il existe des solutions déjà en place qui permettent une interface entre un langage comme le C/C++ et JINI. Ces solutions permettent l'utilisation de JINI sur des systèmes embarqués ne disposant pas de Java Virtual Machine.

2.2.3 Leasing (ou Bail)

Pour pouvoir remplir les objectifs de JINI, il faut être capable de détecter la disparition d'un service. Le mécanisme mis en oeuvre pour cette détection est le "Leasing".

Le principe est assez simple: un système de bail est mis en place entre le service et le lookup. Ce bail est caractérisé par deux durées:

- **une durée de vie.** Cette durée peut être limitée ou non dans le temps. Elle définit la durée durant laquelle le lookup gardera l'entrée du service dans sa table.
- **une durée de renouvellement.** Le service doit périodiquement renouveler son bail durant la durée de vie de celui-ci.

Le principe fonctionnement du bail est le suivant: un service reste enregistré durant toute la durée de vie du bail sauf si ce bail n'a pas été reconduit au terme de la durée de renouvellement.

2.2.4 Evénements

JINI définit une série d'événements associés au fonctionnement du lookup. Ces événements sont assez simples: apparition, disparition d'un service et modification des attributs d'un service.

Un client peut demander au lookup de lui envoyer tout ou une partie de ces événements. Cela lui permet de réagir à la situation du réseau.

2.2.5 Quelques conventions

Avant de poursuivre plus loin, voici quelques conventions de diagrammes qui seront utilisées dans le reste de ce document. (voir Figure 2.1, page 7)

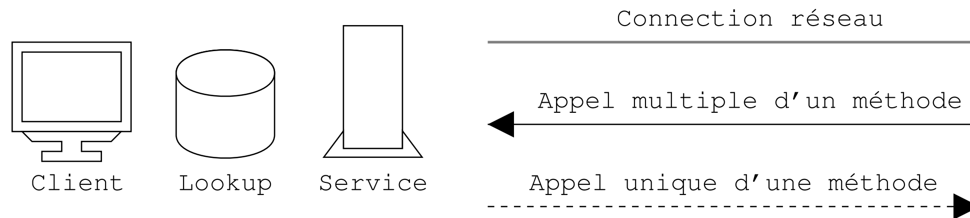

FIG. 2.1 – Conventions utilisées pour les diagrammes

2.3 Fonctionnement d'un système JINI

Un système JINI est articulé autour de trois axes:

- Les clients ;
- Le(s) lookup ;
- Les services.

Le point central est le lookup. C'est lui qui crée le lien entre les services et les clients. Nous aurons donc une structure en triangle avec pour sommet le lookup, et comme bases, un ensemble de clients et de services. Le schéma type est représenté à la Figure 2.2.

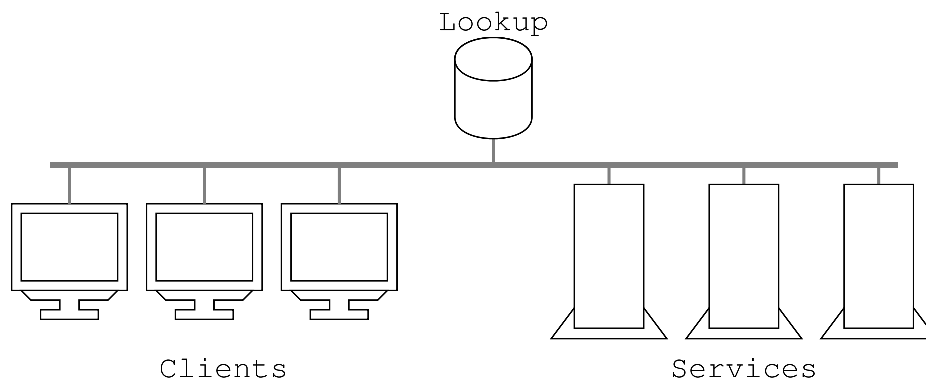

FIG. 2.2 – Structure d'un système JINI

Le fonctionnement du système s'articulera autour de trois opérations:

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

2.3.1 L'enregistrement d'un service auprès du lookup

La première chose à faire lorsqu'un service veut se faire connaître, c'est trouver un lookup auprès duquel s'enregistrer. Pour ce faire, il faut:
- Soit déjà connaître l'adresse d'un lookup ;
- Soit rechercher un lookup sur le réseau. Cette recherche se fait par l'envoi de paquets multicast sur le réseau. De leurs côtés, les lookup écoutent le réseau et répondent à ces paquets si besoin.

Une fois le lookup découvert, il faut s'enregistrer auprès de ce dernier. Pour ce faire, le service envoie un objet réalisant les interfaces souhaitées ainsi qu'un ensemble d'attributs. Ces objets se retrouveront alors dans la table de correspondance du lookup (voir Figure 2.3). La méthode d'envois des objets est le protocole RMI qui sera brièvement expliqué dans la Section 2.4.

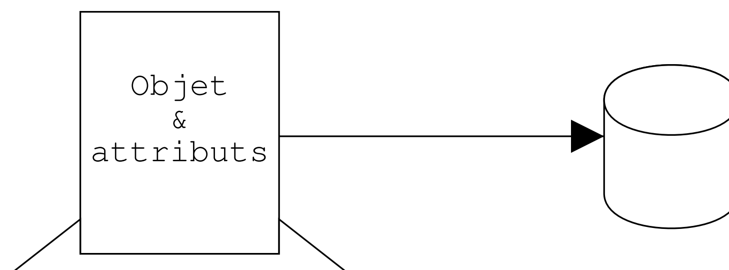

FIG. 2.3 – Passage d'objet et d'attributs du service vers le lookup

Lors de l'enregistrement avec le lookup, le service reçoit un numéro d'identification unique et un bail est créé. C'est le service lui-même qui décide des temps de renouvellement et de vie de ce bail. En général, le temps de vie du bail est infini. Le lookup conserve toujours le service. Par contre, le temps de renouvellement varie selon le type de service offert. Il restera alors au service à renouveler périodiquement son bail.

2.3.2 La recherche d'un service par le client

Lorsqu'un client recherche un service, il doit commencer par trouver un lookup. La découverte de celui-ci se fait de la même manière que pour le serveur: soit en connaissant l'adresse d'un lookup, soit en faisant une recherche par multicast.

Lorsqu'un lookup a été découvert, deux solutions s'offrent au client:
- Soit utiliser directement le lookup et faire une recherche sur une interface et un ensemble d'attributs afin de trouver un objet nécessaire ;
- Soit s'enregistrer auprès du lookup afin de recevoir une série d'événements. Ces événements peuvent être l'ajout, le retrait ou la modification d'un service. On peut spécifier pour chaque événement, le type d'interface à prendre en compte et éventuellement une série d'attributs qui serviront de filtre.

Il est possible de combiner ces deux méthodes afin, par exemple, de commencer par découvrir tous les services déjà sur le réseau, et ensuite s'enregistrer pour découvrir tous les services qui viendront s'ajouter par la suite.

Lorsqu'un service est découvert, le client reçoit du lookup, pour ce service, un `ServiceItem`. Cet objet contient:

- Le numéro d'identification unique du service ;
- L'objet réalisant l'interface souhaitée ;
- L'ensemble des attributs associés à ce service.

Il est dès lors aisé d'utiliser l'objet reçu (voir Figure 2.4).

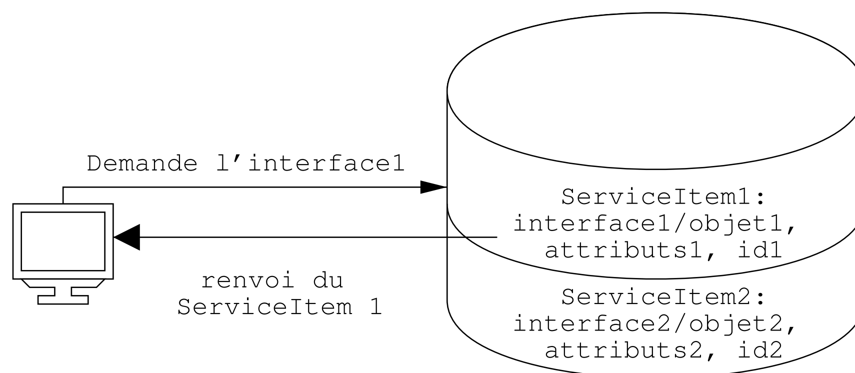

FIG. 2.4 – Passage d'objet, d'attribut, d'interface et de numéro d'identification du lookup vers le client

Dans le cas d'un événement, le client peut récupérer le `ServiceItem`, correspondant au service qui a déclenché cet événement. Il existe alors, deux `ServiceItem`, l'un correspondant au service avant l'événement, l'autre après l'événement (pour autant que cela ait du sens). On peut ainsi savoir, par exemple, quel est le service qui a disparu, en récupérant son numéro d'identification.

2.3.3 La connection du client et d'un service

Une fois que le client a récupéré un objet réalisant l'interface souhaitée, le dialogue entre le service et le client devient indépendant du lookup.

Il existe alors plusieurs types de service. Le premier type de service est réellement un objet qui peut exécuter les méthodes de l'interface demandée. Lorsqu'il est utilisé, il exécute le code de ses méthodes sur la machine cliente. Il n'y a alors pas à proprement parler de dialogue entre le client et le service puisque toute l'exécution se fait côté client. On peut, par exemple, utiliser ce type de service pour crypter des documents. Le service donne alors une méthode de cryptage, mais l'exécution de celle-ci se fait chez le client. Cela évite d'utiliser les ressources de la machine hébergeant le service.

Le deuxième type de service est un service utilisant une passerelle. L'objet qui est alors stocké dans le lookup sert à créer une connexion entre le client et le service. Les méthodes implémentées sont alors appelées à distance, du client vers le service. L'objet transféré sera, par exemple, un stub RMI (voir Section 2.4). La méthode de dialogue est représentée Figure 2.5, page 10. C'est ce type de service qui est utilisé dans ce projet. Il a l'avantage d'exécuter les méthodes sur la machine hébergeant le service.

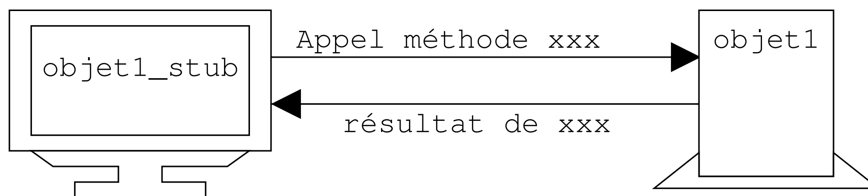

FIG. 2.5 – Dialogue entre client et service utilisant RMI

Il existe un troisième type de service qui est un hybride des deux premiers types. Il s'agit d'un objet dont une partie est exécutée par la machine du client et l'autre par la machine hébergeant le service. Ce type d'objet peut être utilisé, par exemple, pour afficher une interface graphique côté client tout en exécutant des méthodes sur la machine hébergeant le service.

Il est important de préciser que, la méthode de communication utilisée entre le client et le service peut-être quelconque. Elle est indépendante de JINI et même de Java. Il est ainsi possible d'utiliser des services fonctionnant dans d'autres langages et utilisant des protocoles qui leur sont propres.

2.4 Le protocole RMI

Le protocole RMI (Remote Method Invocation) est le protocole mis en place dans le langage Java pour les objets distribués. Il est livré avec tous les JDK (Java Developpement Kit), et est donc fortement standardisé.

RMI permet de faire appel à des méthodes distantes, en connaissant à priori l'URL de celles-ci. RMI est donc proche de JINI, mais il manque la découverte dynamique de cet URL. JINI apporte cette découverte ; lorsque l'on a trouvé l'URL de la méthode à invoquer, il ne reste plus qu'à utiliser RMI pour l'invoquer.

Seuls les principes généraux de fonctionnement d'RMI seront exposés ici.

2.4.1 Les objets sérialisés

RMI fait un grand usage des objets sérialisés de Java. Un objet sous la forme sérialisée est représenté par une suite d'octets. Cette suite représente les données nécessaires à la reconstruction de l'objet à partir de la définition de sa classe.

Pour désérialiser un objet, il faut donc deux éléments: la classe (et son bytecode) à laquelle il appartient et sa suite d'octets issus de la sérialisation.

Le mécanisme de sérialisation prend en charge le stockage des attributs de l'objet. Si ces attributs sont eux-mêmes des objets, ils sont à leur tour sérialisés.

2.4.2 Le code mobile

Puisqu'il y a moyen de sérialiser un objet, et donc de l'envoyer quelque part à travers le réseau, il faut pouvoir être capable de le désérialiser à l'arrivée. Pour ce faire, il faut disposer du bytecode de la classe à laquelle appartient cet objet. RMI met en place un mécanisme très simple pour récupérer ce bytecode: il utilise un serveur HTTP. Il est possible d'implémenter d'autres protocoles. Comme l'implémentation pour le protocole web est livrée avec RMI, c'est elle qui est utilisée en général.

Lors du lancement d'un serveur RMI, il suffit de spécifier un "codebase" pour pouvoir accéder à distance à la ressource voulue. Le "codebase" est un URL pointant vers le lieu de stockage des classes exportables.

2.4.3 Les interfaces et les serveurs RMI

Comme son nom l'indique, RMI (Remote Method Invocation) sert à invoquer des méthodes à distance. La définition des méthodes utilisables se fait par la définition d'une interface au sens Java du terme. Cette interface sera alors exportée par un serveur RMI.

Les serveurs RMI fonctionnent autour du "daemon" `rmid`. Ce programme se charge de créer un lien entre un nom d'URL et les interfaces exportées. Il écoute un port de communication(1099 par défaut) afin de répondre aux requêtes des clients.

A ce programme, viennent s'ajouter les différents objets Java exportant des interfaces. Il existe plusieurs méthodes pour exporter des interfaces:

- **La Méthode simple.**
  L'objet exportant l'interface est exécuté comme un autre programme. Il s'enregistre auprès d'`rmid` et attend les requêtes des clients. L'avantage de cette méthode est d'avoir un serveur toujours prêt. Le désavantage est de monopoliser les ressources du système car il ne tourne pas en tâche de fond. Cette méthode, avec quelques modifications est utilisée dans ce projet par le service de calcul. (voir Chapitre 3)
- **La Méthode avec lancement d'`rmid`.**
  Cette méthode est presque identique à la première. Elle ajoute cependant le lancement du "daemon" `rmid` au sein du serveur. L'avantage de cette méthode est de rendre l'utilisation d'RMI transparente à l'utilisateur, puisque celui-ci n'a alors plus à lancer " à la main" le "daemon" `rmid`.
  Cette méthode est utilisée dans ce projet, dans la version finale du client (voir Section 3.4).
- **Méthode des objets activables.**
  Cette troisième méthode est un peu plus compliquée à mettre en oeuvre. Plutôt que de bloquer les ressources du système, après s'être enregistré auprès du "dameon" `rmid`, le serveur se met en veille et rend la main. C'est alors au "dameon" `rmid` que revient la tâche d'activer l'objet serveur lorsqu'il y a une demande émanant d'un client.

Cette méthode est utilisée par `reggie`, le lookup fourni avec JINI.

Le mécanisme de communication entre un serveur et un client RMI passe par un “stub”. Ce morceau de code, généré par le compilateur `rmic`, à partir de la classe compilée du serveur, permet d'appeler, à travers le réseau, des méthodes, ainsi que d'envoyer les arguments nécessaires à ces appels. Lorsqu'un client à besoin de se connecter à un serveur RMI, il reçoit du “dameon” `rmid` un URL afin de pouvoir télécharger le “stub” du serveur. Une fois le “stub” téléchargé, les appels aux méthodes peuvent se faire directement par celui-ci.

Pour plus de détails quant au fontionnement de RMI, on peut consulter les spécifications [11].

2.5 Programmer avec JINI

JINI comme beaucoup de librairies Java fait appel à un modèle d'objets facilements réutilisables.

Il y a plusieurs niveaux de programmation JINI avec les classes correspondantes dans l'API (Application Programming Interface).

Le premier niveau est une série de classes prêtes à l'emploi, regroupées sous le nom de “Jini Helper Utilities” (voir [12]). Ces classes permettent de programmer facilement un service ou un client JINI sans entrer dans les détails d'implémentation. Tout programmeur voulant utiliser JINI commencera par ces classes.

Pour des cas plus particuliés, il peut être nécessaire de descendre d'un niveau, afin d'être plus près de l'implémentation même de JINI. Pour ce faire, on utilise les classes issues du “Core JINI” (voir [14]).

Il n'est pas réellement nécessaire de s'étendre plus sur la programmation JINI. On peut néanmoins consulter l'Annexe A ou le document [7]

2.6 Comparaison avec CORBA

Le but ici sera de différencier les deux technologies et principalement de voir ce que peut apporter JINI par rapport à CORBA.

Il faut rappeler que JINI fait intensément appel à RMI. Il sera donc plus correct de parler de JINI/RMI. Plusieurs mécanismes qui seront évoqués dans cette comparaison proviennent de RMI ; JINI en hérite donc.

Quelques fonctionnalités de JINI/RMI vont être exposées en recherchant la fonctionnalité correspondante dans CORBA. Les différences seront alors mises en évidence. Plusieurs document ont servi à réaliser cette comparaison outre les spécifications de JINI [13], des documents relatifs à CORBA [9] [6] et des documents de comparaison [10] [15] ont été consultés.

2.6.1 Langage de définition des interfaces

Dans les deux cas, RMI et CORBA, les objets exportés le sont via leur interface. Cette interface comprend toutes les méthodes sur ces objets appelables à distance.

Dans JINI/RMI, le langage de définition des interfaces exportables est Java. Il s'agit d'une simple interface qui a comme particularité d'étendre l'interface `Remote`. La définition est alors aisée ; toute interface standard Java peut être utilisée comme interface pour RMI. Le compilateur `rmic` se charge de la création du stub qui pourra être utilisé par le client. Le skeleton, est contenu dans l'implémentation de l'interface et géré par le "daemon" `rmid`.

Dans CORBA, l'interface est définie par un langage appelé IDL (Interface Definition Language). Ce langage se veut universel ; il doit pouvoir définir des interfaces pour des objets programmés en C/C++, SmallTalk, Java, Eiffel,... Un compilateur IDL se charge de la création des stub et des skeleton.

Dans le cas d'RMI, il est aisé de définir une interface, il suffit de connaître le langage Java. Dans le cas de CORBA, il faut apprendre un nouveau langage: l'IDL.

Dans le cas d'RMI, on est limité au langage Java ou à d'autres langage adaptés à la JVM (Java Virtual Machine). Dans le cas de CORBA, l'interface définie est indépendante du langage de l'implémentation.

2.6.2 Chargement dynamique d'interfaces

Il peut arriver que le client ne connaisse pas, à priori, l'interface dont il aura besoin. Le client n'a alors pas localement une copie du stub permettant de se connecter à cette interface. Il faut alors recréer ce stub ou le télécharger.

Le mécanisme gérant ce fonctionnement dans CORBA est appelé DII (Dynamic Invocation Interface). En gros, DII est une méthode pouvant produire à la demande, à partir d'une définition d'interface, un stub correspondant. Pour recréer l'interface, DII a besoin de réaliser la découverte des méthodes de cette interface.

Dans RMI, ce mécanisme est laissé à la JVM. En effet, si le client ne connaît pas une interface, RMI lui donne un URL où aller la chercher. Il reste alors à la JVM du client à télécharger le stub correspondant et à l'utiliser. Le tout se fait de manière transparente pour le programmeur. Il ne doit se soucier que de placer ses stubs sur un serveur web accessible. Autre avantage: le stub utilisé est le stub créé par le programmeur du serveur et pas un stub recréé.

2.6.3 Service de nommage

Pour pouvoir gérer plusieurs interfaces, il est nécessaire de les identifier. Cette identification passe par un service de nommage. Ce service permet à un client de rechercher un serveur à partir d'un nom (sous forme d'une chaîne de caractères).

Dans CORBA, cette méthode est appelée "Naming Service". Il permet de lier un objet à un nom et inversément.

Dans RMI, le service de nommage est pris en charge par le “daemon” `rmid`. Les fonctionnalités offertes sont les mêmes que celles offertes par CORBA.

2.6.4 Passage des paramètres

Les méthodes appelées à distance peuvent demander une série de paramètres et renvoyer une valeur. Il existe deux méthodes pour envoyer des paramètres: soit par référence, soit par valeur.

L’envoi par référence a l’avantage de ne presque pas prendre de temps ; surtout si la taille de l’objet est importante. L’envoi par valeur a l’avantage de pouvoir traiter cet objet de manière locale sans devoir faire appel à la communication réseau.

CORBA ne permet que l’envoi par référence ; bien que dans les dernières spécifications de CORBA, soit apparu le passage par valeur.

RMI permet sans problème le passage par référence et par valeur. Le choix est laissé au programmeur. Si on veut passer un paramètre par référence, sont type doit être une classe implémentant l’interface `Remote`.

Une petite différence entre CORBA et RMI au niveau du passage par valeur se situe au niveau de l’héritage. Dans CORBA, les sous-classes des paramètres sont tronquées à la classe parente. Par exemple: si une méthode a besoin d’un paramètre de type véhicule et que l’objet à passer est de type vélo (héritant de véhicule), CORBA ne fera que passer les attributs du véhicule. Le client ne peut donc pas savoir que l’objet qui lui a été donné était de type vélo à l’origine. RMI par contre conserve l’objet en entier. En effet, si la classe de l’objet n’est pas connue du côté client, elle est simplement téléchargée à partir d’un serveur web. Le programmeur doit juste s’assurer que la classe est disponible sur le serveur web.

2.6.5 Recherche de services

En plus d’un mécanisme de nommage, permettant de lier un nom à un objet, il peut parfois être utile d’avoir un mécanisme de recherche de service. Cette recherche doit être faite lorsque l’on ne connaît pas à priori la machine sur laquelle se trouve le service.

En CORBA, cette recherche prend forme sous le nom de “Trading Service”. Ce service permet de faire une recherche sur les méthodes et les attributs de l’objet distant souhaité.

La recherche de services est un des points clefs de JINI. Elle est implémentée par le service appelé “Lookup”. Avec JINI, cette recherche peut non seulement se faire sur l’interface, mais aussi sur les attributs ou sur le numéro d’identification unique associé à chaque service.

Alors qu’en CORBA, le “Trading Service” renvoie une référence vers l’objet qu’il a trouvé, JINI/RMI renvoie l’objet lui-même. L’objet renvoyé par JINI peut alors être un objet à part entière dont les méthodes s’exécutent sur la JVM du client ou un proxy permettant de se connecter à une ressource distante. Ce proxy peut être un `stub` RMI si on souhaite appeler des méthodes à distance, mais il peut aussi être un système de communication par sokets ou un stub CORBA. Il s’agit d’un objet exécutable par la JVM du client.

Un plus pour JINI est l'utilisation de la recherche multicast par les services. En effet, un service pour s'enregistrer commence par contacter tous les lookup dont il connaît l'URL et s'y enregistre. Ensuite, il fait une recherche sur tout le réseau local afin de découvrir des lookup. Lorsqu'il en trouve un, il s'enregistre.

JINI apporte aussi la gestion de la disparition de services. Grâce au système de bail, le lookup ne contient que des services accessibles. Lorsqu'un service vient à disparaître, le client peut en être averti par un mécanisme d'évènements.

CORBA est avant tout un système de communication entre services distants, par appel de méthodes distantes.

JINI est un système de communication entre JVM. La communication peut se faire par appel ou envoi d'objets.

2.6.6 Conclusion

Les deux approches analysées se valent. Elles sont toutefois différentes et l'une sera utilisée préférentiellement à l'autre selon le type d'application envisagée. S'il est impératif d'être indépendant du langage de programmation, CORBA semblera indiqué. Si par contre, l'indépendance matérielle est importante, JINI sera préféré.

Il est, de plus, tout à fait possible de faire interopérer les deux solutions. En réalité, JINI/RMI peut encapsuler des services CORBA. En effet, il existe une version de RMI tournant sur IIOP (Internet inter-ORB Protocol) et IIOP permet la connection aux ORB CORBA.

Chapitre 3

Architecture pour le calcul parallèle

3.1 Introduction

Les différents mécanismes offerts par JINI pour la programmation en réseau: gestion des apparitions, et disparitions de services, système d'attributs, appels de méthodes à distances, pas de configuration d'adresse, ... semblaient intéressants à appliquer au calcul parallèle.

L'application de JINI au calcul parallèle peut se faire de différentes manières. Il a donc fallu choisir une architecture sur laquelle se baser pour effectuer le calcul, ainsi qu'un calcul à paralléliser.

Le choix s'est porté sur un calcul simple. Il s'agit d'une multiplication de matrice. Ce choix présente plusieurs avantages: temps d'analyse de l'algorithme et de sa parallélisation réduits, temps d'implémentation réduit, facilités pour créer des données de test et les résultats correspondants.

L'architecture de base quant à elle a plusieurs fois été modifiée. Plusieurs possibilités ont donc été testées et développées. Un premier choix à faire lorsque l'on veut créer un système basé sur JINI est de définir ce que sera un service et ce que sera un client. Il existe deux possibilités, pour le calcul parallèle:

- **Les machines de calcul sont des clients et le demandeur de calcul un service.**
  Dans ce cas, les clients qui se connectent, recherchent un service qui leur propose de faire un calcul. Ils prennent un calcul, l'exécutent et renvoient le résultat et recommencent l'opération autant de fois qu'il le faut.
- **Les machines de calcul sont des services et le demandeur de calcul un client.**
  Dans ce cas, les machines prêtes à fournir du temps de calcul sont des services. Le client est une machine qui a besoin de ce temps de calcul. Le client cherche alors, grâce au lookup, les machines services utilisables, en sélectionne certaines et leur fait exécuter du code.

Les deux solutions sont tout à fait viables. Néanmoins, dès le départ, il semble logique de placer les machines de calcul comme des services offrant du temps CPU. Cette vision des choses adhère parfaitement à la vision JINI.

Dans la première solution, la sélection des machines à utiliser doit impérativement se faire sur base d'un dialogue entre client et service et n'utilise donc pas le système d'attributs mis en place par JINI. De plus, fondamentalement, ce n'est pas la machine qui calcule qui a besoin pour sa gestion de connaître tous

les demandeurs de calcul qui existent, mais plutôt l'inverse. C'est pourquoi, le choix de base s'est porté sur la deuxième solution, les machines de calcul sont des services, le demandeur de calcul un client.

Après avoir choisi cette base, il faut construire la communication entre le client et le service. Ici aussi, plusieurs solutions sont possibles, quelques-unes ont été analysées ici. Toutes ces solutions ont fait l'objet d'un développement logiciel et ont été testées.

3.2 Architecture dédiée

La première solution qui vient à l'esprit est de réaliser une solution axée sur le calcul à effectuer. Cette solution est assez simple à mettre en oeuvre et on arrive assez rapidement à de bons résultats.

Dans ce cas-ci, l'architecture est totalement dédiée. La Figure 3.1 résume assez bien ce concept dans le cas d'une multiplication de matrices.

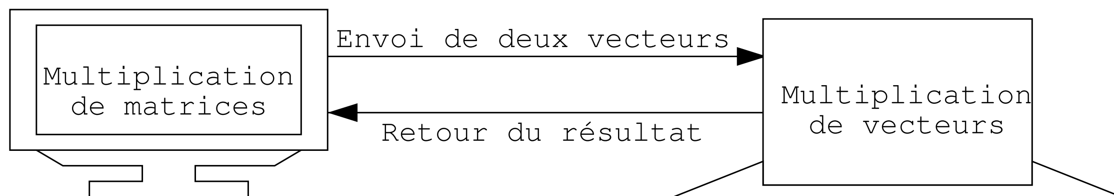

FIG. 3.1 – Architecture dédiée, cas de la multiplication de matrices

3.2.1 Interface du service

L'interface du service de calcul de matrices qui est lancé sur chaque machine offrant sa puissance, est assez simple:

```
public interface ComputingInterface{
    public void initCalc () throws RemoteException;
    public void addLine(int lineNumber, double[] line ) throws RemoteException;
    public void addColumn(int columnNumber, double[] col) throws RemoteException;
    public void doCalc () throws RemoteException;
    public int getLine () throws RemoteException;
    public int getColumn() throws RemoteException;
    public double getResult () throws RemoteException;
    public void endCalc () throws RemoteException;
}
```

A côté de cet interface, un attribut JINI spécifique pour le calcul a été créé. Cet attribut permet de transférer l'état actuel du service. Au départ, le service est dans l'état `READY`. Lorsque l'on modifie cet état, le lookup en avertit le client qui réagira en conséquence.

Voici une description des différentes méthodes:

- **initCalc, endCalc.** Ces deux méthodes permettent de changer l'état du service. initCalc fait passer le service dans l'état `CONNECTED` ; endCalc le fait passer dans l'état `READY`.
- **doCalc.** Cette méthode fait passer le service dans l'état `COMPUTING` et lance un “thread” qui réalise la multiplication de la ligne et de la colonne en cours. Lorsque le calcul est terminé, le service passe dans l'état `DONE`.
- **addLine, addColumn.** Ces méthodes permettent de transférer une ligne et une colonne du client vers le service. Les données importées deviennent respectivement la ligne ou la colonne en cours.
- **getLine, getColumn.** Ces méthodes renvoient le numéro de la ligne ou de la colonne en cours.
- **getResult.** Il envoie le dernier résultat calculé.

3.2.2 Le client

Le client doit se charger de la gestion du calcul. Il est composé de trois méthodes principales qui correspondent directement aux méthodes du Listener d'événements JINI:

- **serviceAdded.** Cette méthode est appelée lorsqu'un nouveau service apparaît dans le réseau. Dans ce cas-ci, la réaction du client à un ajout de service est de tester si ce service est dans l'état `READY`. Si c'est le cas, le service est utilisé ; sinon, il est simplement ignoré.
- **serviceRemoved.** Dans ce cas, le service qui a été enlevé est recherché dans la table des services utilisés. S'il s'y trouve, on remet la partie de calcul qui lui était affectée dans la liste des parties non encore calculées.
- **serviceChanged.** Cette méthode est la méthode la plus importante. Elle se charge des transitions entre états des services. Comme elle connaît le nouvel état, elle est capable de déterminer la séquence d'actions à effectuer pour parvenir dans l'état suivant.
  Cette méthode se charge aussi de l'ajout des services qui n'étaient pas `READY` et qui le deviennent.

Le client tient une table des services utilisés en correspondance avec la partie de calcul qu'ils sont occupés à exécuter. Les services sont stockés par leur numéro d'identification unique. Cela permet de facilement déterminer quel était la partie de calcul en cours lorsqu'un service disparaît.

3.2.3 Déroulement d'un calcul

La Figure 3.2, page 19, illustre conceptuellement le déroulement des opérations dans le cas idéal où un service devient disponible et effectue tous les calculs qui lui sont demandés.

Le déroulement du calcul est assez systématique. Chacun de leur côté, clients et services s'enregistrent auprès du lookup. Les clients s'enregistrent afin d'écouter les événements qui les intéressent. Les services enregistrent leur objet implémentant l'interface de calcul et leurs attributs. Le dialogue du client vers le service se fait par RMI et donc, l'objet enregistré est le “stub” RMI.

Lorsque le client découvre un service dans l'état “READY”, il l'utilise. Le reste du dialogue est alors une suite:

- de changements d'états du service,
- d'envois d'événements du lookup vers le client,
- de réactions du client par des appels à des méthodes distantes vers le service.

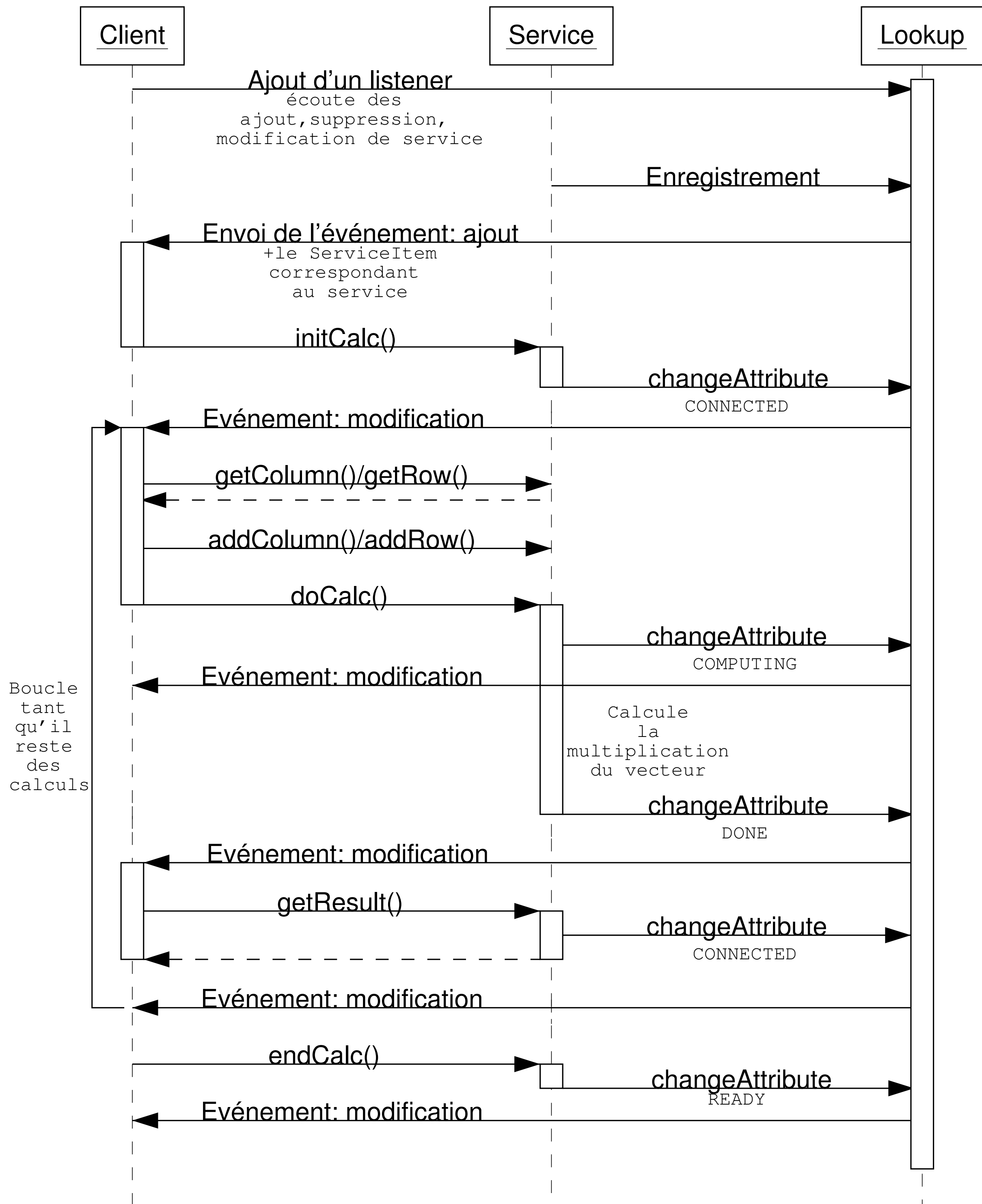

FIG. 3.2 – Diagramme de séquence pour la communication avec l’architecture dédiée.

Cette suite se répète tant qu'il y a des calculs à réaliser. Elle se termine par un dernier changement d'attributs du service qui retourne dans l'état "READY"

Cette première méthode fait intensément appel au changements d'attributs du service. Ces changements demandent un appel au lookup qui lui doit propager l'information vers les clients intéressés. Cette méthode entraîne un trafic réseau intense autour du lookup et donc des temps de latence importants. Une première chose à faire pour réduire le trafic est de diminuer le nombre d'états possibles. La seconde chose à faire est d'établir une connection bidirectionnelle entre le client et le service. Ces solutions sont implémentées dans la troisième méthode (voir Section 3.4).

L'utilisation d'une architecture dédiée, bien que très simple au niveau des concepts, n'apporte pas plus en comparaison avec des solutions ou des projets déjà existants. On peut trouver sur Internet des projets tels "Seti@home" (voir leur site [4]) qui utilisent le même principe. Cette architecture a cependant le mérite d'illustrer la faisabilité d'un système de calcul parallèle basé sur JINI. Elle a aussi, au cours du développement vers l'application finale, permis de mettre en évidence certains problèmes.

3.3 Architecture généraliste, avec modifications de statut

L'étape suivant a été d'intégrer à l'architecture le mécanisme de code mobile offert par RMI. Ce système offre un modèle plus souple. En effet, en utilisant le code mobile, il est possible de passer un objet comme paramètre d'une méthode. Cet objet peut contenir tout ce que le programmeur désire. Dans ce cas-ci, l'objet contiendra à la fois les données du calcul, comme pour l'architecture dédiée, mais aussi la méthode de calcul. Le fonctionnement du service est alors très simple:

- il reçoit un objet (qui sera appelé "Runner" par la suite),
- il exécute une méthode particulière de cet objet (par exemple doCalc()),
- il renvoie l'objet à l'expéditeur, avec le résultat contenu dans cet objet.

Le tout fonctionne comme un tour de passe passe, le service ne sait pas du tout ce qu'il va exécuter comme calcul, mais il reçoit du bytecode Java, l'exécute et renvoie le résultat.

Un schéma reprenant les différentes ineractions possibles entre client, lookup et service est donné Figure 3.3

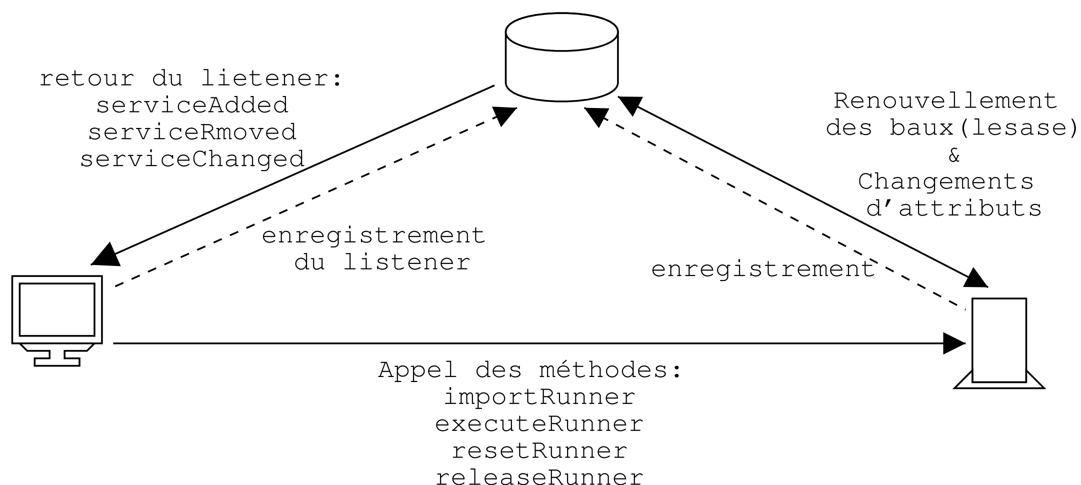

FIG. 3.3 – Interactions possibles pour une architecture générique avec modifications de statut.

Le nombre d'états possibles a aussi diminué. Il n'y a plus que trois états: READY, BUSY, DONE. Ce qui permet de diminuer les temps de latence.

3.3.1 Interface du service

L'interface du service est totalement différente de celle qui existe pour l'architecture dédiée. Sa fonction principale est l'importation et l'exportation du Runner. Voici la définition Java de cette interface:

```
public interface ComputingInterface{
    public void importRunner(ComputingRunner cr) throws Exception;
```

```
        public void executeRunner() throws Exception;
        public void resetRunner () throws Exception;
        public ComputingRunner releaseRunner() throws Exception;
    }
```

Le comportement de ces méthodes est assez simple en soi:
- **importRunner.** Se charge d'importer le Runner et de l'initialiser.
- **executeRunner.** Se charge de lancer un "thread". Lequel exécute le Runner.
- **resetRunner.** Remet le Runner à zéro et arrête le "thread" d'exécution si celui-ci existe.
- **releaseRunner.** Renvoie au client le Runner exécuté.

Outre le service, le Runner est lui même une interface qui sera implémentée différemment selon le calcul à effectuer.

3.3.2 Interface du Runner

Chaque action que le client peut avoir sur le service, doit être répercutée sur le Runner. L'interface du Runner est donc assez similaire à celle du service:

```
    public interface ComputingRunner{
        public void init ();
        public void execute ();
        public void reset ();
    }
```

Quelques mots d'explications pour ces méthodes:
- **init.** Cette méthode est appelée après que le Runner ait été importé. Elle réalise quelques fonctions d'initialisation si nécessaire.
- **execute.** La méthode principale. C'est elle qui contient l'exécution de la portion de calcul assignée au Runner.
- **reset.** Une méthode de remise à zéro. Si au sein du Runner des variables globales sont modifiées, cette méthode est le moyen de les remettre à une valeur initiale.

3.3.3 Le client

Le fonctionnement du client reste, de manière conceptuelle, plus ou moins le même. Du point de vue de l'implémentation, le fait de pouvoir réaliser plusieurs types de calculs, entraîne une recherche de modularité. C'est pourquoi, pour cette deuxième itération du modèle, le client a subi une refonte complète. Le diagramme de classe Figure 3.4, page 23 donne un aperçu de la structure du client.

Une séparation a été faite entre la gestion réseau et la gestion du calcul. Ainsi, une couche intermédiaire a été créée afin de simplifier la programmation d'un gestionnaire de calcul. Cette couche se décompose en deux classes:

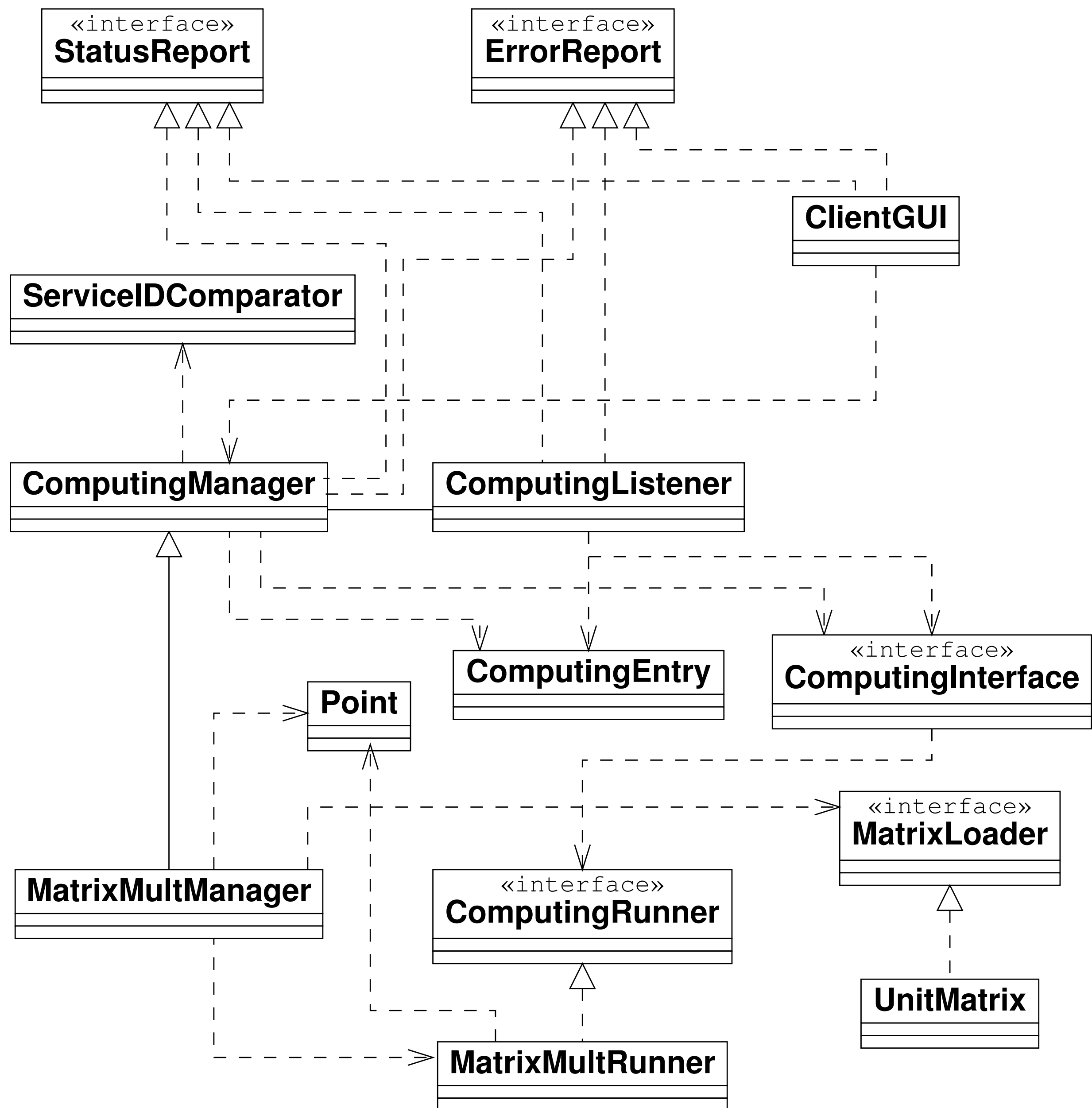

FIG. 3.4 – Diagramme de classes du client (Cas générique avec attribut)

– **ComputingListener.** Cette classe prend en charge toute la partie JINI. C'est elle qui intercepte les ajouts, modifications et retraits de services. Ces différentes méthodes exécutent un premier filtrage des événements et des services disponibles avant de passer la main à des méthodes plus ciblées de la deuxième classe.
– **ComputingManager.** Il s'agit en réalité d'une classe abstraite. Cette classe intègre une table de correspondance permettant de relier un service avec un objet générique. Cet objet générique permet, par exemple, dans le cas d'une multiplication de matrice, de stocker le point calculé par le service correspondant.
  Outre la gestion de la liste, cette classe demande à être surchargée pour la gestion des événements liés aux services: ajout, retrait, exécution, libération et sélection. Elle permet aussi la gestion du statut du calcul: pourcentage effectué (évolution), calcul terminé ou non.

Sur cette couche vient alors se greffer deux classes spécifiques à la multiplication de matrices:

– **MatrixMultManager.** Cette classe étend ComputingManager. Elle réalise l'implémentation des différentes méthodes abstraites tout en spécialisant le calcul à la multiplication de matrices.
– **MatrixMultRunner.** Implémente l'interface ComputingRunner et réalise le produit scalaire de deux vecteurs.

3.3.4 Déroulement d'un calcul

La séquence des différents messages envoyés entre les différents intervenants est illustrée Figure 3.5, page 25.

On peut tout de suite remarquer la diminution des appels vers le lookup, ce qui permet de diminuer les temps de latence.

Au niveau de la gestion des événements, la classe `ComputingListener` s'occupe de les récupérer, de les analyser et d'appeler les méthodes du `ComputingManager` en place. Les méthodes appelées dépendent de l'événement entrant et de l'état du service.

Le déroulement en lui-même est axé entièrement sur l'envoi et la récupération d'un objet. Le client envoie son Runner, le service l'initialise et change son statut lorsqu'il est prêt. Le client demande alors au service d'exécuter le Runner.

Du côté service, est alors créé un "thread" qui se chargera d'effectuer le calcul. Lorsque ce "thread" est terminé, le service passe dans l'état `DONE`. Le client sait alors qu'il peut récupérer le résultat.

Une fois le résultat récupéré, le service retourne dans l'état `READY` et le Runner est détruit. Le lien entre un service et un client est donc très volatil. En effet, comme après chaque calcul le service revient dans l'état `READY`, la puissance de calcul du service peut être utilisée par un autre client. Le fait de détruire le Runner n'est pas non plus optimal car pour chaque calcul, il faudra envoyer à la fois les données et la méthode. Alors que si un même client utilise deux fois de suite le même service, il est inutile de renvoyer la méthode de calcul car seules les données changent. Une solution est proposée dans la Section 3.4.

L'option de créer un "thread" du côté service est plus économique en mémoire et en temps CPU que la solution de créer un "thread", du côté client, pour chaque service en train de calculer. Cette solution qui paraît économique au point de vue ressources du client pose néanmoins un problème: elle exige l'envoi d'un événement ce qui demande du temps. Une solution existe pour éviter ce problème: l'utilisation d'RMI à la fois du côté service et du côté client.

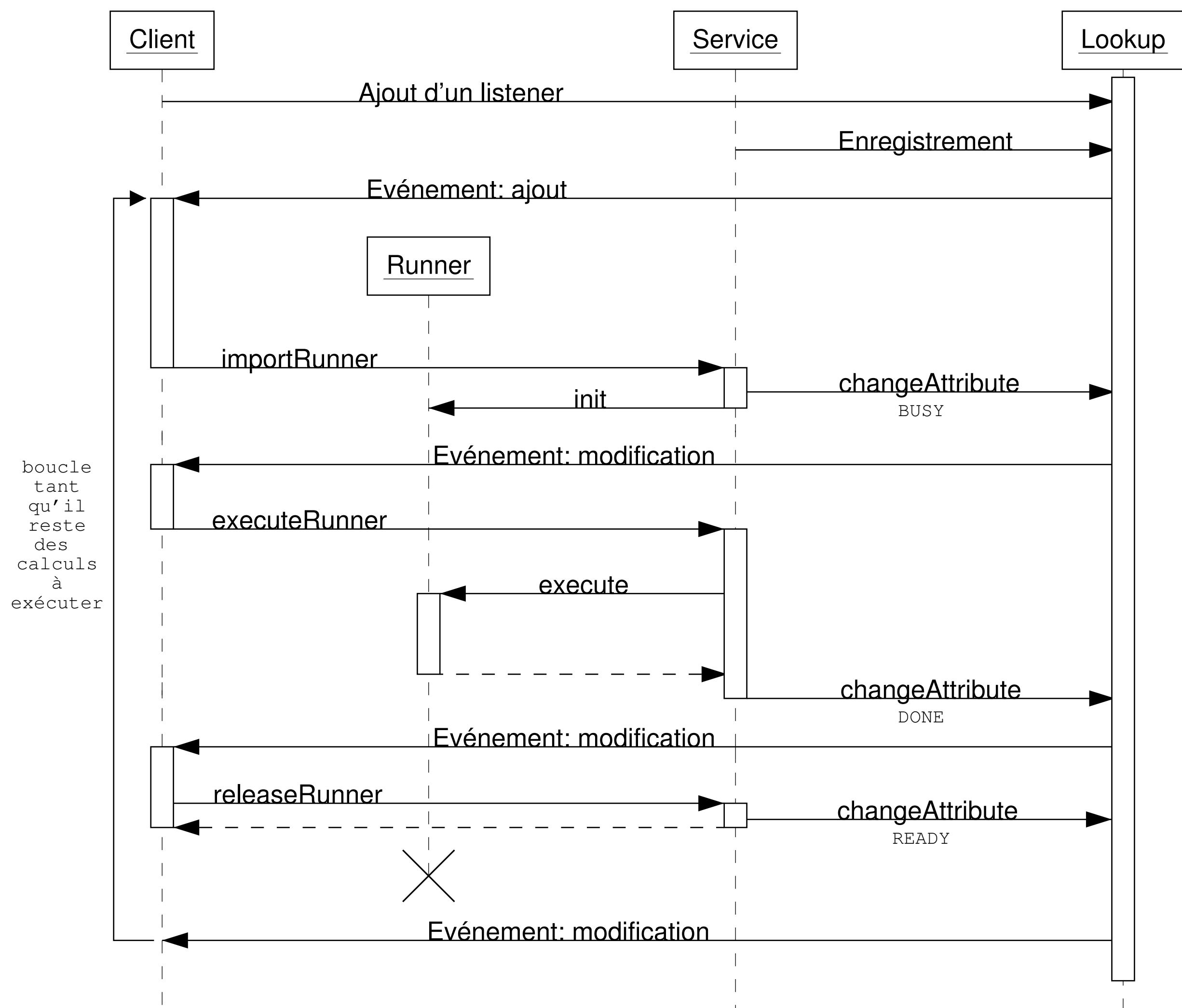

FIG. 3.5 – Diagramme de séquences (Cas générique avec attribut)

3.4 Architecture généraliste, sans modifications de statut

Les architectures précédentes étant incomplètes, il fallait encore améliorer le concept. Cette amélioration devait se faire au niveau du dialogue entre le client et le serveur. Pour s'affranchir des changements trop réguliers d'état au niveau du lookup, il faut établir une communication directe du service vers le client. Cette communication se fera par des appels à des méthodes distantes du service vers le client. Ces appels passeront bien entendu par le protocole RMI.

Un schéma des différentes interactions possible entre client, lookup et service est donné Figure 3.6. En comparaison avec la Figure 3.3, page 21 il faut noter:
- Qu'il existe des méthodes de retour du service vers le client ;
- Que le nombre de changements d'état, via les attributs, a diminué.

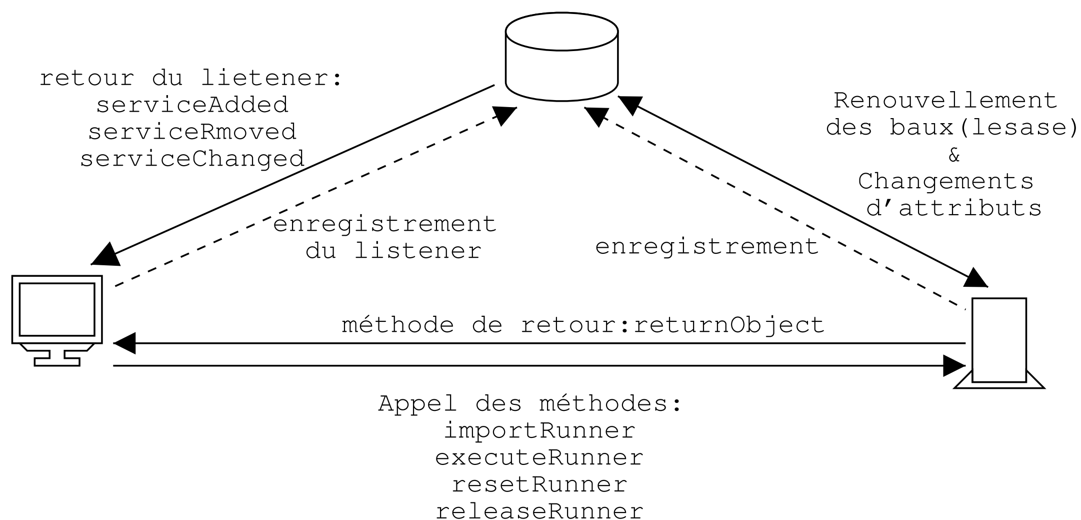

FIG. 3.6 – Interactions possibles pour une architecture générique sans modifications de statut

Ici encore, le nombre d'états possibles pour le service a diminué. Il est possible avec deux états de totalement caractériser le service. En effet, toutes les autres transistions se font par appel RMI. Il ne reste donc plus que les états `READY` et `BUSY`.

3.4.1 Interfaces du service et du Runner

Ces interfaces n'ont pas subi de grands changements. L'interface du service est restée identique. Celle du Runner s'est vue ajouter une méthode `stop()`. Cette méthode permet d'arrêter un calcul en cours sur le service. Il faut savoir qu'en Java, la destruction des objets est laissée au GarbageCollector. L'utilisateur n'a donc pas le contrôle sur le moment de la destruction d'un objet. Ainsi, si notre objet n'est plus utilisé par le service, mais existe encore dans la mémoire de la JVM, il se peut qu'il continue à effectuer son calcul, utilisant inutilement des ressources CPU.

L'implémentation de ces interfaces a bien entendu été modifiée afin de répondre aux nouvelles spécifications du client. La méthode `init()` du Runner réalise maintenant la connection avec le client via RMI. L'adresse du client est passée au constructeur du Runner. On envoie ainsi le Runner avec l'URL de connection en une fois au service.

3.4.2 Le client

Le client a subi un profond changement. Il a dû devenir un serveur RMI à part entière. Pour ce faire, il implémente l'interface suivante:

```
public interface ComputingClientListener extends Remote{
    public void returnObject(ServiceID id, Object obj) throws RemoteException;
}
```

Cette interface permet de renvoyer un objet générique du service vers le client. Cette interface constitue une base de travail. Pour la multiplication de matrice, il a été nécessaire d'y ajouter une autre interface:

```
public interface MatrixMultInterface extends Remote{
    public Point getPoint(ServiceID id) throws RemoteException;
    public double[] getRow(int i) throws RemoteException;
    public double[] getColumn(int i) throws RemoteException;
}
```

Cette interface permet l'établissement d'un réel dialogue entre le service et le client. Il n'est plus nécessaire d'envoyer plusieurs fois le même algorithme de calcul vers le client. Il suffit d'envoyer un runner comprenant une méthode de calcul ainsi qu'une méthode de chargement des données à distance.

3.4.3 Déroulement d'un calcul

Comme pour les deux autres types d'architecture, le déroulement d'un calcul est illustré par un diagramme de séquences Figure 3.7, page 28.

Ce diagramme met en évidence le nombre restreint de communications avec le lookup. De même, la taille de la boucle est restreinte, ce qui entraîne une diminution de la taille des objets transférés entre le client et le service.

Le rôle du client dans la gestion des services est primordial. En effet, il est responsable de libérer les services lorsqu'il n'en a plus besoin. Si un client gère mal ses services, il se pourrait que tous les services qu'il a utilisés restent en mode `BUSY` et ne soient donc plus disponibles pour les autres clients. Pour éviter ce problème, une fois que le Runner a fini son exécution, le service se met en attente d'un release. Si au bout d'un temps de pause, la méthode releaseRunner n'a pas été appelée, le service considère que la connection avec le client a été perdue. Il se remet alors dans son état initial(`READY`) et le Runner est détruit.

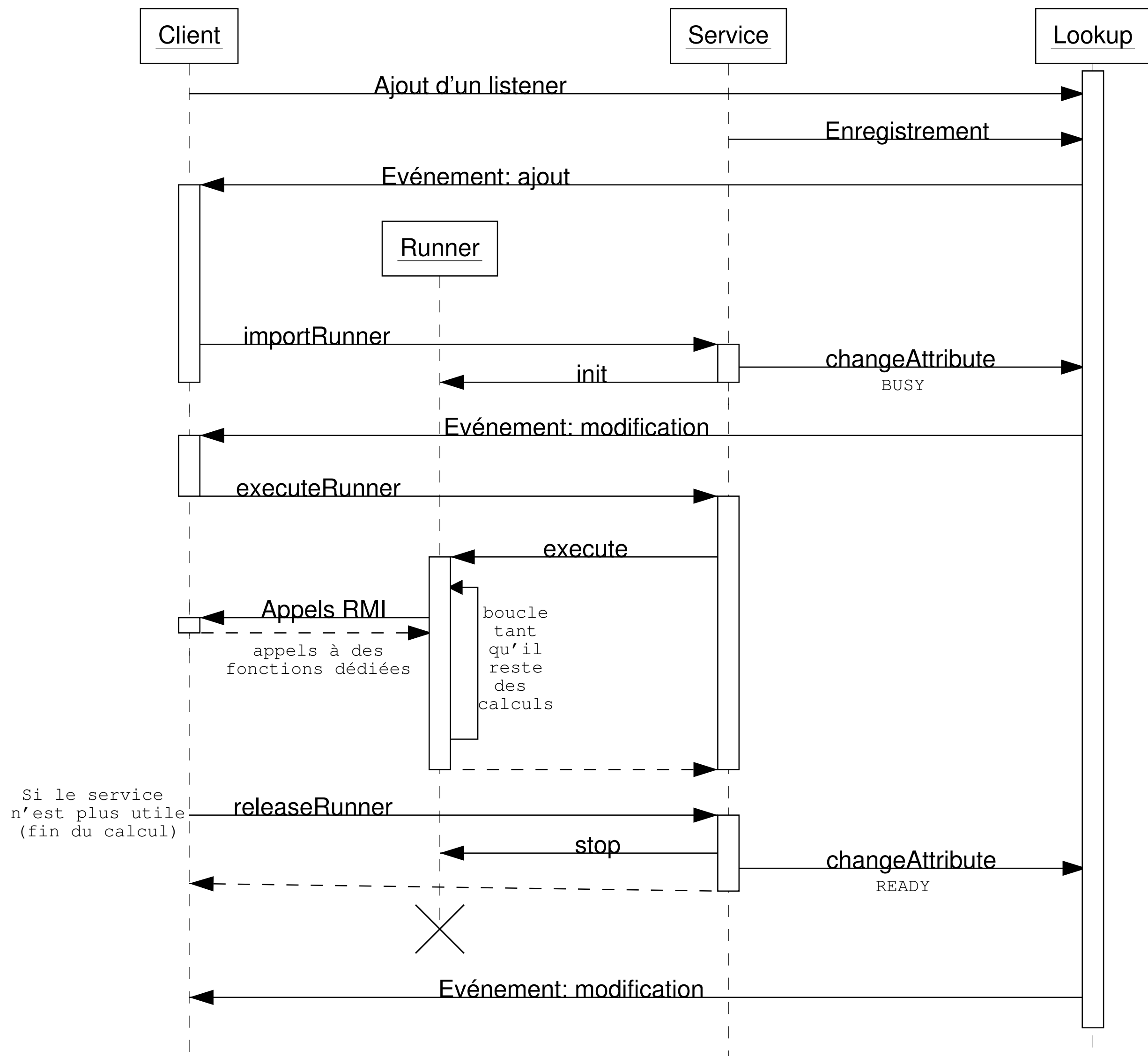

FIG. 3.7 – Diagramme de séquences (Cas générique sans attribut)

De même, si une erreur survient lors d'un appel à une méthode distante du client, le service repasse immédiatement dans l'état initial. Ceci améliore la robustesse du système: le service est le plus possible disponible pour du calcul.

Cette méthode a été implémentée avec plus de soin et testée plus en profondeur que les deux autres méthodes. Plus de détails quant à l'implémentation et aux résultats des tests seront donnés dans le Chapitre 4.

3.5 Autres améliorations possibles

Il est toujours possible d'améliorer un modèle. Dans ce cas-ci, puisque le but du système est le calcul parallèle, il peut être nécessaire d'essayer d'accélérer les communications et le calcul.

3.5.1 Modification d'architecture

Une première idée pour accélérer les communications serait d'éliminer la phase d'importation. On pourrait utiliser la méthode `executeRunner` pour importer et démarrer le calcul. Cette méthode a l'avantage et l'inconvénient de supprimer l'attente de réponse du lookup. Il existe un avantage au niveau gain de temps, mais d'un autre côté, attendre que le lookup envoie l'événement de changement d'attribut assure que ce changement est bien effectif et qu'aucun autre client n'utilisera ce service.

3.5.2 Compression

Une deuxième idée pour accélérer la communication est de compresser les données. Il est évident que si le volume de données à transférer est important, il est nécessaire de tout faire pour en réduire la taille. Si les procédés de stockage classique(ex: liste liée pour les matrices creuses) ne sont pas suffisants, il suffit d'utiliser le package `java.utils.zip` qui est livré avec toutes les JVM pour pouvoir compresser les données.

3.5.3 Code natif

Un troisième axe à explorer afin d'accélérer le calcul est l'utilisation de code natif. Evidement, l'utilisation de code natif demande une recompilation pour chaque plate-forme utilisée. Cette vision des choses est donc incompatible avec la notion de code mobile qu'apporte l'architecture JINI. Il y a cependant une solution à ce problème: la création d'une librairie mathématique. Ce genre de librairie n'existe pas encore en standard pour le langage Java, si ce n'est les fonctions de bases (sinus, cosinus, ...). Le but serait donc de créer une librairie optimisée pour certains calculs tels la multiplication de matrices, de vecteurs, ...

Cette librairie pourrait alors être programmée entièrement en Java pour les ordinateurs d'architecture “exotique”. Pour tous les systèmes et OS courants, il serait alors possible d'implémenter une partie de cette librairie en code natif afin d'accélérer le plus possible le temps d'exécution.

3.5.4 Sélection

Un quatrième axe peut être exploré afin d'essayer d'accélérer le calcul. Il s'agit de la sélection des services à utiliser, ainsi que le suivit de leurs performances. Par exemple, prenons deux services, tous deux immédiatement disponibles. L'un d'entre eux (appelé L) met en moyenne 5 unités de temps pour effectuer un bloc de calcul. L'autre service (appelé R) met 1 unité de temps en moyenne pour effectuer le même bloc. Admettons que le calcul compte 10 blocs.

- **Sans sélection,** le client choisira tous les services disponibles. Il donnera donc du travail au service L et au service R. Sur le temps que L calcule un bloc, R en calcule 5. C'est ainsi qu'au bout de 5 unités de temps, il y a 6 blocs calculés. Si il n'y a pas de sélection, le client continuera à utiliser L et R. Le problème est qu'il ne reste que 4 blocs de calculs. Si le client en assigne un à L et un à R, R va effectuer 3 calculs. Le client devra alors attendre que L ait fini. Le calcul durera en tout 10 unités de temps.
- **Avec une sélection du plus rapide,** on n'utilise que les services les plus rapides. Dans ce cas-ci, L n'intervient plus. Le temps de calcul est alors de 10*1 unité= 10 unités de temps.
- **Avec une sélection intelligente,** il faut analyser quelques données sur les services utilisables, comme par exemple, leur temps de calcul pour un bloc donné. On pourra alors déterminer s'il est avantageux de donner un calcul à un service ou non.
  Dans ce cas-ci, L et R commencent tous deux à travailler. Au bout de 5 unités de temps, 6 blocs sont calculés. Il n'est clairement pas avantageux de donner encore un bloc à L ; donc R terminera le calcul. Le résultat est un temps de calcul pour les 10 blocs de 9 unités de temps.

L'exemple ci-dessus illustre bien la problématique du choix des calculateurs et de la politique à adopter.

Durant les tests du projet, la sélection des service a été implémentée. Elle constistait à utiliser tous les services disponibles ; on se retrouve alos dans le cas "sans sélection". Cette sélection qui peut paraître inutile était nécessaire pour introduire le modèle de sélection dans la structure orientée objet (voir Section 4.2). Il sufffit dès lors de surcharger une méthode pour pouvoir définir une autre méthode de sélection.

Toujours dans la sélection, mais à un autre niveau, il peut être intéressant de n'utiliser qu'une partie des services et de laisser les autres disponibles pour d'autres tâches.

De plus, en se basant sur la sélection, il est possible de définir un mécanisme de priorité afin de pouvoir créer une hiérarchie d'importance entre les calculs. Par exemple, le développeur qui veut tester un nouvel algorithme en réseau, pourrait le faire avec une faible priorité car dans ce cas, les performances ne sont pas importantes. Ce qui compte, c'est le test à grande échelle. A l'opposé, on trouvera le calcul scientifique demandant beaucoup de ressources, avec un niveau de priorité élevé. Dans ce cas, les performances sont très importantes.

3.5.5 Et d'autres...

Bien entendu, il existe encore d'autres améliorations possibles pour cette architecture, comme par exemple, la sécurisation des transferts de données, l'utilisation de bases de données pour accéder aux données, … L'architecture proposée ici amène cependant de nombreux avantages, et l'implémentation

qui a été réalisée, bien qu'imparfaite, permet de facilement intégrer toutes ces améliorations sans grande difficulté.

Chapitre 4

Plate-forme de test et résultats expérimentaux

4.1 Introduction

Après avoir décrit les différentes architectures possibles, une description de l'implémentation qui a été réalisée sur base de l'architecture générique sans modification du statut est nécessaire.

Cette implémentation a été créée de manière itérative à partir de l'architecture dédiée. Elle a alors suivi l'évolution du projet jusqu'à la dernière architecture. On peut donc dire que toutes les architectures ont été testées, mais seule la dernière a été pleinement finalisée.

Cette plate-forme a été utilisée pour les tests expérimentaux à grande échelle. Elle a permis de mettre en évidence les avantages et les inconvénients de l'architecture JINI.

4.2 Plate-forme de test

Dès le début du projet, le développement a été axé autour de la réutilisation de code. La solution qui en découle peut donc facilement servir de plate-forme pour d'autres expérimentations.

La plate-forme a été divisée en 4 "package" Java:

- **common** qui contient des classes et des interfaces communes au client et au service. Par exemple, l'interface `ComputingInterface`.
- **server** qui contient toutes les classes et interfaces nécessaires à la mise en oeuvre d'un service générique de calcul. Ce service implémente l'interface `ComputingInterface`.
- **client** qui contient les classes et interfaces de base nécessaires à la création d'un client, ainsi qu'un petit utilitaire visuel permettant de lancer et d'arrêter un calcul. Cet utilitaire permet aussi la visualisation des noms des ordinateurs travaillant pour le client.
- **matrixmult** qui contient tout ce qui est nécessaire à la multiplication de matrice: à la fois le Runner et le Manager, plus quelques autres classes utilitaires tel le chargement d'une matrice en mémoire.

Les classes qui composent ces différents package sont regroupées dans des fichiers JAR (Java ARchive) afin de faciliter leur utilisation. Il existe un fichier pour chaque package, ainsi que trois fichiers plus spécifiques composés des éléments téléchargeables de chaque package. Cela permet de transférer facilement le système d'un serveur http à un autre.

Les différents package vont être passés en revue ci-dessous. Une documentation plus détaillées des différentes méthodes et variables de toutes les classes et interfaces est donnée en Annexe B par la JavaDoc.

4.2.1 Le package common

Ce package contient principalement des interfaces. Il est utilisé par tous les autres packages. Les sources sont séparées afin d'éviter de redéfinir le même code à deux endroits différents ce qui pourrait entraîner des erreurs. Il se divise comme suit:

- *interfaces* **ErrorReport et StatusReport.** Utilisées pour répercuter les messages d'erreurs et les messages de statuts d'un objet à l'autre.
- *interface* **ComputingInterface.** Définit les différentes méthodes appelables du client vers le service.
- *interface* **ComputingRunner.** Définit les méthodes communes à tous les Runner
- *classe* **ConfigFileParser.** Utilise la package JAXP afin de pouvoir décoder des fichiers XML. Le but est de fournir au client et au serveur un système de chargement de configuration permettant de préciser via un fichier XML:
  - Une éventuelle série de lookups utilisables ;
  - Le fichier policy à utiliser par le gestionnaire de sécurité ;
  - Le fichier exec-policy à utiliser par le gestionnaire de sécurité ;
  - Le(s) URL où se trouve le code mobile associé au client ou au serveur.
- *classe* **ConfigStructure.** Permet de stocker toutes les informations extraites du fichier XML de configuration.
- *classe* **ComputingEntry.** Définit les attributs des services au niveau du lookup. Il s'agit d'une extension aux attributs JINI classiques.
- *classe* **ComputingException.** Exception qui peut être utilisée pour renvoyer des erreurs lors de l'exécution d'un calcul. Par exemple: la multiplication scalaire de deux vecteurs de taille différente.

4.2.2 Le package server

Ce package contient tout ce qui est nécessaire à la création d'un service JINI. Comme la première itération du projet était un système dédié, la recherche de modularité au niveau client était déjà importante. Il a alors suffit d'améliorer ce qui était nécessaire, et d'implémenter l'interface demandée par l'architecture générique.

Le serveur se compose de cinq classes et d'une interface.

- *interface* **ComputingInterfaceProxy.** Fait la jointure entre l'interface `ComputingInterface` et l'interface `Remote` servant à RMI. Cela permet de séparer la définition des méthodes du comportement en méthodes distantes.
- *classe* **ComputingServer.** Se charge

  - D'analyser le fichier de configuration ;
  - D'exécuter une série d'actions en rapport avec la configuration ;
  - De créer un serveur JINI à partir de la classe `ComputingImplementation`
- *classe* **JiniServer.** Est un serveur JINI générique. Elle crée à partir du nom d'une classe et d'une série de `LookupLocator` (URL pointant vers des lookup) un service JINI à part entière.
- *classe* **JiniServiceTemplate.** Utilisées par le JiniServer doivent hériter de la classe `JiniServiceTemplate`. Cette classe contient une série de méthodes et d'attributs permettant à toutes les classes qui en héritent de devenir un service JINI sans écrire une seule ligne de code ayant rapport avec JINI.
- *classe* **ServerConnectionManager.** Sert aux renouvellements des baux(lease) avec les lookup.
- *classe* **ComputingImplementation.** Est la classe principale du service. C'est elle qui implémente l'interface `ComputingInterface`.

4.2.3 Le package client

Le package client est divisé en deux grosses parties fortement interconnectées. D'un côté, il y a la gestion du calcul avec la gestion de la connection JINI/RMI. De l'autre, il y a une interface utilisateur en mode graphique.

La gestion du calcul est une classe abstraite car elle doit être redéfinie pour chaque type de calcul. La gestion de la connection JINI est quand à elle autonome.

L'interface utilisateur fait appel à un objet de gestion de calcul. C'est cet objet qui devra lancer le calcul et l'arrêter. Une série de méthodes standard entre gestionnaire de calcul et interface utilisateur permet à l'interface de donner une série d'informations sur l'état d'avancement du calcul, sur les ordinateurs utilisés pour le calcul, ...

Voici les différentes classes et interfaces pour la partie client:
- *interface* **ComputingClientListener.** Définit les méthodes distantes appelables du service vers le client.
- *classe* **ComputingManager.** Abstraite crée le lien entre le lookup JINI et la gestion du calcul. Pour créer un nouveau calcul, il faut hériter de cette classe et redéfinir les 5 méthodes relatives à la gestion pure du calcul.
- *classe* **ServiceIDComparator.** Est utilisée par la classe `ComputingLookupListener` afin de créer un ordre pour les identificateurs de services. Cela permet de stocker les services actifs dans une table style `TreeMap` qui est très efficace pour les opérations de recherche.
- *classe* **ClientGUI.** Crée l'interface graphique qui permet de suivre le déroulement du calcul sous différents angles. Cette classe lance le calcul à partir du nom de son gestionnaire.
- *classe* **ComputingLookupListener.** Est utilisée par JINI pour avertir le client d'une modification dans l'un des lookup (ajout, suppression , modification de service). Cette classe gère aussi la table des services utilisés.

4.2.4 Le package matrixmult

Ce package contient les classes et interfaces utilisées pour la multiplication de matrices. Elles constituent un exemple de ce qui peut être réalisé à partir de la plate-forme de base.

Les différentes classes et interfaces sont:

- *interface* **MatrixLoader.** Définit une série de méthodes permettant d'accéder aux valeurs d'une matrice. Le but étant de pouvoir utiliser plusieurs types de stockage pour une matrice tout en conservant une seule manière d'accéder à ses éléments.
- *interface* **MatrixMultInterface.** La méthode contenue dans l'interface `ComputingClientListener` n'étant pas suffisante pour implémenter une multiplication de matrice, il a été nécessaire d'ajouter d'autres méthodes appelables à distance. Par exemple: le chargement d'une ligne ou d'une colonne des matrices à multiplier.
- *classe* **MatrixMultRunner.** Définit les objets qui pourront s'exécuter sur les machines de calcul. Elle contient la multiplication scalaire d'un vecteur.
- *classe* **MatrixMultManager.** Est le centre du calcul de multiplication de matrice. C'est elle qui sépare les calculs et les envoie vers les différentes machines utilisables.
- *classe* **Point.** Permet de stocker un point par ses coordonnées (x,y).
- *classe* **UnitMatrix.** Implémente l'interface `MatrixLoader` et permet de charger une matrice unité. C'est l'élévation de cette matrice qui a servi de test pour tous les calculs.

4.2.5 Améliorations possibles de la plate-forme de tests

La plate-forme de tests telle que présentée ici comporte de lacunes. Elle demande à être finalisée afin de pouvoir en faire une plate-forme totalement réutilisable. Il subsite quelques problèmes de gestion d'erreurs. Ainsi, si un service n'arrive pas à renouveler son bail avec un lookup, il y a de fortes chances pour qu'il s'arrête tout simplement. Il suffit alors de le relancer ; ce qui n'est pas pratique pour l'utilisateur.

Une autre amélioration à apporter est la redéfinition des méthodes de certaines classes surtout au niveau client. On pourrait ainsi peut-être fusionner les classes `ComputingListener` et `ComputingManager`. Au niveau serveur, par contre, il y a, à priori, peu d'améliorations à apporter car il n'a subi que quelques petites modifications depuis les premiers tests.

Au niveau client, l'interface graphique peut être améliorée. Elle ne fait que créer, pour l'instant, un gestionnaire de multiplication de matrice. On pourrait arriver à demander à l'utilisateur d'entrer quelle est la classe à charger pour réaliser la gestion du calcul. On pourrait aussi ajouter quelques infos supplémentaires, en plus du nom de la machine, dans la liste des services travaillant pour le client.

Pour finir, il faudrait compléter la documentation du code.

4.3 Résultats expérimentaux

Les résultats donnés ici sont intéressants à analyser. Il vont permettre d'ouvrir la discussion quant à l'intérêt de JINI dans le calcul parallèle.

Une première série de tests a été faite sur une seule machine servant à la fois de client, de lookup, de serveur web et de service. Les tests ont été réitérés plusieurs fois dans les mêmes conditions logicielles et matérielles. La machine utilisée pour ce test est à base de Pentium II cadencé à 400Mhz avec 192 Mb de mémoire.

Une deuxième série de tests a été réalisée dans la salle Plato de l'ULB sur une douzaine de machines identiques comme services. Le client/lookup/serveur web était une autre machine plus puissante. Les machines de la salle Plato sont basées sur des Pentium II cadencés à 450 Mhz avec 128Mb de mémoire. La machine servant de client, de lookup et de serveur web est basée sur Pentium III cadencé à 800 Mhz avec 512Mb de mémoire.

Une troisième série de tests a été réalisée dans les mêmes conditions que la deuxième série de tests. Elle a été utile pour la recherche d'un calcul rentable.

4.3.1 Première série de tests

La première série de tests a consisté en trois séries de 10 fois le même calcul. Successivement, a été testés l'élévation au carré de matrices 1x1, 30x30, 100x100. Ces différents tests ont été comparés à l'élévation au carré de la même matrice sans utiliser le système JINI.

On peut résumer les résultats obtenus par le Tableau 4.1

Taille de la matrice	*Temps (ms) de calcul moyen avec JINI*	*Temps (ms) de calcul moyen sans JINI*
1	1 471	1
30	11 710	13
100	67 900	78

TAB. 4.1 – Résumé des résultats expérimentaux

On peut tout de suite remarquer la nette différence de vitesse entre le calcul avec JINI et le calcul sans JINI.

En si on prend le calcul de la matrice 1x1, on peut négliger le temps de calcul réel et déterminer un temps de transmission avoisinant les 1,5 secondes.

Il faut savoir que ce temps comprend: l'initialisation de la communication JINI et l'envoi du Runner. Ensuite, le système tourne sur lui même grâce à RMI. En comparant les chiffres pour les matrices 30x30 et 100x100, on remarque une accélération dans le cas de la matrice 100x100. Cette accélération peut avoir plusieurs origines:

- Le cache RMI qui permet de ne pas transférer deux fois la même classe mais de réutiliser celle déjà chargée ;
- Le compilateur JIT (Just In Time) qui garde le code compilé en mémoire pour la machine et permet une nette accélération dans les programmes courants ;
- Des gains d'échelle par rapport aux temps d'initialisation des communications.

Si on compare les temps de calcul avec et sans JINI pour une matrice de même taille, on tombe plus ou moins sur un même facteur. Ce qui tend à prouver l'importance du compilateur JIT.

Cette série de tests permet d'avoir une idée des ordres de grandeur des temps de calcul. Il aideront par la suite à comprendre les conclusions.

4.3.2 Deuxième série de tests

La deuxième série de tests a consisté en un lancement du calcul sur une douzaine de machines en parallèle. Le code n'ayant pas changé entre les deux tests.

Les tests ont été effectués avec des tailles de matrices différentes.

Il n'est pas nécessaire de produire ici de chiffres car les temps de calcul furent très décevants. Plutôt que de raccourcir le temps de calcul, le système en parallèle l'allongeait.

Ce résultat semblait bizarre. Une recherche possible d'explication fut la suivante: le temps d'exécution d'un calcul étant de l'ordre de la milliseconde, ce temps devenait négligeable par rapport au temps de communication entre les services et le client. Les services ayant rapidement finis leurs calculs, attendaient, pour des raisons de synchronisation, constamment une réponse du client. Il y avait alors une congestion au niveau de la communication réseau qui ralentissait tout le système.

Cette conclusion fut un peu hâtive car une troisième série de tests allait apporter la réponse au problème: il existait bien un problème au niveau de la communication réseau, mais ce problème n'était pas d'ordre logiciel. En effet, la cause du ralentissement était une carte réseau de fonctionnement douteux. La troisième série de tests ayant été exécutée dans les même conditions que la deuxième, excepté le changement de cette carte réseau, et ayant donné de biens meilleurs résultats, il semble que la cause du problème était bien matérielle.

La leçon à en tirer est qu'il faut s'assurer, pour la connection du client, d'avoir du matériel parfaitement fonctionnel.

4.3.3 Troisième série de tests

La troisième série de tests a consisté en la simulation d'un algorithme ayant un temps d'exécution supérieur à celui de la multiplication de matrice.

L'idée est très simple: tout en utilisant le code de la multiplication de matrice, on ajoute lors de la multiplication de deux vecteurs un temps d'attente simulant un calcul plus compliqué. La conservation de la multiplication de matrice permet de faire intervenir un certain taux de communication entre le service et le client au cours du calcul.

Les tests ont été effectués à deux reprises. La première sur une machine isolée servant à la fois de serveur, de lookup et de client. La seconde fois, cette machine a été utilisée comme lookup et client ; et 10 machines de la salle Plato étaient utilisées comme services.

Le test a donc consisté à lancer plusieurs fois, avec des temps de latence différents, l'élévation au carré d'une matrice 30x30.

Les résultats sont présentés dans le Tableau 4.2, page 38.

On peut immédiatement remarquer la rentabilité de l'utilisation de JINI lorsque le calcul dure un certain temps. Utiliser l'architecture proposée est déjà rentable sur 10 machines pour des calculs de durée dépassant les 100ms. Cette durée correspond sur une machine à base de Pentium II cadencé à 400Mhz à l'élévation au carré(en Java) d'une matrice 110x110. Il n'est donc pas difficile de trouver des algorithmes

Temps de latence introduits dans le calcul (ms)	Vitesse théorique atteignable sur une machine isolée (ms)	Vitesse atteinte sur la machine isolée (ms)	Vitesse théorique atteignable en utilisant 10 ordinateurs (ms)	Vitesse atteinte dans la salle Plato (ms)
0	0 [1]	12812	0 [1]	4899
100	90000	104149	9000	51334
500	450000	468198	45000	56827

[1] Sans compter le temps de calcul de la multiplication de deux vecteurs. A titre d'information, l'élévation au carré d'une matrice 30x30 en Java prend 12 ms sur une machine à base de Pentium II cadencé à 400Mhz.

TAB. 4.2 – Résultats obtenus en introduisant un temps de latence

à paralléliser plus complexes que la multiplication de matrice pour lesquels le temps d'exécution d'une unité de calcul est supérieur à 100 ms sur un machine courante.

Des examples d'algorithmes peuvent se trouver dans des projets existants tesl le raytracer POV-Ray [2] qui existe en version parallélisée. Ou le projet de recherche en intelligeance extraterrestre "Seti@home" [4].

On peut remarquer que lorsque le temps d'exécution d'une unité de calcul augmente, les temps de connection deviennent négligeables. Dès lors, l'utilisation de dix machines multiplie presque la vitesse de calcul par dix.

Chapitre 5

Apports de JINI au calcul parallèle

5.1 Introduction

La faisabilité d'un calcul parallèle, basé sur JINI, n'est plus à démontrer. Le fait d'avoir pu réaliser une implémentation et des tests en sont la preuve. Il reste à savoir si JINI peut apporter quelque chose en plus par rapport aux systèmes existants basés, par exemple, sur le protocole RPC.

En plus des apports de l'architecture JINI en elle-même, il faut créer une architecture de communication qui permettra le calcul. La manière de la concevoir peut aussi apporter des idées nouvelles pour le calcul parallèle. D'autre facteurs tournant autour de JINI, tel le langage Java, apportent également leur lot de nouveautés.

Il faut aussi éveluer les inconvénients du système. L'analyse de tous les inconvénients permettra mieux cibler les domaines aux-quels l'utilisation de JINI et Java dans le calcul parallèle peuvent être utils.

5.2 Utilité de la programmation objet

La conception objet permet de simplifier la programmation d'un système de calcul parallèle au niveau de la conception. En effet, elle permet de réaliser au niveau implémentation une séparation nette entre gestion de la communication réseau et gestion du calcul.

Le principal apport de la programmation objet est l'héritage et la surcharge de méthodes. L'héritage évite de devoir reprogrammer plusieurs fois la même chose ou de faire de la copie de code. Cela permet donc de n'écrire qu'une seule fois le code et donc de ne devoir le tester qu'une seule fois.

La surcharge de méthodes permet de rapidement définir un nouveau comportement pour une classe d'objet. Dans le cas du calcul parallèle, on peut réutiliser une classe de calcul qui fonctionne pour en créer une nouvelle ayant les même bases mais qui implémente un autre algorithme.

Un des désavantages de la programmation orientée objet est le temps d'exécution. Il est reconnu que la vitesse d'exécution d'un programme écrit dans un langage orienté objet est inférieure à la vitesse

d'exécution du même programme écrit dans un langage procédural classique. Cette perte de temps n'est visible que lors de la communication entre objets.

La programmation objet apporte beaucoup au programmeur aussi bien en réduisant le temps de développement qu'en réduisant le temps de tests et déboggage.

5.3 Utilité du langage Java et de la JVM

Le langage Java peut apporter beaucoup au calcul parallèle, surtout si le calcul est réalisé sur un parc de machines hétérogène.

Le langage Java est avant tout un langage orienté objet. Il permet donc l'héritage, et la surcharge de méthodes. Il a fait l'objet, lors de sa conception, d'une attention toute particulière pour ne pas répéter des erreurs commises dans d'autres langages orientés objet. Java limite l'héritage à un seul objet, mais à plusieurs interfaces. Ceci évite les héritages avec deux méthodes de même nom provenant de deux parents différents.

Le langage Java a supprimé le mécanisme de pointeurs. Il a été remplacé par des pointeurs implicites, gérés par le compilateur et la JVM. On a du même coup supprimé toute l'arithmétique sur pointeur qui donnait lieu plus souvent à des erreurs de programmation qu'à des avantages.

La gestion des objets en Java est différente de celle en C++. Java utilise un Garbage Collector qui se charge de la gestion des objets à supprimer. Le programmeur ne doit donc plus se soucier de la suppression des objets qu'il a créé ; le système s'en charge pour lui.

Java apporte encore d'autres avantages tels

- une gestion des erreurs par mécanismes d'exceptions ;
- une bibliothèque de classes et de méthodes touchant un grand nombre de domaines ;
- un mécanisme de sécurité interne permettant à l'utilisateur de restreindre les accès aux ressources du système. Dans le cas du calcul parallèle envisagé ici, il serait prudent de restreindre l'accès aux fichiers systèmes afin d'éviter qu'un programmeur mal intentionné ne crée un `Runner` allant modifier des fichiers sensibles ;
- un système de signature permettant d'identifier les objets et leur source. La signature de l'objet peut alors être utilisée par le système de sécurité pour éventuellement permettre l'accès à certaines ressources système.

Tous ces apports du langage sont intimement mêlés avec le concept de JVM (Java Virtual Machine) qui est en réalité l'implémentation de tous les avantages apportés par le langage Java. L'utilisation de la JVM entraîne l'excécution d'un langage semi-compilé. La JVM fait l'intermédiaire entre le bytecode (créé par le compilateur java) et la machine sur la quelle tourne en réalité le programme.

Le principal avantage de la JVM est de pouvoir exécuter du bytecode sur une machine d'architecture quelconque. Il s'agit là du concept de :"Compile once, run everywhere". Une seule compilation suffit ; la différence d'architecture entre les ordinateurs est prise en charge par la JVM.

Cette approche présente un problème majeur: la vitesse d'exécution. Tous le mécanismes de sécurité, l'interprétation du bytecode, . . . entraînent un ralentissement de l'exécution du programme. Ceci constitue la principale faiblesse de Java.

Il existe cependant des moyens de contourner ce problème:

- **L'utilisation d'un compilateur JIT.** Ce compilateur qui collabore avec la JVM se charge de traduire au moment de l'exécution le bytecode Java en code natif. Le code natif est alors mis en mémoire et peut être réutilisé. Grâce à ce mécanisme, on peut remarquer, dans le cas d'un algorithme itératif, une nette amélioration des performances. Un désavantage de cette méthode est le besoin de mémoire. Ce désavantage devient de moins en moins visible à mesure de l'évolution actuelle de l'informatique. La puissance des machine actuelles diminue aussi le temps mis par le JIT à traduire le code.
- **L'utilisation de JNI (Java Native Interface).** Permet à un programmeur de créer une bibliothèque de fonctions compilées pour la machine sur laquelle elles s'exécuteront et interfacées avec le langage Java. Dans le cas du calcul parallèle, une bibliothèque de calcul pourrait être mise en place. La vitesse d'exécution de certaines fonctions mathématiques classiques serait alors foretement accélérée. Cela demande un effort de programmation mais qui sera bénéfique en temps d'exécution.

L'utilisation d'un langage ou d'un autre est principalement un choix de programmeur. Le langage Java a l'avantage d'être bien conçu et d'évoluer grâce au suivi :

- des nouvelles technologies par la création de nouvelles bibliothèques ;
- des problèmes du langage par l'ajout de nouvelles fonctionnalités tout en gardant une syntaxe correcte.

5.4 Utilité de la couche JINI/RMI

La couche JINI/RMI permet une solution intéressante au niveau réseau pour le programmeur.

JINI apporte la base de la gestion d'ajout et de suppresion de services. RMI apporte la facilité de programmation des appels de méthodes à distance. RMI apporte aussi le mécanisme de code mobile.

Ces apports permettent comme il a déjà été exposé de créer une architecture de calcul parallèle originale.

Comme le couche JINI/RMI est basée sur le langage Java et la JVM, elle profite de tous les avantages offerts par ceux-ci ; mais elle hérite aussi de tous les inconvénients. Principalement au niveau de la vitesse.

Chapitre 6

Conclusion

L'application de JINI au calcul parallèle ne s'est pas avéré être un mauvais choix. Le calcul parallèle tel qu'il est souvent vu aujourd'hui est souvent limité à un système client-serveur fixe. Une salle est installée en cluster, elle est configurée de manière optimale pour un calcul bien précis, et ce calcul est effectué.

Des projets visant à utiliser des machines pouvant apparaitre ou disparaitre sur le réseau commencent à émerger. L'un des plus connus est sûrement le projet "Seti@Home", qui est un projet de recherche d'intelligence extraterrestre. Il utilise les ordinateurs disponibles sur Internet afin de réaliser des détections de signaux intelligents dans les ondes radios.

Chaque projet de ce type utilise un autre système de gestion du réseau d'ordinateurs. Il demande, de plus, d'installer sur chaque machine un logiciel qui leur est propre.

L'approche que j'ai imaginée ici se veut différente en ce sens que l'utilisateur ne doit installer qu'un seul logiciel universel qui pourra servir à plusieurs projets de calcul. Il n'aura plus à se soucier de suivre l'évolution du logiciel du projet auquel il désire participer. Si une partie de l'algorithme de calcul est modifiée, il suffit, au programmeur de cet algorithme, de placer la nouvelle version sur son serveur web. Tous les calculs qu'il lancera par la suite utiliseront automatiquement la nouvelle version ; sans réinstallation préalable sur les ordinateurs distants.

Le désavantage de cette approche est qu'elle repose sur la technologie Java qui utilise un système de machine virtuelle. Cette machine virtuelle exécute un code semi-compilé appelé bytecode. Pour pouvoir exécuter le programme sur un ordinateur, une traduction du bytecode est nécessaire. L'exécution du programme est donc ralentie.

Il existe heureusement comme nous l'avons vu précédement des solutions qui tentent de diminuer cet effet. De plus, la puissance des ordinateurs ne faisant que s'accroitre, le temps nécessaire à la traduction diminue de plus en plus. Nous arriverons un jour à exécuter un programme Java sans que la perte de vitesse ne soit sensible.

Un avantage de l'approche telle qu'imaginée dans ce travail est la recherche de séparation entre gestion du réseau et gestion du calcul. Une fois que la gestion du réseau est au point, elle ne doit plus être modifiée. Le programmeur peut alors se concentrer sur la réalisation d'algorithmes. Le temps de développement d'un algorithme en parallèle est donc diminué.

Le cas où cette approche sera le plus rentable est l'utilisation du calcul parallèle pour des calculs très complexes mais dont le nombre d'utilisations de l'algorithme restera faible. Dans ce cas, le temps de développement est important.

JINI peut apporter beaucoup au calcul parallèle mais aussi à d'autees domaines d'application. JINI est une technologie jeune qui doit encore mûrir et se faire une place dans le monde de l'informatique mais qui a un bel avenir devant elle.

Annexes

Annexe A

Programmer avec JINI

A.1 Introduction

Le but de ce cette annexe est d'expliquer par un exemple simple la programmation d'un service JINI. Ne seront abordés ici que les concepts de base. Seules seront utilisées les classes utilitaires accompagnant JINI et permettant la programmation aisée d'un service et d'un client. Pour plus de renseignements, quand à une utilisation plus avancée de JINI, voir les documents [5] [8] [7].

A.2 Exemple d'implémentation

L'exemple présenté ici sera le classique "Hello World" qui sera adapté à JINI. Ce programme n'apporte rien si ce n'est son caractère didactique. Il permet d'illustrer l'implémentation.

Le diagramme des classes à implémenter est donné à la Figure A.1, page 46.

Le programme est donc divisé en deux interfaces et deux classes:

– *interface* **HelloWorldInterface.** Interface qui est offerte par le service et recherchée par le client.
– *interface* **HelloWorldProxy.** Hérite de la précédente. Elle lui ajoute l'interface `Remote` qui permettra au client de faire un appel RMI vers le service.
– *classe* **HelloWorldService.** Implémente les deux premières. Outre l'implémentation, cette classe va s'enregistrer auprès du lookup JINI.
– *classe* **HelloWorldClient.** Tente de trouver un service implémentant l'interface `HelloWorldInterface` et affiche la chaîne de caractères retournée.

A.3 Programmation de la partie commune

La partie qui est commune aux client et au service est l'interface `HelloWorldInterface`. Voici son code:

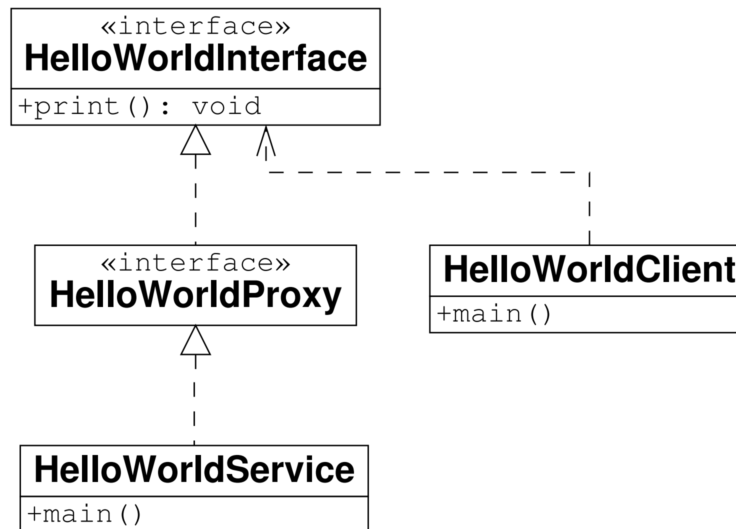

FIG. A.1 – Diagramme de classes pour le programme Hello World

```
import java.rmi.RemoteException;

public interface HelloWorldInterface {
  public String getText () throws RemoteException;
}
```

Comme il s'agit d'une méthode appelée à distance, l'exception `RemoteException` est obligatoire.

A.4 Programmation du service

La programmation du service tient en deux parties: l'interface proxy et l'implémentation du service.

Voici l'interface du proxy:

```
import java.rmi.Remote;

public interface HelloWorldProxy extends Remote, HelloWorldInterface {
}
```

On ne fait donc qu'ajouter l'héritage de l'interface `Remote`. Pour l'implémentation du service en lui-même, le code est plus complexe. Tout d'abord, une série de packages sont utiles:

```
import java.rmi.*;
import java.rmi.server.*;

import net.jini.core.lookup.*;
```

```
import net.jini.core.entry.*;
import net.jini.discovery.*;
import net.jini.lookup.*;
import net.jini.lookup.entry.*;
```

Cela permettra d'utiliser sans problème les objets RMI et JINI.

Il faut, ensuite, un prototype pour notre classe. Elle doit implémenter différentes interfaces:
- HelloWorldProxy ;
- ServiceIDListener.

L'interface `ServiceIDListener` est demandée par la classe `JoinManager` qui sera utilisée plus tard. Comme on veut faire de notre service un serveur RMI, on doit étendre la classe `UnicastRemoteObject`.

La définition de la classe sera donc:

```
public class HelloWorldService extends UnicastRemoteObject
        implements HelloWorldProxy, ServiceIDListener {
}
```

Il reste à compléter les méthodes présentes dans la classe. Tout d'abord l'implémentation de `getText()`:

```
public String getText(){
    return "Hello World";
}
```

Ensuite, il faut définir la méthode `serviceIDNotify` demandée par l'interface `ServiceIDListener` qui permet de récupérer le numéro d'identification qui est donné au service par le lookup:

```
public void serviceIDNotify(ServiceID id){
    System.out.println("Server ID="+id);
}
```

Pour finir, il reste l'enregistrement auprès du lookup. Tout ce qui suit est inclu dans la fonction main de la classe.

La première étape est de choisir un SecurityManager pour le service. Cette sécurité est demandée par RMI et on est donc obligé de l'insérer dans le code.

```
System.setSecurityManager(new RMISecurityManager());
```

Le deuxième étape est de créer un objet à exporter:

```
HelloWorldService hws = new HelloWorldService();
```

La troisième étape est de définir un ensemble d'attributs:

```
Entry [] attrs = {
    new Name("Hello World"),
    new Comment("Service in developpement use only for tests.")
};
```

Il existe un ensemble d'attributs prédéfinis ; s'ils ne suffisent pas, il est aisé de créer soi-même ses propres attributs.

La quatrième étape est de rechercher un lookup:

```
LookupDiscoveryManager mgr= new LookupDiscoveryManager(
    LookupDiscovery.ALL_GROUPS, null, null);
```

Les différents paramètres du constructeur de cette classe sont:

- Un nom de groupe auquel appartiendra le service. On le met par simplicité dans le groupe général. La notion de groupe permet de regrouper des services de même nature.
- Un ensemble de "Locator" qui permettent de spécifier l'URL d'un lookup si on la connaît. De toutes façons, le LookupDiscoveryManager fera une recherche des lookups existants par multicast.
- Un Listener qui permet au service d'être averti lors de l'arrivée de nouveaux lookup. Cette fonctionnalité ne sera pas utilisée ici.

La cinquième étape consiste à attendre qu'au moins un lookup ait été découvert:

```
while(mgr.getRegistrars().length<1)
    try{Thread.sleep(100);} catch(Exception e){};
```

La dernière étape est de s'enregistrer auprès des lookup découverts:

```
JoinManager manager = new JoinManager(hws, attrs, hws, mgr, null);
```

Les cinq paramètres du constructeur de cette classe sont:

- Un objet à exporter.
- Un ensemble d'attributs pour cet objet.
- Un SeviceIDListener, qui est dans ce cas la même classe que le service.
- Un LookupDiscoveryManager
- Un LeaseRenewalManager qui sera chargé de gérer les baux(lease). Si le paramètre passé est "null", la classe en crée un par défaut.

Ensuite, on peut attendre l'appui d'une touche par l'utilisateur afin de terminer proprement le service.

Au final, le code Java de ce service est:

```
import java.rmi.*;
import java.rmi.server.*;

import net.jini.core.lookup.*;
import net.jini.core.entry.*;
import net.jini.discovery.*;
import net.jini.lookup.*;
import net.jini.lookup.entry.*;
```

```
public class HelloWorldService extends UnicastRemoteObject
        implements HelloWorldProxy, ServiceIDListener {

    public HelloWorldService(){
    }

    public String getText(){
        return "Hello World";
    }

    public void serviceIDNotify(ServiceID id){
        System.out.println("Server ID="+id);
    }

    public static void main(String[] args){
        System.setSecurityManager(new RMISecurityManager());
        HelloWorldService hws = new HelloWorldService();
        Entry[] attrs = {
            new Name("Hello World"),
            new Comment("Service in developpement use only for tests.")
        };
        LookupDiscoveryManager mgr= new LookupDiscoveryManager(
            LookupDiscovery.ALL_GROUPS, null, null);
        while(mgr.getRegistrars().length<1)
            try{Thread.sleep(100);} catch(Exception e){};
        JoinManager manager = new JoinManager(hws, attrs, hws, mgr, null);
        byte[] b={ 0 };
        System.out.println("Press enter to end.");
        System.in.skip(System.in.available());
        System.in.read(b);
        manager.terminate();
    }
}
```

A.5 Programmation du client

La programmation du client se résume à la recherche d'un lookup, ensuite, la recherche d'un objet réalisant l'interface souhaitée afin d'utiliser cet objet.

Pour commencer la programmation du client, quelques importations sont nécessaires:

```
import java.rmi.*;
import net.jini.lookup.*;
```

```
import net.jini.core.lookup.*;
import net.jini.discovery.*;
```

Le prototype de la classe est simple:

```
public class HelloWorldClient{
}
```

On peut directement entrer dans la fonction main. Comme pour le service, on va changer le gestionnaire de sécurité du système:

```
System.setSecurityManager(new RMISecurityManager());
```

Comme pour le service, il faut rechercher un lookup et attendre d'en avoir trouvé au moins un:

```
LookupDiscoveryManager mgr= new LookupDiscoveryManager(
    LookupDiscovery.ALL_GROUPS, null, null);
while(mgr.getRegistrars().length<1)
    try{Thread.sleep(100);} catch(Exception e){};
```

On va alors utiliser un `ServiceDiscoveryManager` qui permet la recherche de services:

```
ServiceDiscoveryManager sdm=
    new ServiceDiscoveryManager( ldm, new LeaseRenewalManager());
```

Le constructeur de cette classe requiert deux paramètres:
- Un LookupDiscoveryManager
- Un LeaseRenewalManager

Il faut alors créer un `ServiceTemplate` qui servira de filtre lors de la recherche de services. On se limitera ici à la recherche sur une interface.

```
Class[] name = new Class[] { ComputingInterface.class };
ServiceTemplate st=new ServiceTemplate(null, name, null);
```

La création d'un template demande trois paramètres:
- Un numéro d'identification,
- Un ensemble d'interfaces que doit implémenter le service,
- Un ensemble d'attributs que doit avoir le service.

Seules les valeurs ne valant pas "null" sont utilisées.

On peut alors entreprendre la recherche d'un objet qui pourra être utilisé.

```
ServiceItem si=sdm.lookup(st, null);
if(si!=null){
    try{
        System.out.println(" resultat ="+
            ((HelloWorldInterface)si.service).getText());
    }catch(Exception e) {
    }
}
```

La méthode lookup du `ServiceDiscoveryManager` demande deux paramètres: un template et un éventuel filtre qui permet une sélection plus fine du service à choisir. Dans l'appel qui est fait ici, seul le premier service qui répond au template est renvoyé. Il existe d'autres méthodes pour retrouver tous les services possibles.

On est obligé d'exécuter le service dans un "try and catch" car on risque une erreur en cas de déconnection réseau.

Voici le code du client tout en un:

```
import java.rmi.*;
import net.jini.lookup.*;
import net.jini.core.lookup.*;
import net.jini.discovery.*;

public class HelloWorldClient{
    public static void main(String [] args){
        System.setSecurityManager(new RMISecurityManager());
        LookupDiscoveryManager mgr= new LookupDiscoveryManager(
            LookupDiscovery.ALL_GROUPS, null, null);
        while(mgr.getRegistrars().length<1)
            try{Thread.sleep(100);} catch(Exception e){};
        ServiceDiscoveryManager sdm=
            new ServiceDiscoveryManager( ldm, new LeaseRenewalManager());
        Class [] name = new Class [] { ComputingInterface.class };
        ServiceTemplate st=new ServiceTemplate(null, name, null);
        ServiceItem si=sdm.lookup(st, null);
        if(si!=null){
        try{
            System.out.println(" resultat ="+
                ((HelloWorldInterface) si.service).getText());
        }catch(Exception e) {
        }
    }
}
```

Pour exécuter ces exemples, il faut bien entendu commencer par lancer un lookup JINI. Ensuite, on peut démarrer en premier soit le client, soit le service. Dans les deux cas, le résultat sera identique.

Annexe B

Documentation Java (JavaDoc)

B.1 package edu.oleastre.computing.common

B.1.1 Interfaces

public interface ErrorReport

Simple code to handle exceptions.

This may be use to transfer the message from a class to another.

It may be used to display a message.

public errorReport(String s)

public interface ComputingInterface

Interface between Computing Client and Server.
public importRunner(ComputingRunner cr)

public executeRunner()

public resetRunner()

public releaseRunner()

public interface ComputingRunner

Object used to transfer calc code and data.
public init()

public execute()

public reset()

public stop()

public interface StatusReport

Simple code to transfer messages between classes

It may be used to display a message.

public statusReport(String s)

B.1.2 Classes

public class ConfigFileParser

Extends:
- Object

public ConfigFileParser( )

public static parse(String fileName)

public class ConfigStructure

Extends:

- Object

public ConfigStructure( )

public addLookupURL(String hostName, Integer port)

public addCodebaseURL(String hostName, String port, String path)

public getLookupLocators()

public getCodebase()

public getPolicy()

public setPolicy(String s)

public getExecPolicy()

public setExecPolicy(String s)

public class ComputingEntry

Computing Service status property.

Extends:

- AbstractEntry

Implements:

- ServiceControlled

public ComputingEntry( )

public ComputingEntry(Integer s, String hostname)

public toString()

B.1.3 Exceptions

public class class ComputingException

Computing library exceptions. When new funcitions are added to this lib, there may be have some errors to be added.

Extends:

- Exception

public ComputingException( )

public ComputingException(int err)

public getMessage()

B.2 package edu.oleastre.computing.client

B.2.1 Interfaces

public interface ComputingClientListener

Implements:

- Remote

public returnObject(ServiceID id, Object obj)

B.2.2 Classes

abstract public class ComputingManager

When Services matching ComputingInterface are added, changed, removed, ComputingLookupListener calls the **useService** or **notusedService** , **removeService** , **releaseService** ,**executeService** functions with the ServiceItem as argument. Service selection is done into the select function, useService is called is select return true notUsedService if called otherwise. Thoses five functions have to be redefined if you

want to handle your own calc.

Extends:

- UnicastRemoteObject

Implements:

- StatusReport
- ErrorReport
- ComputingClientListener

public ComputingManager(StatusReport statusReporter, ErrorReport errorReporter, Integer port, InetAddress addr, ConfigStructure config)

public statusReport(String s)

public errorReport(String s)

public getHostsList()

public getComputingLookupListener()

Return the current ComputingLookupListener.

public start()

Start the ComputingLookupListener.

public stop()

Stop the ComputingLookupListener.

public getRMIRegistry()

Return the rmiRegistry created to dialog with server.

public getAddress()

Return callback internet address (=client host address).

public getPort()

Return callback rmi access port.

public getConfig()

Return parent's config Structure.

abstract public useService(ServiceItem si)

This function is called when a service is in READY state and is selected. After calling this function, the service have to come into the BUSY state. The returned object is associated into a serversList (in the ComputingLookupListener) with the seviceID. You have to rewrite this function to meet your needs.

abstract public removeService(ServiceItem si)

This function is called when a service is no more availabale. (shutdown, network problem, ...) You have to rewrite this function to meet your needs.

abstract public select(ServiceItem si)

This function is called to select services. It return true if the service have to be used by the client. It return false otherwise. You have to rewrite this function to meet your needs.

abstract public getEvolution()

This function return a double representing the current calc status. This number n have to be in the interval: 0<=n<=1 You have to rewrite this function to meet your needs.

abstract public isDone()

This function have to return true if the calc is terminated. You have to rewrite this function to meet your needs.

public class ServiceIDComparator

This class is used into ComputingManager to be able to class ServiceIds. With this class, Services may be stored into a treemap.

Extends:

- Object

Implements:

- Comparator

public ServiceIDComparator( )

public compare(Object obj1, Object obj2)

public equals(Object o)

public class ClientGUI

This class provides a nice launcher for the computing client.

Extends:

- JFrame

Implements:

- ActionListener
- StatusReport
- ErrorReport

public ClientGUI(ConfigStructure config)

public errorReport(String s)

public statusReport(String s)

public setStatus(String s)

public actionPerformed(ActionEvent e)

public static main(String[] args)

public class ComputingLookupListener

This is the base class for ComputingManagers.

The main functions are:
– Create a LookupDiscoveryManager to find lookups.
– Wait for a lookup.
– Create a ServiceDiscoveryManager to find services.
– If the stop method is called, it terminates all connections.

Extends:
- Object

Implements:
- Runnable
- ServiceDiscoveryListener

public ComputingLookupListener(ComputingManager parent)

public start()

public stop()

public run()

public serviceAdded(ServiceDiscoveryEvent event)

public serviceRemoved(ServiceDiscoveryEvent event)

public serviceChanged(ServiceDiscoveryEvent event)

public getLookupCache()

public getServersList()

public setListValue(ServiceID key, Object obj)

Associate a serviceID key with obj into the serversList.
public getListValue(ServiceID key)

Return the object associated with a serviceID into the serversList.
public isInList(ServiceID key)

Return true if serversList contains key.
public getService(ServiceID id)

Return the ServiceItem corresponding to ID.
public getHostsList()

Return a string vector containing used hostnames.

B.3 package edu.oleastre.computing.matrixmult

B.3.1 Interfaces

public interface MatrixLoader

Interface to load matrix. Simply implement this to be ready to load a matrix into MatrixMultManager.
public getLine(int i)

public getColumn(int i)

public getWidth()

public getHeight()

public interface MatrixMultInterface

Remote Interface to MatrixMultManager used by MatrixMultRunner.
Implements:
- Remote

public getPoint(ServiceID id)

public getRow(int i)

public getColumn(int i)

B.3.2 Classes

public class MatrixMultRunner

Implementation of a ComputingRunner to multiply matrix.
Extends:
- Object

Implements:
- ComputingRunner
- Serializable

public MatrixMultRunner(String rmiCallBack, ServiceID id)

public init()

public reset()

public stop()

public execute()

public class Point

Simple point class. Used to store points reference into MatrixMultManager.
Extends:
- Object

Implements:
- Serializable

public Point(int x, int y)

public Point( )

public getX()

public getY()

public setX(int x)

public setY(int y)

public setValues(int x, int y)

public toString()

public class UnitMatrix

Implementation of MatrixLoader. The returned matrix is a unit matrix. (0 for all elements except for the diagonal where elements are 1).
Extends:
- Object

Implements:
- MatrixLoader

public UnitMatrix(int size)

public getLine(int x)

public getColumn(int x)

public getWidth()

public getHeight()

public class MatrixMultManager

This is a particular ComputingManager designed to multiply matrix.

Extends:

- ComputingManager

Implements:

- MatrixMultInterface
- ComputingClientListener

public MatrixMultManager(StatusReport statusReporter, ErrorReport errorReporter, Integer port, InetAddress addr, ConfigStructure config)

public useService(ServiceItem si)

public removeService(ServiceItem si)

public select(ServiceItem si)

public getEvolution()

public isDone()

public returnObject(ServiceID id, Object obj)

public getPoint(ServiceID id)

public getRow(int i)

public getColumn(int i)

B.4 package edu.oleastre.computing.server

B.4.1 Interfaces

public interface ComputingInterfaceProxy

Computing Service proxy.

Simple link between ComputingInterface and RMI.
Implements:
- ComputingInterface
- Remote

B.4.2 Classes

public class ComputingServer

Computing Server class.

Load Server config and call a JiniServer to create the server.
Extends:
- Object

public ComputingServer( )

public static main(String[] args)

public class JiniServer

Jini Server class.

Generic class taking a class name as argument. This class must be an child of JiniServiceTemplate. The server use this class and propagate it as a jini service.
Extends:
- Object

Implements:
- ErrorReport

public JiniServer(String s, LookupLocator[] locators)

public errorReport(String s)

public static main(String[] args)

abstract public class JiniServiceTemplate

This class is a base for any JiniService. Extend it and use the JiniServer to create a fully functionnal jini service.

Extends:

- UnicastRemoteObject

public getJiniAttributes()

public setJiniAttributes(Entry[] attrs)

public getJiniLeaseRenewDelay()

public setJiniLeaseRenewDelay(long del)

public setServerConnectionManager(ServerConnectionManager scm)

public getServerConnectionManager()

public class ServerConnectionManager

This is an other Jini JoinManager. The difference between this and the sun implementation is that with this one you're able to set the LeaseRenewal time.

Extends:

- Object

Implements:

- DiscoveryListener

public discovered(DiscoveryEvent ev)

public discarded(DiscoveryEvent ev)

public terminate()

public modifyAttributes(Entry[] template, Entry[] attrs)

public class ComputingImplementation

Computing Service implementation.

This class is THE computing service. It manage the calc.
Extends:
- JiniServiceTemplate

Implements:
- ComputingInterfaceProxy
- Runnable

public ComputingImplementation( )

public importRunner(ComputingRunner cr)

public executeRunner()

public resetRunner()

public releaseRunner()

public start()

public stop()

public run()

Table des matières

1 Introduction **2**

2 Architecture Jini **4**

2.1 Introduction 4

2.2 Concepts de base 4

2.2.1 Services 5

2.2.2 Lookup 5

2.2.3 Leasing (ou Bail) 6

2.2.4 Evénements 6

2.2.5 Quelques conventions 6

2.3 Fonctionnement d'un système JINI 7

2.3.1 L'enregistrement d'un service auprès du lookup 8

2.3.2 La recherche d'un service par le client 8

2.3.3 La connection du client et d'un service 9

2.4 Le protocole RMI 10

2.4.1 Les objets sérialisés 10

2.4.2 Le code mobile 11

2.4.3 Les interfaces et les serveurs RMI 11

2.5 Programmer avec JINI 12

2.6 Comparaison avec CORBA 12

2.6.1 Langage de définition des interfaces 13

2.6.2 Chargement dynamique d'interfaces 13

2.6.3 Service de nommage 13

2.6.4 Passage des paramètres 14

2.6.5 Recherche de services 14
2.6.6 Conclusion 15

3 Architecture pour le calcul parallèle 16
3.1 Introduction 16
3.2 Architecture dédiée 17
3.2.1 Interface du service 17
3.2.2 Le client 18
3.2.3 Déroulement d'un calcul 18
3.3 Architecture généraliste, avec modifications de statut 21
3.3.1 Interface du service 21
3.3.2 Interface du Runner 22
3.3.3 Le client 22
3.3.4 Déroulement d'un calcul 24
3.4 Architecture généraliste, sans modifications de statut 26
3.4.1 Interfaces du service et du Runner 26
3.4.2 Le client 27
3.4.3 Déroulement d'un calcul 27
3.5 Autres améliorations possibles 29
3.5.1 Modification d'architecture 29
3.5.2 Compression 29
3.5.3 Code natif 29
3.5.4 Sélection 30
3.5.5 Et d'autres... 30

4 Plate-forme de test et résultats expérimentaux 32
4.1 Introduction . . . 32
4.2 Plate-forme de test . . . 32
4.2.1 Le package common . . . 33
4.2.2 Le package server . . . 33
4.2.3 Le package client . . . 34
4.2.4 Le package matrixmult . . . 35
4.2.5 Améliorations possibles de la plate-forme de tests . . . 35
4.3 Résultats expérimentaux . . . 35
4.3.1 Première série de tests . . . 36
4.3.2 Deuxième série de tests . . . 37
4.3.3 Troisième série de tests . . . 37

5 Apports de JINI au calcul parallèle 39
5.1 Introduction . . . 39
5.2 Utilité de la programmation objet . . . 39
5.3 Utilité du langage Java et de la JVM . . . 40
5.4 Utilité de la couche JINI/RMI . . . 41

6 Conclusion 42

A Programmer avec JINI 45
A.1 Introduction . . . 45
A.2 Exemple d'implémentation . . . 45
A.3 Programmation de la partie commune . . . 45
A.4 Programmation du service . . . 46
A.5 Programmation du client . . . 49

B Documentation Java (JavaDoc) 52

B.1 **package** edu.oleastre.computing.common 52

B.1.1 Interfaces 52

B.1.2 Classes 53

B.1.3 Exceptions 55

B.2 **package** edu.oleastre.computing.client 55

B.2.1 Interfaces 55

B.2.2 Classes 55

B.3 **package** edu.oleastre.computing.matrixmult 59

B.3.1 Interfaces 59

B.3.2 Classes 60

B.4 **package** edu.oleastre.computing.server 63

B.4.1 Interfaces 63

B.4.2 Classes 63

Table des matières 66

Table des figures 70

Liste des tableaux 71

Bibliographie 72

Table des figures

2.1 Conventions utilisées pour les diagrammes 7
2.2 Structure d'un système JINI 7
2.3 Passage d'objet et d'attributs du service vers le lookup 8
2.4 Passage d'objet, d'attribut, d'interface et de numéro d'identification du lookup vers le client 9
2.5 Dialogue entre client et service utilisant RMI 10

3.1 Architecture dédiée, cas de la multiplication de matrices 17
3.2 Diagramme de séquence pour la communication avec l'architecture dédiée. 19
3.3 Interactions possibles pour une architecture générique avec modifications de statut. 21
3.4 Diagramme de classes du client (Cas générique avec attribut) 23
3.5 Diagramme de séquences (Cas générique avec attribut) 25
3.6 Interactions possibles pour une architecture générique sans modifications de statut 26
3.7 Diagramme de séquences (Cas générique sans attribut) 28

A.1 Diagramme de classes pour le programme Hello World 46

Liste des tableaux

4.1 Résumé des résultats expérimentaux 36
4.2 Résultats obtenus en introduisant un temps de latence 38

Bibliographie

[1] Mosix.
`<http://www.mosix.org>`.

[2] Pov-ray the persistence of vision raytracer.
`<http://www.povray.org>`.

[3] Pvm: Parallel virtual machine.
`<http://www.epm.ornl.gov/pvm/>`.

[4] "seti@home".
`<http://setiathome.ssl.berkeley.edu>`.

[5] W. Keith Edwards. *Core Jini*. Prentice-Hall, first and second edition, 1999-2001.

[6] Franck Guerrero. Présentation de corba.
`<http://www.laas.fr/~killijia/corba_french_tutorial.html>`.

[7] Jan Newmarch. *Jan Newmarch's Guide to JINI Technologies*, 26 August 2000.
`<http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml>`.

[8] Scott Oaks and Henry Wong. *JINI in a Nutshell*. O'Reilly, first edition, March 2000.

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, October 2000.

[10] Gopalan Suresh Raj. A detailed comparison of corba, dcom and java/rmi.
`<http://www.execpc.com/~gopalan/misc/compare.html>`.

[11] SUN Microsystems. *Java™Remote Method Invocation Specification*, December 1999.
`<http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmi-title.html>`.

[12] SUN Microsystems. *A Collection of Jini™Technology Helper Utilities and Services Specifications*, October 2000.
`<http://www.sun.com/jini/specs/jini1.1html/collection-title.html>`.

[13] SUN Microsystems. *Jini™Architecture Specification*, October 2000.
`<http://www.sun.com/jini/specs/jini1.1html/jini-title.html>`.

[14] SUN Microsystems. *Jini™Technology Core Platform Specification*, October 2000.
`<http://www.sun.com/jini/specs/jini1.1html/core-title.html>`.

[15] Bill Venners. Jini versus corba.
`<http://www.artima.com/jini/jf/vision/messages/45.html>`.