UNIVERSITÉ
LAVAL

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE

Laboratoire de vision et systèmes numériques

DESCRIPTION DE JAVA RMI ET COMPARAISON AVEC CORBA ET DCOM

Document préparé par
Mario Couture

Août 2000

Table des matières

Table des matières ii
Liste des figures iii
Avant-propos iv
1.0 Introduction 1
2.0 Java RMI (*Remote Method Invocation*) 3
2.1 Vue globale de Java RMI 4
2.2 Le passage de paramètres 6
2.3 La communication entre les applications distantes 8
2.3.1 La couche application 8
2.3.2 La couche *Proxy* ou *Stub* 9
2.3.3 La couche de références distantes (*Remote Reference Layer*) 9
2.3.4 La couche transport (*Transport Layer*) 10
2.4 Le problème des murs de feu (*Fire Walls*) 10
2.5 Le gestionnaire de registres dans RMI (*Naming*) et la connexion client 11
2.6 La hiérarchie des classes et des interfaces en Java RMI 12
3.0 Revue des autres technologies de distribution d'applications 14
3.1 RPC (*Remote Procedure Call*) 14
3.2 CORBA (*Common Object Request Broker Adapter*) 15
3.3 DCOM (*Distributed Component Object Model*) 19
4.0 Exemple d'utilisation de Java RMI 23
4.1 Description de l'exemple 23
4.2 Partie serveur de l'application distribuée 25
4.3 Partie cliente de l'application distribuée 27
4.4 Compilation et exécution 28
5.0 Conclusion 34
Bibliographie 37
Annexe 1 38
Annexe 2 43

Liste des figures

Figure 1. Couches de communication associées à Java RMI (partiellement tidée de [2]).. 8
Figure 2. Hiérarchie des objets exportés (partiellement tiré de [2])..... 13
Figure 3. Le modèle d'architecture de l'OMG (tiré de [11]). 16
Figure 4. Architecture de l'ORB de CORBA (tiré de [11])..... 17
Figure 5. Composantes de DCOM (figure tirée de [12]). 19
Figure 6. *Use Case* des deux parties de l'application client-serveur. 24
Figure 7. Diagramme d'activités illustrant le fonctionnement du serveur. La déconnexion constitue une requête comme les autres. 24
Figure 8. Diagramme d'activités illustrant le fonctionnement du client. L'activité "déconnexion" est manquante au graphique. 25
Figure 9. Vue statique de l'application client-serveur (les deux modules). 26
Figure 10. Compilation et lancement du serveur. 29
Figure 11. Compilation et lancement de l'*Applet* client. 30
Figure 12. L'*Applet* client avant connexion au client. 30
Figure 13. L'*Applet* client après connexion au client et après avoir passé une requête. ... 31
Figure 14. Fenêtre DOS du serveur après le lancement de l'*Applet*, après la connexion du client au serveur et après avoir passé la requête au serveur. 31
Figure 15. Fenêtre DOS du client après le lancement de l'*Applet*, après la connexion du client au serveur et après avoir passé la requête au serveur. 32
Figure 16. Fichier `client.log` après le lancement de l'*Applet*, après la connexion du client au serveur et après avoir passé la requête au serveur. 32

Avant-propos

Le lecteur trouvera, dans ce document, beaucoup de termes techniques anglais qui n'ont pas été traduits en français. En informatique, il arrive souvent que la sémantique des termes anglais soit plus proche des concepts qu'ils représentent que celle des traductions françaises. Nous avons pris la liberté d'utiliser certains de ces termes anglais (ils sont en italique dans le texte).

De plus, certains chapitres contiennent des parties de code illustrant le propos du texte. Ces lignes de code, ainsi que toutes les commandes qui doivent être tapées au clavier par l'usager, ont été écrites en utilisant la fonte `Courrier New`.

Ce document se veut un résumé des lectures qui ont été réalisées sur Java RMI. La grande majorité des informations contenues dans ce document provient des sources listées dans la bibliographie de ce document. Pour des raisons de clarté, les citations n'ont pas été incluses à même le texte.

1.0 Introduction

Le besoin de communication entre applications informatiques n'a jamais cessé de croître avec la popularisation du réseau Internet et du WWW dans les années 80 et 90. L'architecture client/serveur est née avec la possibilité de transférer de l'information entre le **module** "serveur" et le **module** "client" d'une **application distribuée**.

À ses débuts, la technologie impliquait une programmation assez cryptique et difficile d'approche (les *Sockets BSD* sont un exemple). Entre autres, elle imposait au concepteur de logiciels de mettre au point des protocoles de communication relativement complexes. Ces protocoles devaient contrôler la transmission des données, effectuer les vérifications d'usage (représentation différente selon les plates-formes) et effectuer le dé/codage ces données. Le programmeur devait donc réserver une bonne part de son temps à la conception du protocole, hypothéquant le temps consacré au processus de conception du logiciel lui-même.

On a ensuite développé une technologie de plus haut niveau (*Remote Procedure Call*, RPC). RPC permet de séparer une application en deux modules (le serveur et le client) de fonctionnalités différentes, et de les distribuer sur des machines différentes. Encore utilisée aujourd'hui, cette technologie permet aux applications de provoquer l'exécution de procédures appartenant à des applications distantes et ce, de façon transparente. Avec cette technologie, le programmeur n'a plus à développer des protocoles gérant les méthodes de transferts, de vérifications et de dé/codages. Ces opérations sont entièrement prises en charge par RPC.

L'évolution des langages de programmation a amené de nouveaux outils aidant à la conceptualisation des problèmes en informatique. En effet, la venue de l'orienté-objet facilite l'abstraction du problème à résoudre en fonction des données du problème lui-même (par l'utilisation de classes et d'objets). Aussi, la venue de la programmation-objet a donné lieu à de nouvelles technologies de distribution des applications informatiques. Les concepts d'héritage, d'encapsulation et de polymorphisme sont mis à profit avec l'orienté-objet.

Java RMI est un exemple de technologie orienté-objet (grâce à Java) permettant de distribuer une application (grâce à RMI). Entre autres, RMI permet le transfert de comportements (objets) d'un module à un autre. Un système informatisé de collecte des dépenses des employés d'une entreprise est un bon exemple. Dans ce cas, le département des finances désire récolter les dépenses mensuelles de ses employés, mais la politique de dépense change de mois en mois. Le problème des multiples mises à jour des applications clientes se pose, il y en aurait une à chaque mois pour chaque employé. La solution "RMI" permet d'éliminer ce problème en exportant, sur demande des clients, l'objet contenant la politique en vigueur. Ces derniers utilisent alors cet objet avant de transmettre leur rapport de dépenses "adapté" au serveur.

Les applications distribuées peuvent prendre plusieurs formes. Le système le plus simple est constitué d'un client et d'un serveur échangeant des informations et ce, sans autre interaction avec le reste du ou des réseaux. Le serveur attend et répond aux requêtes des clients, et les clients transmettent leurs requêtes (demande de connexion et autres commandes). Les informations peuvent être échangées de façon bidirectionnelle entre les deux modules.

Le client et le serveur peuvent, à leur tour, devenir simultanément serveur et client pour d'autres applications distantes. On a alors un réseau mixte de modules distribués. Chaque module peut alors jouer le rôle de client, de serveur ou les deux à la fois. Ces modules constituent une application distribuée s'ils interagissent ensemble (exclusivement) pour régler un problème commun. Dans le cas contraire, ils constituent plusieurs applications distribuées (les modules, qui ne se connaissent pas *à priori*, travaillent à régler plusieurs problèmes qui ne sont pas nécessairement reliés).

Finalement, ces modules peuvent avoir une relative autonomie de fonctionnement et de prise de décision. On tend à leur donner le nom d'agent, nous n'entrons pas dans le détail des agents dans ce document.

La distribution d'applications (sur le même, ou sur plusieurs ordinateurs) se fait en utilisant un *Framework*, lequel joue le rôle de pont de communication entre les modules de l'application distribuée. En effet, ces *Frameworks* (RMI, CORBA et DCOM sont des exemples bien connus) fournissent tous les outils nécessaires à la gestion de la communication entre les modules. Ils sont appelés *Middlewares* puisqu'ils ne sont pas vraiment des exécutables, leur structure est répartie entre les modules de l'application.

Nous nous proposons, dans ce document, d'examiner les fonctionnalités de Java RMI et d'en faire ressortir tant les possibilités que les limites. Le chapitre 2 donne une description relativement complète de cette technologie. Afin d'aider à mieux situer RMI dans l'univers des *Middlewares*, les deux technologies concurrentes (CORBA et DCOM) sont ensuite brièvement décrites au chapitre 3. On retrouve dans le chapitre 4 un exemple générique d'application de type client/serveur, il met en valeur plusieurs concepts propres aux *Middlewares*. Une comparaison "haut niveau" entre RMI, CORBA et DCOM est présentée en guise de conclusion au chapitre 5.

2.0 Java RMI (*Remote Method Invocation*)

Java RMI propose une méthode relativement simple et puissante pour distribuer des applications sur des machines virtuelles différentes (une machine virtuelle étant une instance de l'interpréteur de *Bytecode*).

C'est à partir de JDK 1.1 que le concept original de machine virtuelle de Java a été étendu à un nouveau modèle dans lequel les classes et les objets peuvent fonctionner dans un environnement réseau (les machines virtuelles peuvent être localisées à la fois sur le même ordinateur ou encore sur un réseau).

Toutes les fonctionnalités de Java sont disponibles au programmeur RMI (entre autres celles de la sécurité, du processus de *Serialization* des données, de la connexion aux systèmes déjà existants par JNI et aux bases de données par JDBC). Les principaux avantages à utiliser RMI (et Java) sont les suivants:

- À la base, Java est entièrement basé sur le concept de l'orienté-objet. Les concepts de réutilisabilité (héritage), de protection de l'information (encapsulation) et de l'accès dynamique (polymorphisme) sont directement utilisables en Java. De plus, Java ne permet pas l'héritage multiple.
- Il permet le transfert transparent d'objets d'une application à une autre et ces objets ne subissent pas de changement lors des transferts (l'intégrité des objets transférés est respectée).
- Son utilisation est sécuritaire (les classes de gestion de la sécurité sont disponibles et le programmeur peut, s'il le désire, programmer les siennes).
- Contrairement à un *Framework* comme MFC, il est relativement facile et rapide d'écrire des applications Java et d'utiliser les classes RMI.
- Il permet un accès facile et simplifié aux systèmes déjà en place (utilisant des langages comme Java, C, C++, ...).
- Il élimine le coulage de mémoire par l'utilisation du *Garbage Collection*.
- Il permet l'exécution en parallèle d'applications (locale ou distante).
- Le transfert de comportements (objets) d'une application à une autre permet d'exploiter toute la force des *Design Patterns* (qui s'appuient très souvent sur des comportements).
- Depuis JDK 1.2, Java supporte l'activation d'objets distants.
- Depuis JDK 1.3, Java supporte les spécifications IIOP (Internet Inter-ORB Protocol) du Object management Group (OMG). Cette nouvelle fonctionnalité de Java (maintenant appelée RMI-IIOP) permet l'intégration de Java et de CORBA.

Les architectes de Sun ont réussi à rendre transparente l'utilisation d'objets distants. Après avoir localisé les objets appropriés, le programmeur utilise les méthodes de ces objets comme si ces dernières faisaient partie de son application. Tout le travail de codage, de décodage, de vérification et de transfert est effectué par RMI.

2.1 Vue globale de Java RMI

Toutes les applications distribuées utilisant RMI sont faites de la même façon. Elles sont divisées en au moins deux modules; le client et le serveur. Le serveur rend disponibles des objets aux clients en les enregistrant au gestionnaire de registres `rmiregistry`.

Du point de vue du serveur, les méthodes d'un objet sont rendues disponibles pour les clients distants grâce aux opérations suivantes :

1. Définir une interface publique qui hérite de `java.rmi.Remote`.
2. Dans cette interface, déclarer les méthodes qui seront vues et utilisées par les clients distants. Chacune de ces méthodes doit définir `java.rmi.RemoteException`.
3. Créer l'implémentation du serveur qui hérite de Java.rmi.UnicastRemoteObject et qui implémente l'interface définie en 1.
4. Implémenter les méthodes définies dans l'interface (et possiblement d'autres) dans l'implémentation du serveur.
5. La méthode `main()` du serveur exporte les méthodes en liant l'objet serveur au gestionnaire de registres (préalablement lancé par l'application, ou par l'usager avec l'utilitaire `rmiregistry`)

Le client, ayant un lien local ou réseau avec l'ordinateur du serveur, contacte le gestionnaire de registres pour connaître l'emplacement exact de l'objet distant cherché (en spécifiant l'adresse IP ou DNS du serveur et le nom de l'objet exporté). Une fois l'adresse connue, le client n'a plus à contacter ce service. Il effectue les transactions directement avec le serveur en utilisant l'adresse obtenue.

Lors du passage d'objets d'une machine virtuelle à une autre, Java RMI traite les objets distants d'une façon différente des objets non-distants. Les objets distants sont accédé par le biais d'un *Stub*, lequel agit comme un représentant de l'objet distant (un *Proxy*) auprès du client. Un appel à une méthode distante passe par le *Stub* et ce dernier est responsable de transmettre l'appel à l'objet distant. Un *Stub* implémente la même interface que l'objet distant. Ceci explique pourquoi un client ne peut voir que les méthodes déclarées dans l'interface de l'objet distant.

Il existe des nuances importantes entre un objet local et un objet distant (vue du client), en voici une liste non-exhaustive.

- Un objet local est implémenté localement avec une classe Java alors qu'un objet distant est rendu publique avec une interface héritant de `java.rmi.Remote`. Son implémentation est réalisée par la classe Java qui implémente l'objet distant (le serveur).

- Pour un objet local, une nouvelle instance d'un objet est réalisée localement par l'opérateur `new` alors qu'une nouvelle instance d'un objet distant est réalisée sur l'ordinateur hôte par l'opérateur `new`. Un client ne peut directement créer un nouvel objet distant (à moins d'utiliser *Remote Object Activation*).
- Un objet local est accédé directement à l'aide d'une variable référence à l'objet alors qu'un objet distant est accédé à l'aide d'une variable référence à l'objet qui pointe vers le *Stub*.
- Dans une application non-distribuée, la référence à un objet pointe directement vers l'objet dans le *Heap* alors que, pour un objet distant, une référence distante est un pointeur à un objet *Stub* dans le *Heap* local. Le *Stub* contient l'information qui lui permet de se connecter à un objet distant.
- Dans une application non-distribuée, un objet est considéré vivant s'il y a au moins une référence qui lui est rattachée. Dans un environnement distribué, il peut survenir des problèmes détruisant une connexion par exemple. Un objet est considéré comme ayant une référence distante vivante s'il a été accédé depuis un certain temps (*The Least Period*).
- Quand une référence locale à un objet a été posée à nulle, cet objet est un candidat pour le *Garbage Collection*. Le *Garbage Collector* distant fonctionne avec le *Garbage Collector* local. S'il n'y a pas de référence distante et que toutes les références aux objets distants ont été posées à nulle, alors les objets distants deviennent des candidats pour le *Garbage Collector*.
- Si un objet local implémente la méthode `Finalize()`, cette méthode est appelée avant que le *Garbage Collection* ne fasse son travail. Si un objet distant implémente la méthode `Unreferenced()`, cette méthode de l'objet non référencé est appelée quand toutes les références distantes sont nulles.
- Le compilateur Java force un programme à considérer toutes les exceptions. RMI force les programmes à considérer toutes les exceptions de type `RemoteException` qui peuvent survenir.
- Comme pour les interfaces locales, les interfaces distantes peuvent contenir des champs statiques. Un initialiseur est lancé pour chaque machine virtuelle qui charge l'interface distante, créant une nouvelle variable statique. Une copie séparée de cette variable existe dans chaque machine virtuelle.

En Java, le chargeur de classe est responsable de charger dynamiquement en mémoire les classes nécessaires à une application. Différent types de *Class Loader* existent et Java RMI utilise la classe `RMIClassLoader`. `RMIClassLoader` a pour fonction de charger le *Stub* et le *Skeleton* utilisés par le système RMI ainsi que d'autres classes utilitaires secondaires.

En général, `RMIClassLoader` tentera de charger les classes à partir du système local en utilisant la variable d'environnement `CLASSPATH`. Si les classes ne peuvent être chargées localement, `RMIClassLoader` tentera d'extraire le URL de l'objet *Marshalled* (*Serialized*, voir plus loin) et de l'utiliser comme *Codebase* pour localiser et télécharger les classes cherchées.

2.2 Le passage de paramètres

Il existe des nuances importantes concernant le passage de paramètres dans une application utilisant une seule machine virtuelle et une application distribuée utilisant plusieurs machines virtuelles. Ces différences sont expliquées dans cette section.

Pour un machine virtuelle Java simple:

En général, Java passe les paramètres par valeur. Quand un paramètre est passé à une méthode, Java fait une copie de la valeur du paramètre, place cette copie sur le *Stack* et exécute la méthode. Quand la méthode a besoin du paramètre, un accès est fait au *Stack* pour prendre la valeur du paramètre. Les valeurs retournées par les méthodes sont copiées de la même façon.

Pour les types simples (`boolean, byte, short, int, long, char, float ou double`) le processus est simple. Par contre, le processus de passage d'objets comme paramètres est plus complexe. En Java, les objets résident dans le *Heap* et sont accédé par des variables références. L'exemple suivant montre le processus:

```
String laString = "La valeur de la String" ;
System.out.println(laString) ;
```

Dans la deuxième ligne de code, une copie de la valeur de la référence à l'objet `laString` est placée sur le *Stack*. La méthode utilise la copie de cette référence pour accéder à l'objet. Il est important de mentionner que la méthode recevant la copie de la référence à un objet comme paramètre a un impact définitif sur l'objet. Si par exemple, la méthode modifie un des paramètres de l'objet en question, la modification est tout de suite effective et définitive (même à la sortie de la méthode).

Pour un machine virtuelle Java RMI:

Quand un paramètre de type simple est passé à une méthode d'un objet distant, RMI passe ce paramètre par valeur en effectuant une copie du type simple, et en transmettant cette copie à la méthode distante. Si la méthode retourne un type simple, il est aussi est retourné par valeur.

Lorsqu'un objet est passé à une méthode distante, le processus est différent de celui décrit plus haut (pour une machine virtuelle simple). RMI envoi une copie de l'objet lui-même (pas une copie de sa référence). L'objet est passé par valeur, ce n'est pas sa référence qui est transmise. De la même façon, le retour d'objet par une méthode transmet une copie de l'objet lui-même.

Le passage d'objets entre machines virtuelles pose le problème des objets complexes (héritage, agrégation, composition, association, …). Pour que la machine virtuelle qui reçoit l'objet soit en mesure de reconstituer cet objet dans toute sa complexité, il faut que toute la structure de l'objet soit transférée. Le programmeur doit donc porter attention aux objets qu'il désire transférer. De grosses structures peuvent engorger exagérément le CPU des ordinateurs, ainsi que la largeur de bande disponible.

RMI règle le problème du transfert des objets complexes en utilisant le *Object Serialization*. Ce processus transforme l'objet (et ses références) en une suite linéaire (nommé *Stream*) facilement transférable entre machines virtuelles. Les objets ainsi linéarisés (*Marshalled*) sont ensuite délinéarisés (*Unmarshalled*) sur réception.

RMI introduit un autre type de paramètre qui est la référence à un objet distant (permettant les *Callback* par exemple). L'exemple suivant illustre les *Callbacks* (l'exemple du chapitre 5 contient également un *Callback*) :

```
// Deux objets dont les classes ont été définies ailleurs
GestionnaireBanque gestionnaireBanque ;
Compte referenceAuCompte ;

// L'adresse RMI du service
String adresseRMIService = "rmi://Banque/ServiceGestionBanque" ;

Try
{
   // Trouver l'adresse de l'objet distant
   gestionnaireBanque = (GestionnaireBanque)
                              Naming.Lookup(adresseRMIService) ;

   // Obtenir la reference
   referenceAuCompte = gestionnaireBanque.getCompte("Celine Dion") ;

   // MMmmm, voir le contenu du compte (pour ensuite transférer les $)
   System.out.Println(referenceAuCompte.getLeContenu() ) ;
}
catch (RemoteException e)
{
   System.out.println("Erreur : RemoteException dans le programme XXX :
                       " + e ) ;
   System.exit(1) ;
}
```

La référence au compte de Céline Dion (l'objet) permet d'invoquer les méthodes de cet objet et de consulter le contenu de celui-ci.

D'une machine virtuelle à l'autre, les valeurs sont passées de façon à ne pas dépendre des plates-formes.

2.3 La communication entre les applications distantes

La communication RMI est essentiellement réalisée en utilisant 4 couches inter-reliées; la couche application, la couche *Proxy* ou *Stub*, la couche *Remote Reference* et la couche *Transport* (voir la figure 1). L'utilisation d'une architecture divisée en plusieurs couches permet la modification d'une de ces couches sans affecter le reste du système. Par exemple, la couche de transport pourrait être remplacée par une autre couche utilisant le protocole UDP et ce, sans affecter les autres couches (TCP de TCP/IP est habituellement utilisé) .

2.3.1 La couche application

La première couche est constituée par les modules client et serveur. On y retrouve les appels de haut niveau pour exporter les objets et y accéder.

L'architecture RMI est basée sur le fait que le comportement exporté (les méthodes) est défini dans l'interface (définition des services). L'interface ne contient pas de code exécutable et elle hérite de `java.rmi.Remote`. L'interface `Remote` est essentiellement utilisée pour déclarer aux éventuels clients l'accessibilité à distance d'un objet. Son implémentation est réalisée dans une classe du module serveur.

Le client et le serveur sont exécutés en utilisant deux machines virtuelles différentes.

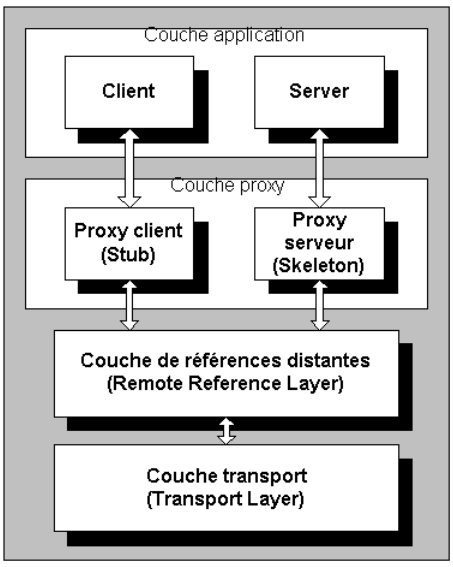

Figure 1. Couches de communication associées à Java RMI (partiellement tidée de [2]).

2.3.2 La couche *Proxy* ou *Stub*

Les deux parties de cette couche (le *Stub* et le *Skeleton*) sont générées lors de la compilation de l'implémentation et de l'interface. L'utilitaire `rmic` est utilisé pour effectuer la compilation (voir la section 4.4). Dans la version 1.2 de JDK, le compilateur ne génère qu'un *Stub*, le *Skeleton* n'est plus nécessaire.

Comme il a été mentionné plus haut, le *Stub* est le lien entre le client et l'objet distant. Cette couche intercepte donc les appels faits aux méthodes des objets distants par le client. Elle les redirige ensuite vers le service RMI approprié.

Du point de vue du client, l'objet distant est d'abord localisé, créé et il est ensuite retypé (*Casted*) du nom de l'interface. Lorsque le client effectue un appel à une des méthodes de l'objet distant, cet appel est transféré au *Stub*. Ce dernier effectue ensuite le travail de base (*Marshalling*) pour envoyer l'information vers le serveur sous la forme de *Streams*. Rendue au serveur (*Skeleton*), l'information subit la transformation inverse (*Unmarshalling*) et la méthode est appelée avec les bons paramètres. Si la méthode produit un retour, cette information subit le chemin inverse vers le client. Pendant la connexion entre un serveur et un client, Java remplace les objets distants par leur *Stub*, peu importe leur localisation dans la structure.

Le *Skeleton* (ou le serveur lui-même pour JDK 1.2) est localisé du côté serveur et fonctionne directement avec l'implémentation des méthodes exportées. Ses fonctions sont essentiellement les mêmes que celles du *Stub*, en plus de recevoir les appels de ce dernier.

2.3.3 La couche de références distantes (*Remote Reference Layer*)

Cette couche définit et supporte la sémantique des invocations de la connexion RMI. Elle interprète et gère les références des objets distants. La couche de références distantes est en fait une couche abstraite entre le *Stub,* le *Skeleton* et les protocoles de communication (qui sont gérés par la couche de transport). Les interactions entre ces deux couches seront toujours effectuées comme si la connexion était de type connecté (avec un *Stream*), mais le transport de l'information peut quand même être réalisé en utilisant un protocole non-connecté (UDP de TCP/IP).

Dans la version 1.1 de JDK, cette couche connecte les clients aux objets distants par une connexion de type un à un (*Unicast*). Avant qu'un client ne puisse utiliser les services d'un objet distant, le service doit être instantié sur le serveur et exporté au système RMI.

Dans JDK 1.2, cette couche a subi des améliorations visant à supporter des objets distants dormants (*Remote Objet Activation*). Quand un appel à une méthode d'un objet distant dormant est effectué par le client, RMI détermine l'état du service. S'il est effectivement dormant, RMI instantit l'objet et restore son état.

D'autres types de connexion sont également possibles. Par exemple, en mode *Multicast* un simple *Stub* pourrait simultanément envoyer une requête à plusieurs implémentations distantes d'un objet et accepter la première réponse.

2.3.4 La couche transport (*Transport Layer*)

C'est la couche de transport qui effectue la connexion entre les machines virtuelles et ce, même en présence d'obstacles réseaux (murs de feu). Elle est responsable de la mise en contact des machines virtuelles, de la gestion des connexions, de la vérification des connexions mortes et de la réception de demandes de connexion distantes.

Toutes les connexions sont des *Streams* et utilisent le protocole TCP de TCP/IP. Il est également possible d'utiliser le protocole UDP à la place de TCP. Les connexions sont basées sur les adresses IP et les numéros de port des ordinateurs communiquant. Un nom DNS peut être utilisé à la place de l'adresse IP.

En parallèle avec TCP/IP, RMI utilise un protocole propriétaire de bas niveau appelé *Java Remote Method Protocol* (JRMP). Deux versions de JRMP existent. La première est associée à la version 1.1 de JDK. Elle oblige l'utilisation d'un *Skeleton* du côté serveur. La deuxième version est apparue avec la version 1.2 de JDK et élimine l'utilisation du *Skeleton*. Le compilateur `rmic` peut générer les classes de la couche *Proxy* selon les deux versions (avec le paramètre `-v1.1` ou `-v1.2`).

Comme il a été mentionné plus haut, l'architecture en couches du processus de communication permet le remplacement du protocole de bas niveau (de la couche de transport) par d'autres alternatives. Il est également possible de modifier cette couche afin de permettre le transport de *Streams* encryptés, l'intégration de nouveaux algorithmes de sécurité et l'optimisation de performance. Ces changements n'auront aucun impact sur les couches supérieures.

Même si deux machines virtuelles fonctionnent sur le même ordinateur, la connexion RMI doit passer par le service TCP/IP du système d'opération. La couche de transport permet donc les connexions RMI locales (sur le même ordinateur) ou réseau.

2.4 Le problème des murs de feu (*Fire Walls*)

Dans le domaine des applications de type client/serveur, les murs de feu posent souvent un problème aux concepteurs de logiciels. Typiquement, ces murs bloquent tout le trafic réseau sauf celui associé à certains ports bien identifiés (comme le port 80 par exemple).

Comme la couche de transport de RMI utilise des sockets (ces ports sont bloqués par les murs de feu) pour faciliter la communication, le JRMP (*Java Remote Method Protocol*) se trouve bloqué par les murs de feu. Les designers de RMI ont anticipé ce problème et

ont développé une solution à l'intérieur de la couche de transport de RMI. Pour traverser les murs de feu, RMI encapsule les appels aux objets distants à l'intérieur de requêtes HTTP POST.

Cas d'un client derrière un mur de feu:

Quand la couche de transport tente d'établir une communication avec le serveur et qu'un mur de feu empêche la connexion, RMI relance la tentative de connexion en encapsulant l'appel (JRMP) dans une requête HTTP POST. L'en-tête de la requête POST prend alors la forme suivante:

```
http://hostname:port
```

Comme la plupart des murs de feu reconnaissent le protocole HTTP, le serveur *Proxy* est habituellement capable de transférer l'appel à l'extérieur du mur et sur le bon port (celui que le serveur écoute).

Cas du client et du serveur derrière deux murs de feu:

Dans ce cas, le mur de feu empêche la réception des appels, la couche de transport RMI utilise une autre méthode. Elle place l'appel (JRMP) dans des *Packets* HTTP et les transfère au port 80 du serveur. L'en-tête de la requête POST prend alors la forme suivante:

```
http://hostname:80/cgi-bin/java-rmi?forward=<port>
```

Ceci cause l'exécution d'un script CGI, lequel invoque la machine virtuelle Java, décode les *Packets* et transfère l'appel au serveur sur le bon port. Le retour d'information s'effectue de la même manière.

Il est évident que des dégradations de performances accompagnent bien souvent ces deux solutions.

2.5 Le gestionnaire de registres dans RMI (*Naming*) et la connexion client

Une étape importante permettant la réalisation de la communication entre le client et le serveur est le processus d'identification des objets distants sur le réseau (*Naming*). C'est en utilisant les API de RMI (`Naming.lookup()`) que le client obtient l'adresse complète de la localisation de l'objet distant et qu'il arrive à se connecter au serveur et ainsi, à invoquer les méthodes des objets distants.

RMI utilise le gestionnaire de registres (côté serveur) qui sert en quelque sorte de bottin téléphonique tant pour le serveur (pour s'enregistrer) que pour le client (pour y trouver une adresse de l'objet cherché). Chaque ordinateur hôte, offrant un service local ou

distant d'objets et qui accepte les requêtes des clients pour ces objets, doit avoir un gestionnaire de registres qui fonctionne (par défaut sur le port 1099).

Le gestionnaire de registres peut être lancé par l'utilisateur à l'aide de `rmiregistry`. Les API de RMI permettent de plus de lancer ce service à même l'application serveur (dans l'application du chapitre 5 de ce document, le serveur lance lui-même le gestionnaire de registres).

Une fois le service lancé, le serveur peut associer les objets qu'il exporte à un nom public, reconnaissable par les éventuels clients. L'application serveur est ensuite placée en attente de clients. Normalement, une application serveur, pouvant être connectée avec plusieurs clients simultanément, traite ces derniers dans des *Threads* différents.

Comme il a été mentionné plus haut, le client accède le gestionnaire de registres (côté serveur) en utilisant la méthode `lookup`() de la classe `Naming` pour localiser l'objet cherché. Il utilise un URL (Uniform Resource Locator) de la forme:

```
rmi://<ordinateur hôte>[:<port>/<nom du service>]
```

où le nom de l'ordinateur hôte doit être reconnu dans un réseau local ou par un service de nom de domaine (DNS) sur Internet. Le numéro du port doit être spécifié seulement si le service de registes côté serveur utilise un autre port que le 1099.

Les étapes à la connexion du client sont les suivantes:

1. Le client doit d'abord obtenir une référence au serveur. Il utilise donc la méthode `Naming.lookup()`.
2. La méthode `lookup()` utilise le *Stub* pour faire un appel distant au `rmiregistry`.
3. Le `rmiregistry` retourne la référence de l'objet distant demandé à la couche transport (côté client). Chaque référence distante contient le nom du serveur et le port que les clients peuvent utiliser pour se connecter à la machine virtuelle contenant l'objet distant.
4. Une fois la référence du serveur obtenue, la couche transport (côté client) utilise le nom et le numéro de port pour ouvrir une connexion de type socket au serveur.
5. La référence obtenue est transférée à la couche application (côté client).

2.6 La hiérarchie des classes et des interfaces en Java RMI

La figure 2 illustre la hiérarchie et les dépendances d'un objet exportable. La définition des différentes composantes de cette figure est la suivante: La classe `java.lang.Object` est la classe de base à tous les objets en Java. La classe `java.rmi.server.RemoteObject` est une interface héritant de `Object` et de

l'interface `java.rmi.server.Remote`. La classe `java.rmi.server.RemoteServer` hérite de `RemoteObject`.

Dans l'interface qui définit les méthodes distantes, le programmeur n'utilise (par héritage) que l'interface `Remote`. Dans l'implémentation de l'interface, il n'utilise (par héritage) que la classe `java.rmi.server.UnicastRemoteObject` (et possiblement `Activable`) et n'implémente que l'interface `MonInterfaceDistante`.

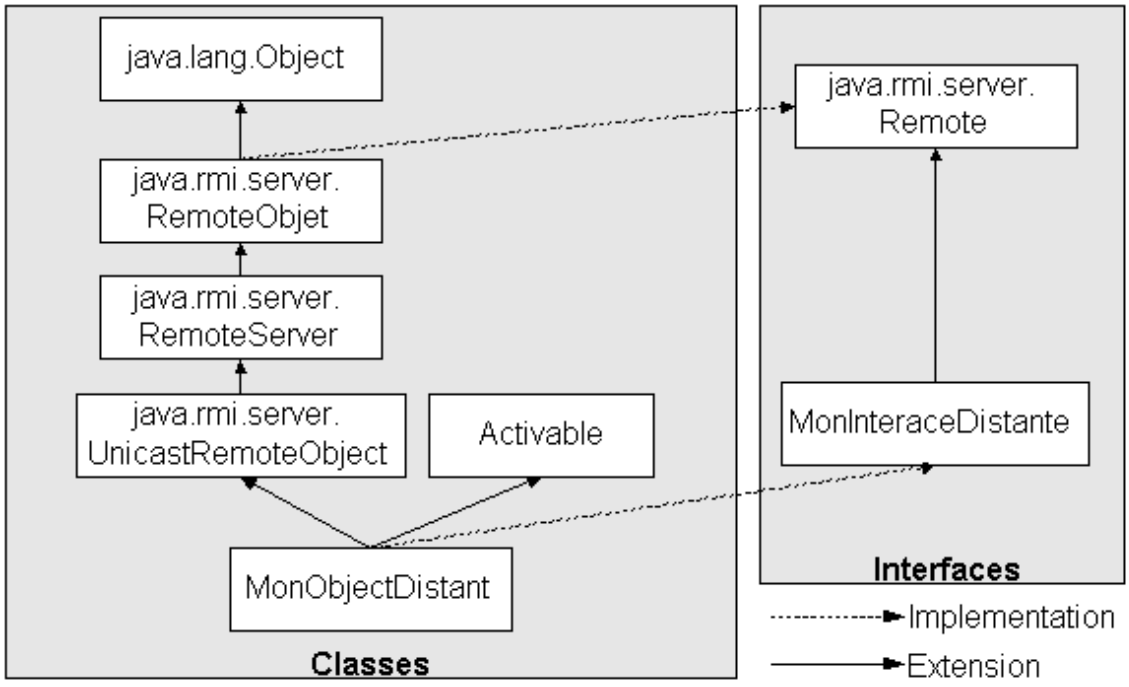

Figure 2. Hiérarchie des objets exportés (partiellement tiré de [2]).

Le lecteur trouvera une brève description des interfaces, des classes et des *Exceptions* de Java RMI (JDK 1.2) en annexe 1 de ce document.

3.0 Revue des autres technologies de distribution d'applications

Nous décrivons, dans ce chapitre, les principales technologies utilisées pour distribuer des applications (RPC, CORBA et DCOM).

3.1 RPC (*Remote Procedure Call*)

La technologie RPC permet à un client d'exécuter une fonction d'un serveur distant. C'est un système essentiellement basé sur la procédure, qui n'intègre pas l'orienté-objet (exception faite de DCE-RPC).

Les caractéristiques principales de RPC sont les suivantes:

- Les procédures peuvent être distribuées sur la même machine ou sur un réseau.
- C'est un système non-connecté (*Connectionless*). La connexion n'existe que pour le temps de l'appel à la procédure.
- Les détails du mécanisme de transport utilisés par RPC sont cachés au programmeur. L'appel à une procédure distante apparaît comme s'il était fait localement.
- RPC supporte la portabilité et l'interopérabilité et ce, sous différents systèmes d'exploitation.
- RPC supporte également le *Multithreading*, la sécurité réseau, l'intégrité des ressources et l'intégrité des données (lors des transferts).

L'application distribuée est divisée en deux modules; le client et le serveur. Comme pour Java RMI, les applications RPC doivent s'enregistrer (au *Port Mapper*) afin d'être visibles pour les clients. Le client demande ensuite un ou des services (de procédures) au *Port Mapper*, lequel retourne l'adresse du service demandé. Le *Port Mapper* peut déservir plusieurs services différents et le serveur peut déservir plusieurs clients simultanément.

La communication entre le serveur et le client se fait en passant par le *Stub* côté client et le *Skeleton* côté serveur. C'est au niveau du *Stub* et du *Skeleton* que se fait tout le travail de transaction entre les deux parties de l'application distribuée.

Ce système représente une simplification importante par rapport aux sockets. Par exemple, le développement de protocoles de communication complexes n'est plus nécessaire avec RPC. Un inconvénient accompagne toutefois cette technologie. On doit créer une interface entre le client et le serveur et la compiler séparément.

RPC existe sous plusieurs bannières. On note entre autres: OSF RPC, Sun RPC, Netwise RPC, IBM OS/2 DPL et Sybase RPC.

3.2 CORBA (*Common Object Request Broker Adapter*)

CORBA est un standard qui a été développé par l'OMG (*Object Management Group*). Plus spécifiquement, c'est un *Framework* intégrant les spécifications de l'OMG qui est utilisé pour distribuer des applications de façon indépendante aux plates-formes et aux langages de programmation. Sur le marché, plusieurs vendeurs fournissent leur version de CORBA (plus de 20).

CORBA permet à une application cliente d'effectuer une requête à un objet distant (serveur) et de recevoir en retour le résultat de l'opération effectuée. Tout le travail de communication est automatiquement effectué par le *Middleware* (processus de localisation des objets distribués, le *Un/Multiplexing*, la gestion des erreurs de communications, le *Un/Marshaling*, etc).

Les principales caractéristiques de CORBA sont les suivantes:

- CORBA offre une complète interopérabilité. Les applications intégrant CORBA communiquent en utilisant le protocole de communication IIOP (un sous-ensemble de GIOP) qui est un véritable standard de l'industrie. Ce protocole ne dépend pas des langages de programmation ni des plates-formes. Il peut faire communiquer n'importe quel élément supportant TCP/IP.
- Intégration aux systèmes existants. L'architecture de CORBA est ouverte et permet l'intégration de systèmes hétérogènes.
- Les objets CORBA externationalisent leur définition dans une interface. L'implémentation et la définition d'un objet CORBA sont complètement dissociées.
- CORBA entretient un lien privilégié avec la dernière version de Java (et le C++).

Pour décrire l'architecture de CORBA il faut d'abord parler de l'OMA (*Object Management Architecture*). L'OMA est l'architecture globale respectant toutes les spécifications de l'OMG. L'OMG a donc créé l'OMA qui contient plusieurs éléments dont l'ORB, qui en est la partie centrale. CORBA implémente uniquement les fonctionnalités de l'ORB et non l'ensemble du *Framework* OMA. Par abus de langage, on confond OMA et CORBA. CORBA est devenu le nom désignant le *Middelware* s'appuyant sur les spécifications de l'OMG.

L'ORB (*Object Request Broker*) est au coeur de CORBA. Les services (ou facilités) disponibles aux programmeurs CORBA gravitent autour de l'ORB. Ces services sont les suivants (la figure 3 illustre l'OMA):

- **Les *Object Services***. Ces interfaces sont utilisées par les développeurs d'applications distribuées pour effectuer des tâches précises. Des exemples de ces services sont: le *Naming Service* (permet aux clients de trouver les objets avec le nom de ceux-ci), le *Trading Service* (permet aux clients de trouver les objets en

utilisant leurs propriétés), les services de sécurité, de transactions et plusieurs autres. Chaque vendeur d'ORB est libre d'implémenter le ou les services qu'il désire. C'est en fait un facteur déterminant lors du choix d'un d'ORB.

- **Les *Common Facilities***. Les services communs se placent à un niveau d'abstraction supérieur; ce sont les outils utilisés par les applications. Citons par exemple les gestionnaires d'impression, les outils de gestion documentaire ou les boîtes à outils.
- **Les *Domain Interfaces***. Ce sont les services dédiés à des champs d'applications donnés. Une fédération de Caisses populaires utilisant une application distribuée dans chacune de ses succursales par exemple, pourrait avoir un service propre aux caisses (commun à toutes les succursales).
- **Les *Application Interfaces***. Les objets développés par les programmeurs, qui s'insèrent dans l'architecture prévue par l'OMG, sont vus comme des objets spécialisés qui utilisent tous les services, utilitaires et interfaces.

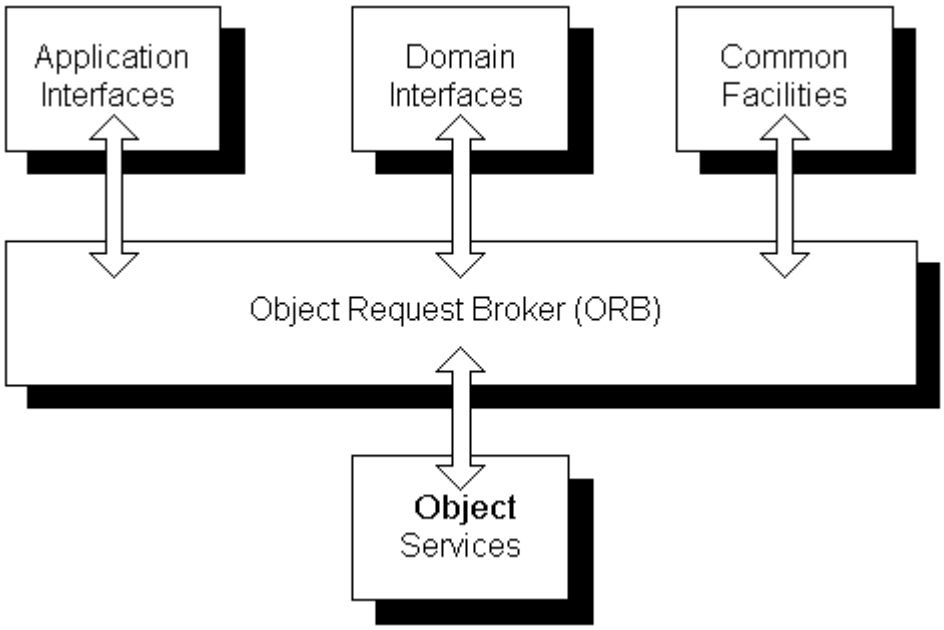

Figure 3. Le modèle d'architecture de l'OMG (tiré de [11]).

L'ORB est divisé en plusieurs parties de fonctionnalité différente. La figure 4 (tiré de [11]) illustre l'architecture de CORBA.

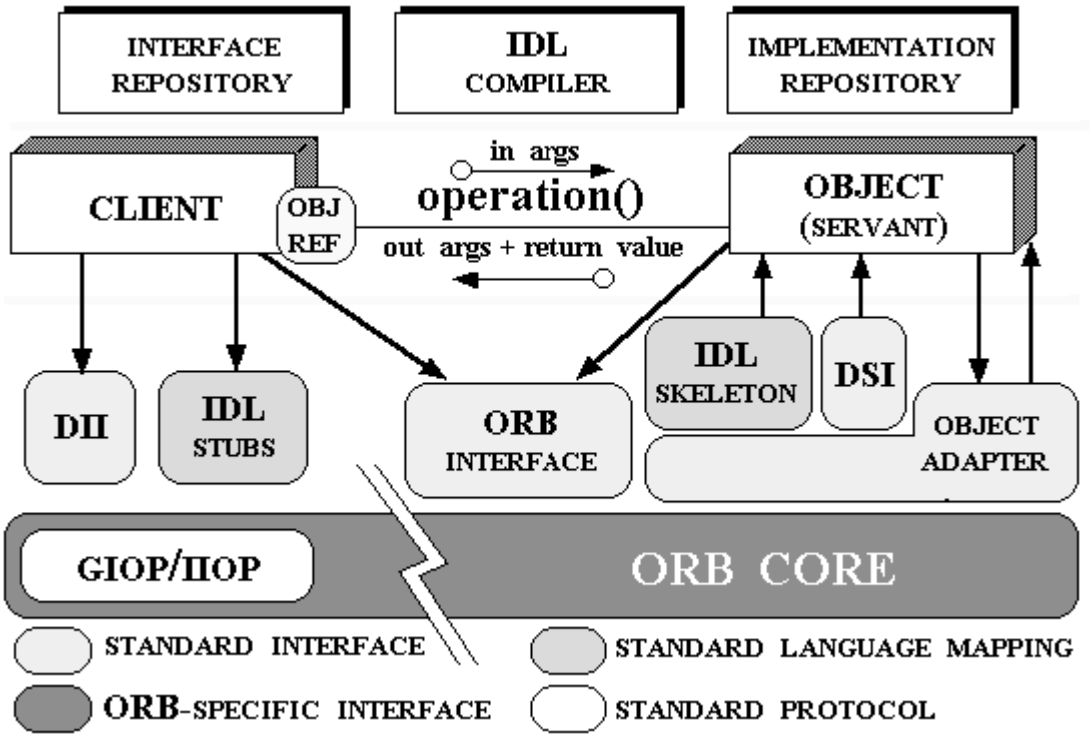

Figure 4. Architecture de l'ORB de CORBA (tiré de [11]).

Les éléments de l'architecture sont les suivants:

- **L'ORB (*Object Request Broker*)**. Souvent appelé bus logiciel, l'ORB est au coeur de l'architecture définie par l'OMG. Il est responsable de la mise en relation et de la communication entre tous les éléments présents. Il prend en charge et rend transparent le dialogue entre le serveur et les clients. Entre autres, il effectue des opérations de gestion ou de maintenance comme la gestion des erreurs ou leur destruction.
- **L'interface ORB**. C'est l'interface au ORB, la même pour tous les ORB. Elle ne dépend pas de l'interface objet ou de l'*Object Adapter*.
- **Le client**. Le client d'un objet distribué a accès à une référence à cet objet et à la possibilité d'invoquer ses méthodes. Le client ne connaît que la structure logique de l'objet distribué (par l'interface) et il expérimente les comportements de l'objet par l'invocation des méthodes. Il est important de mentionner qu'un objet serveur/client peut à son tour devenir client/serveur pour d'autres objets.
- **Le serveur (implémentation de l'objet distribué)**. L'implémentation fournit la sémantique de l'objet en spécifiant les méthodes et les variables de l'objet distribué. Généralement, elle ne dépend pas de l'ORB ou de la façon dont le client invoque l'objet.

- **L'*Object Reference* (*Obj. Ref.*)**. L'étape de connexion entre un client et un serveur est appelée *Binding*. Elle consiste à obtenir une référence à un objet distant afin de l'identifier sans ambiguité (la référence se nomme: *Interoperable Object Reference*, IOR). Lors du *Binding*, cette référence est retournée au client (au *Stub*) qui le stocke dans une variable interne du *Stub* (et non du client comme sur la figure). Ensuite, lors de chaque accès à l'objet, cette référence est utilisée.
- **L'*Interface Definition Language* (IDL) de l'OMG**. L'interface IDL définit le type des objets distribués en spécifiant leur interface. L'IDL est donc le moyen par lequel un serveur rend publiques les opérations distantes, et comment elles doivent être invoquées par les éventuels clients. L'OMG a définit les règles de transformation (*Mapping*) d'une interface écrite en IDL dans les langages utilisés pour l'implémentation des objets distants (Java, C, C++, Smalltalk, COBOL, Ada).
- **Le *Stub* (côté client) et le *Skeleton* (côté serveur)**. Ces deux modules sont générés lors de la compilation de l'interface IDL. Ils servent d'interface à la communication entre le client et le serveur.
- **Le *Dynamic Invocation Interface* (DII)**. Cette interface permet l'invocation dynamique des objets distants (au lieu d'utiliser le *Stub*). En utilisant le DII, on construit dynamiquement les requêtes vers des objets CORBA pour lesquels on ne possède pas de *Stub*.
- **Le *Dynamic Skeleton Interface* (DSI)**. C'est l'interface (côté serveur) qui permet l'invocation dynamique des objets distants (au lieu d'utiliser le *Skeleton*). Avec le DSI, on intercepte les requêtes en provenance des clients pour lesquelles il n'existe pas de *Skeleton*.
- **L'*Object Adapter***. C'est l'outil (côté serveur) qui permet l'interaction entre les objets et l'ORB. Il permet l'appel des méthodes pour construire, détruire, activer et déterminer l'état des objets. Il transmet à l'objet les requêtes du ou des clients.
- **L'*Interface Repository* et l'*Implementation Repository***. L'*Interface Repository* est un référentiel contenant des informations sur les objets distants utilisés lors de l'exécution de l'application. L'*Implementation Repository* contient des informations qui permettent à l'ORB de localiser et d'activer les différentes implémentations des objets référencés dans l'*Interface Repository*.

Les principaux avantages à utiliser CORBA sont :

- CORBA supporte plusieurs langages de programmation à l'intérieur d'une même application distribuée.
- Les objets distants peuvent fonctionner dans des processus différents et sur des ordinateurs différents.
- CORBA est basé sur l'orienté-objet et favorise la réutilisation des composantes.
- CORBA est basé sur des standards établis de l'industrie. Ceci amène une compétition saine entre les vendeurs (les fournisseurs de l'ORB CORBA), assurant une qualité d'implémentation du standard. Ce standard assure également un haut degré de portabilité entre les implémentations.

- CORBA fournit un haut degré d'interopérabilité. Ceci a pour effet d'assurer que les objets distribués avec différentes versions de CORBA sont entièrement compatibles.
- Plus de 800 entreprises différentes dans le monde utilisent et supportent CORBA (productrices de logiciels, de *Hardware*, les entreprises de câble, de téléphone, les banques, ...).

CORBA est un sérieux compétiteur à RMI à cause de ses nombreuses fonctionnalités mais aussi parce qu'il est compilé et qu'il est plus rapide d'exécution que les programmes Java.

Les vendeurs suivants offrent une implémentation de CORBA: Inprise (VisiBroker), IONA (ORBIX et ORBIXWeb), IBM (SOM Object), Object Oriented Concepts (ORBacus), Expertsoft (CORBA Plus), ORL (OmniORB2) et l'université de Washington (TAO-ACE).

3.3 DCOM (*Distributed Component Object Model*)

DCOM est un ensemble d'extensions basées sur le *Distributed Computing Environment* RPC (DCE-RPC), qui intègre l'orienté-objet. La figure 5 illustre l'agencement des différentes composantes de DCOM. C'est un *Middleware* qui ressemble à CORBA, à la différence qu'il est fortement adapté (lire dédié) aux produits et plates-formes Microsoft (98/NT/2000). Il est une extension de COM en ce sens qu'il permet de séparer une application en plusieurs parties (composantes ou objets) et de les distribuer sur un réseau.

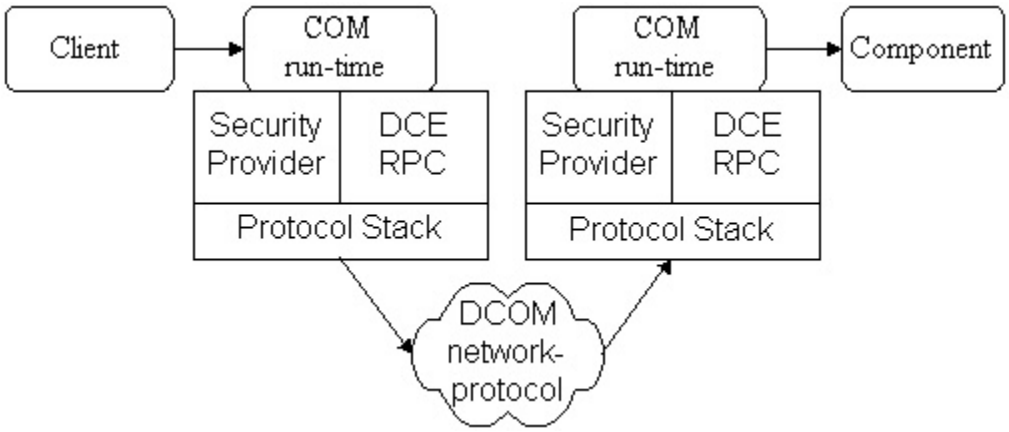

Figure 5. Composantes de DCOM (figure tirée de [12]).

C'est cette technologie qui permet, entre autres, l'utilisation d'une feuille de calcul du logiciel MS-Excel à l'intérieur du logiciel MS-Word et ce, sans que MS-Excel ne soit en fonction. La technologie COM est donc une spécification propriétaire Windows.

Certaines des caractéristiques importantes sont les suivantes:

- Les composantes (objets) DCOM peuvent fonctionner dans des processus différents et sur des ordinateurs différents.
- DCOM favorise la réutilisation des composantes.
- Microsoft affirme que les composantes DCOM sont indépendantes du langage de programmation utilisé.
- Le concept de client avec DCOM: C'est une application qui invoque les méthodes d'une composante DCOM fonctionnant sur un serveur.
- Le concept de serveur DCOM: C'est une application qui rend accessibles des composantes DCOM à des clients.
- Le concept d'interface dans DCOM: C'est un pointeur à un groupe de fonctions qui peut être appelé via DCOM.
- Le concept de classe dans DCOM: C'est la définition d'une abstraction qui implémente une ou plusieurs interfaces.
- Le concept d'objet dans DCOM: C'est une instance d'une classe.
- Le *Marshaling*: C'est une opération qui consiste à transformer et à transférer des données entre le client et le serveur.

Cette spécification impose donc aux compilateurs la façon de générer les fichiers de sorties de telle sorte qu'ils puissent communiquer entre eux et interagir avec le système d'exploitation. Les applications intégrant DCOM peuvent être développées en utilisant les langages et les plates-formes supportés par Microsoft.

C'est une suite logique aux technologies OLE et DLL qui ne permettaient pas la gestion des versions (menant ainsi à des problèmes d'opération) et ne respectaient pas la compatibilité arrière (*Backward Compatibility*). De plus, afin de rencontrer les besoins d'Internet, les composantes COM ont également été remaniées en ActiveX. La gestion de la sécurité, l'amélioration des performances et le design des éléments de données, ajusté aux téléchargements asynchrones, ont été ajoutés.

Cette technologie est basée sur l'orienté-objet et intègre ses concepts:

- **L'identité**. Chaque composante DCOM doit être identifiée de façon unique. On utilise le *Global Unique ID* (GUID) à cette fin.
- **L'état**. Les composantes DCOM doivent effectuer un suivi de leur état.
- **Le comportement**. Les composantes DCOM doivent implémenter les comportements et fonctionnalités pour les clients. L'interface définit les méthodes qui sont accessibles et fournit un pointeur à ce groupe de méthodes.
- **L'encapsulation**. Les attributs et les méthodes ne sont pas tous accessibles aux clients. Les composantes DCOM ne montrent que leurs attributs et méthodes publics.
- **L'héritage**. Une composante DCOM peut être dérivée d'une autre, héritant de ses particularités en plus d'en ajouter de nouvelles.

- **Le Polymorphisme**. Deux composantes DCOM qui implémentent la même interaction peuvent être utilisées "polymorphiquement".

Les applications intégrant la technologie DCOM se divisent en plusieurs modules complémentaires, en voici la liste.

- **Le serveur DCOM**. C'est précisément la partie dans laquelle le programmeur doit concentrer ses efforts. C'est un fichier exécutable (.dll ou .exe) qui contient les éléments DCOM rendus disponibles aux clients.
- **La *Classfactory DCOM***. C'est le service qui retourne un pointeur à l'interface, sur demande du système DCOM. Le client ne voit jamais la *Classfactory*.
- **L'interface**. C'est l'unique partie du système qui est visible du client. L'interface est la spécification des méthodes qu'un serveur fournit pour une composante DCOM. La composante porte un nom spécifique et unique (GUID). Le serveur implémente toutes les méthodes rendues accessibles aux clients. Vue du client, l'interface apparaît comme un objet, son pointeur est lié à un groupe de méthodes. Le serveur ne donne jamais un pointeur direct à une des implémentations.
- **Le *Reference Counting***. Le serveur est chargé en mémoire par le système DCOM. Le client ne peut gérer la mémoire utilisée par le serveur. Le *Reference Counting* est utilisé du côté serveur pour décider quand un serveur n'est plus utilisé et ainsi, l'enlever de la mémoire. C'est un système de comptage de références aux objets distants.
- **L'API DCOM**. Les clients n'ont jamais d'information concernant les .dll et les .exe côté serveur. Windows fournit un API afin de permettre aux clients d'initialiser un service et de demander un pointeur à l'interface d'un serveur en utilisant un GUID.
- **Le client**. C'est la partie du système qui rassemble les autres parties et rend le système opérationnel. Le client initialise le système DCOM, fait les appels à l'API DCOM pour obtenir le pointeur à l'interface et utilise les méthodes supportées par cette interface et implémentées dans le serveur. Lorsque le client en a terminé avec l'interface, il le libère. Lorsqu'il en a terminé avec les services, il ferme l'accès au système.

Le client est donc responsable de provoquer les opérations qui feront fonctionner le système DCOM. Le serveur est passif, il ne fait que répondre aux requêtes du client et ce, d'une manière synchrone. Les étapes du côté client sont donc les suivantes:

1. L'application cliente démarre le serveur.
2. Elle fait ensuite une requête d'accès à l'interface DCOM en spécifiant le GUID. Si le serveur n'est pas en fonction, il est lancé. Côté serveur, la composante DCOM est créée, ensuite l'interface à l'objet DCOM est créée et la *Reference Count* de l'interface est incrémentée.
3. L'application cliente lance ensuite les méthodes distantes de ces composantes DCOM. Du côté serveur, les méthodes sont appelées.

4. Elle délaisse ensuite l'interface du serveur. Ce dernier sera ensuite enlevé de la mémoire de l'ordinateur serveur. Du côté serveur, la *Reference Count* de l'interface est décrémentée. Si elle est égale à 0, la composante DCOM peut être enlevée de la mémoire du serveur. S'il n'y a plus de connexion active (plus de *Ping* périodique du client), le serveur peut être enlevé de la mémoire (se souvenir que la connexion est de type non-connecté).

Les serveurs DCOM sont donc des véhicules pour rendre disponibles des composantes à des clients distants. Comme pour les autres technologies, l'interface est un concept important.

4.0 Exemple d'utilisation de Java RMI

4.1 Description de l'exemple

L'exemple qui a été choisi (le listing apparaît en annexe 2 de ce document) met l'accent sur certains concepts inhérents à Java RMI (JDK 1.2). C'est une application distribuée, de type client-serveur, qui permet à la partie cliente de passer des commandes (faire des requêtes) à la partie serveur en utilisant Java RMI. Dans cet exemple, **une requête** est un objet qui contient plusieurs informations dont le nom de la commande tapée par l'usager et d'autres informations.

Les concepts de RMI que nous désirons faire ressortir par cet exemple sont les suivants:

- Le client et le serveur fonctionnent sur deux machines virtuelles différentes, qui sont localisées sur le même ordinateur (un PC utilisant JDK 1.2).
- Le serveur est une application sans interface graphique qui affiche les accès et requêtes des clients. Plusieurs clients peuvent se connecter simultanément au serveur et passer des requêtes.
- Le serveur lance lui-même le gestionnaire de registres pour l'utilisation du port 1099.
- Le client est un *Applet* offrant à l'usager une interface usager graphique (GUI). L'*Applet* permet la connexion au serveur et le passage de commandes (requêtes) à l'objet distant appartenant au serveur.
- Le serveur ne fait aucun traitement particulier des requêtes, il ne fait qu'enregistrer les commandes des clients dans un fichier `client.log`.
- Le serveur retourne une information (un objet) au client en utilisant une référence à une méthode du client (*Callback*).
- Le client a le loisir de se déconnecter du serveur et de se reconnecter plus tard. Le serveur gère les connexions des clients.

Les figures 6, 7 et 8 illustrent, sous la forme de *Use Case* et de diagrammes d'activités, les différentes tâches des deux modules de l'application. Lorsque le serveur est lancé, il lance le gestionnaire de registres et il enregistre l'objet distant (accessible aux clients). Il attend ensuite la demande de connexion des clients. Lorsque, la connexion est établie avec un ou plusieurs clients, le serveur attend les requêtes en provenance de ces clients.

Lorsqu'un client connecté passe une requête au serveur, le serveur avise le client qu'il a reçu une requête (en retournant un objet au client et en exécutant, par un *Callback*, la méthode `aviser`() du client). Il traite ensuite la requête en en enregistrant la commande de la requête dans le fichier `client.log` (côté serveur).

Pour se connecter, un usager doit entrer son nom et demander la connexion. Il peut ensuite passer autant de requêtes qu'il le désire.

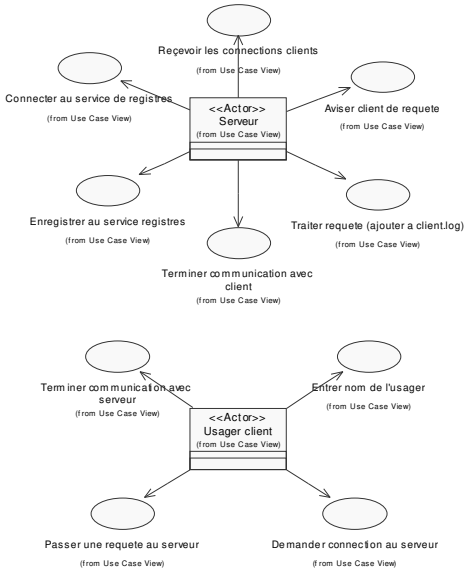

Figure 6. *Use Case* des deux parties de l'application client-serveur.

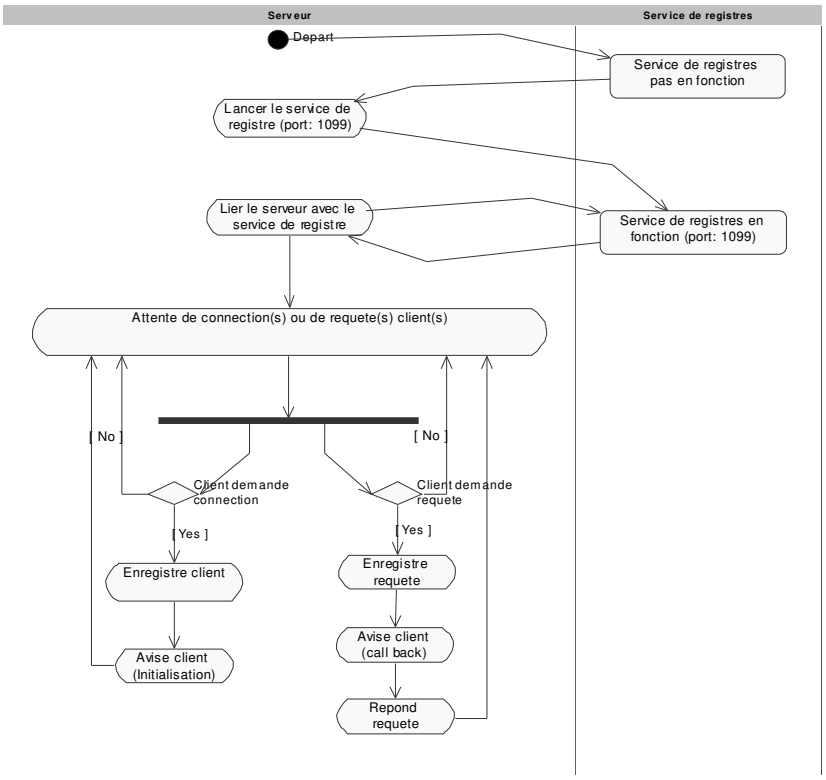

Figure 7. Diagramme d'activités illustrant le fonctionnement du serveur. La déconnexion constitue une requête comme les autres.

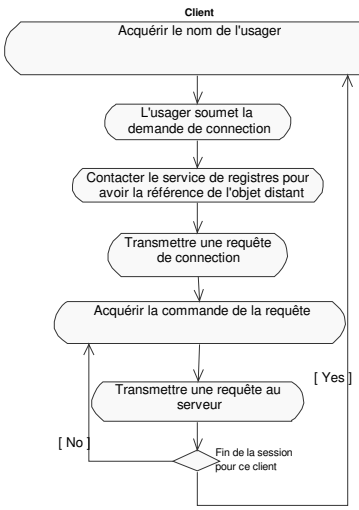

Figure 8. Diagramme d'activités illustrant le fonctionnement du client. L'activité "déconnexion" est manquante au graphique.

4.2 Partie serveur de l'application distribuée

La figure 9 montre la structure de l'application distribuée. La partie serveur de l'application est constituée d'une interface (`Serveur`) qui publie les méthodes qui sont accessibles aux éventuels clients. L'implémentation du serveur est réalisée dans la classe `ServeurImpl`. Un vecteur de cette classe (`personnes`) contient les clients qui sont connectés au serveur. Chaque client qui se connecte au serveur obtient un numéro unique (dans la méthode `enregistrement`()) qui le distingue des autres clients.

La classe `ServeurImpl` possède un lien d'agrégation avec la classe `LienClient`, pour chaque client. La classe `LienClient` a pour fonctions de garder les commandes des requêtes dans un vecteur (`requetes`) en attendant de les traiter. Le *Thread* est en mode pause lorsque le vecteur de requêtes est vide, il est réactivé dans le cas contraire. C'est dans la méthode `run`() que les requêtes sont traitées. Une fois traitée, une requête est éliminée du vecteur.

Lorsque le client exécute la méthode `posterRequete`() du serveur (par RMI), cette dernière identifie qui a lancé la requête (en utilisant le numéro de client) et lance la méthode `ajouterRequete`() de la classe `LienClient`. Une requête est alors placée dans le vecteur de requêtes (`requetes`) de la classe `LienClient` et lorsque la

synchronisation le permet, les requêtes sont traitées. Le *Thread* est en mode pause si le vecteur de requêtes est vide, il est réactivé dans le cas contraire.

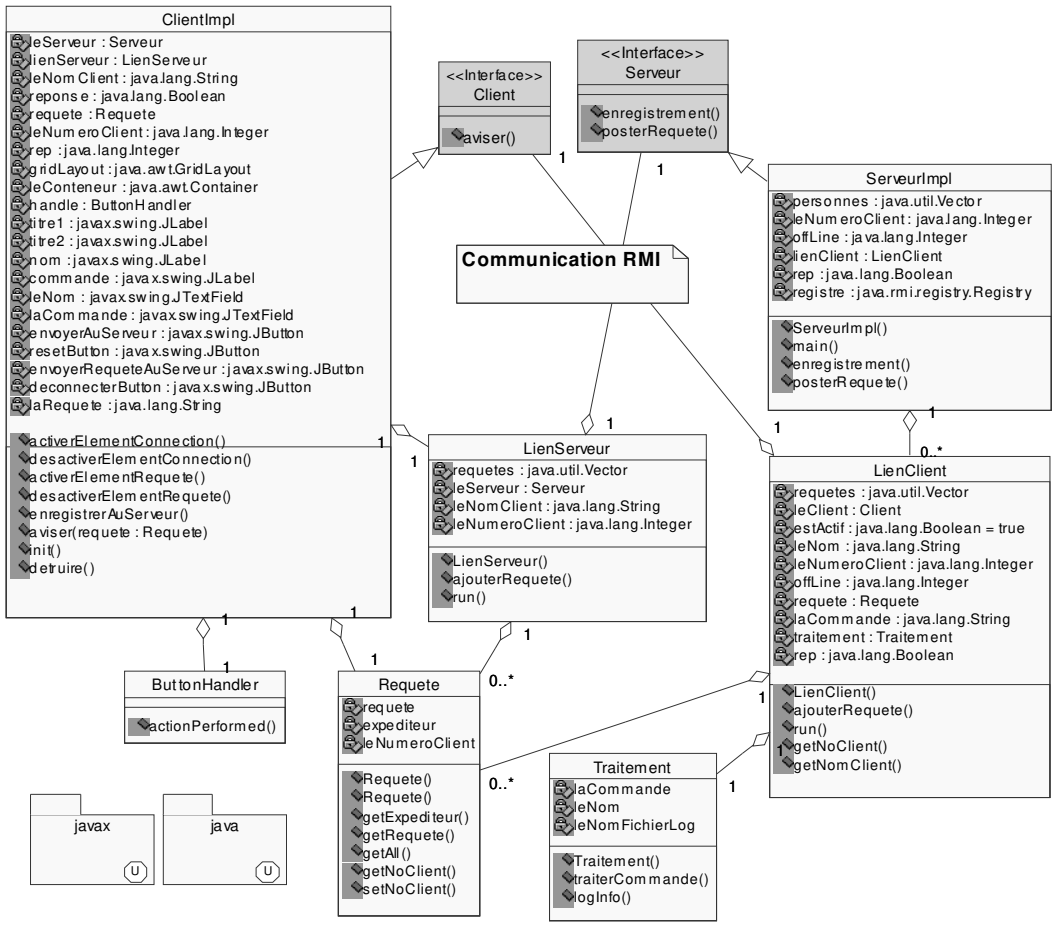

Figure 9. Vue statique de l'application client-serveur (les deux modules).

Le traitement consiste à extraire la commande de la requête. Le serveur avise ensuite le client par un *Callback* (il lance la méthode `aviser`() du client par RMI, en lui passant l'objet `requete`). Le traitement de la requête est ensuite effectué côté serveur (il lance la méthode `traiterCommande`() de la classe `Traitement`). Il élimine ensuite la requête du vecteur de requêtes.

La classe `LienClient` possède un lien d'agrégation avec les classes `Requetes` et `Traitement`. La classe `Requete` contient le nom de la commande de la requête, le nom et le numéro du client qui a effectué la requête. La classe `Traitement` ajoute la commande de la requête (par la méthode `loginfo`()) à la fin du fichier `client.log` (côté serveur).

4.3 Partie cliente de l'application distribuée

L'interface `Client` rend public la méthode `aviser`() afin que le serveur puisse l'utiliser et avertir ce client qu'une requête a été reçue par le serveur. L'implémentation du client se trouve dans la classe `ClientImpl`. Elle hérite de la classe `Applet` et utilise les classes des *Packages* `swing` et `awt`. Elle propose une interface graphique contenant des boutons, les champs permettant l'acquisition de chaînes de caractères (voir la figure 12) et la gestion des événements. La classe imbriquée `ButtonHandler` gère les événements qui sont reliés aux boutons. Les événements reliés aux champs de caractères ne sont pas traités.

Événement traité: bouton "Connecter avec ce nom":

Lorsque l'usager a entré son nom, et qu'il appuie sur le bouton "Connecter avec ce nom", la méthode `enregistrerAuServeur`() est lancée. L'objet client est alors exporté en utilisant la méthode `exportObject`() de la classe RMI `UnicastRemoteObject`. La référence du serveur (et de sont objet exporté) est ensuite obtenue (avec la méthode `naming.lookup`()).

La méthode publique du serveur `enregistrement`() est ensuite lancée avec, comme paramètres, la référence du client et le nom de l'usager. La classe `LienServeur` est ensuite instantiée. Le constructeur de la classe `LienServeur` lance une première requête au serveur (init_requete).

(Réf. 1) La classe `LienServeur` garde les requêtes dans un vecteur (`requetes`) et les transmet au serveur quand la synchronisation le permet. Le *Thread* est en mode pause lorsque le vecteur de requêtes est vide. Il est réactivé dans le cas contraire. C'est la méthode du serveur `posterRequete`() qui est lancée pour transmettre au serveur la requête.

Événement traité: bouton "Envoyer cette requete":

Lorsque la connexion est établie entre le client et le serveur, l'interface graphique désactive le champ et les boutons utilisés pour effectuer la connexion. Ceux du passage de requêtes sont activés.

Lorsque l'usager écrit une commande dans le champ et qu'il appuie sur le bouton "Envoyer cette requete", le contenu du champ est extrait (la commande). La classe `Requete` est ensuite instantiée avec comme paramètres le nom de l'usager, la commande et le numéro du client. La méthode `ajouterRequete`() de la classe `LienServeur` est ensuite lancée avec comme paramètre l'objet `requete`. Le comportement de la classe `LienClient`, décrit plus haut en (Réf. 1), est reconduit.

Événement traité: bouton "Deconnecter":

L'usager a le loisir de se déconnecter du serveur et de se reconnecter plus tard sans que l'`Applet` ne soit détruit. La méthode `détruire`() (côté client) effectue cette déconnexion. Un test identifiant la commande `Exit` dans la méthode `run`() du serveur (`LienClient`) effectue les opérations nécessaires pour enlever le client de la mémoire du serveur.

Il existe donc un lien biunivoque entre le client et le serveur. La communication est entièrement réalisée en utilisant RMI. Pour compléter cette application, il faudrait ajouter un protocole pour mieux contrôler les commandes admises et traitées par le serveur. Si cette application était distribuée sur Internet, il serait impératif d'utiliser les classes de sécurité de Java RMI (l'application ayant été testée localement seulement, nous ne les avons pas incluses).

4.4 Compilation et exécution

La compilation du client et du serveur est relativement simple. Nous utilisons la version 1.2 du compilateur RMI de JDK. Tous les fichiers doivent être dans le même répertoire, l'application n'a pas été divisée en *Package*. Les étapes à la compilation sont les suivantes:

Compilation du serveur et du client:

Dans une fenêtre DOS et dans le répertoire de l'application, lancer le compilateur RMI pour le serveur (ServeurImpl) et le client (ClientImpl).

```
rmi –v1.2 ServeurImpl

rmi –v1.2 ClientImpl
```

Le compilateur compilera tous les fichiers découlant de ces deux implémentations.

Distribution des modules:

Pour faire fonctionner une application RMI, les classes doivent être placées en des endroits visibles par le serveur et les clients.

Du côté serveur, les classes suivantes doivent être accessibles au *Class Loader*.

- L'interface distante définissant les méthodes accessibles aux clients.
- L'implémentation des méthodes accessibles aux clients.
- Toutes les autres classes nécessaires au fonctionnement du serveur.

Du côté client, les classes suivantes doivent être accessibles au *Class Loader*.

- L'interface distante définissant les méthodes accessibles aux clients.
- Le *Stub* pour l'implémentation des classes.
- Toutes les autres classes nécessaires au fonctionnement du client.

Il serait également possible de charger les classes à partir de sites FTP ou de serveur HTTP (en utilisant `java.rmi.RMIClassLoader`). Ceci permettrait le déploiement des classes en un nombre limité d'endroits connus. La façon dont RMI charge les classes est contrôlée par des propriétés qui sont spécifiées lors du lancement de l'application.

```
java [ -D<propertyName>=<Propertyvalue> ] + <ClassFile>
```

Exécution des modules:

Il n'est pas nécessaire de lancer `rmiregistry`, le serveur le lance automatiquement lors de son démarrage. Les étapes pour lancer les applications sont les suivantes:

1- Lancer le serveur en invoquant l'interpréteur Java (figure 10):

```
java ServeurImpl
```

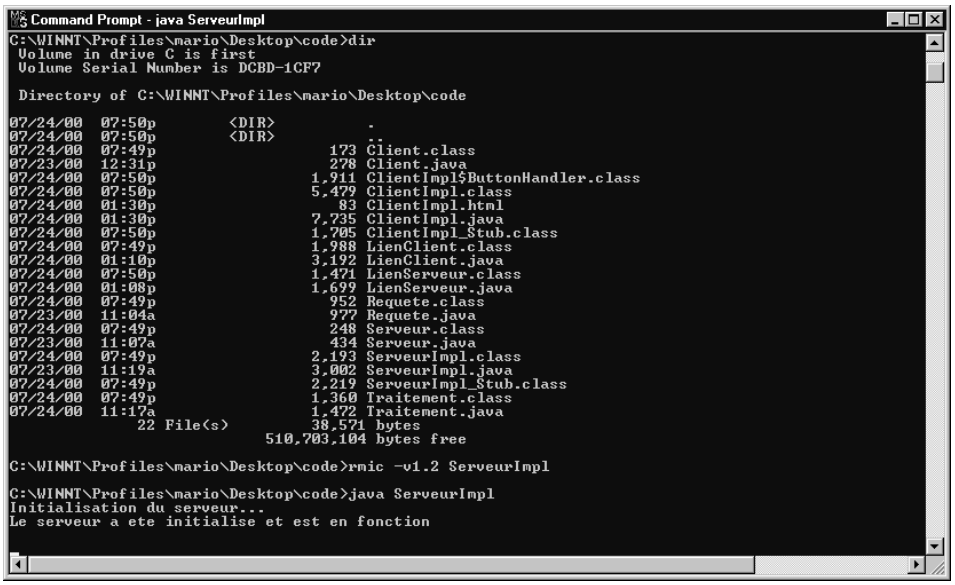

Figure 10. Compilation et lancement du serveur.

2- Lancer l'*Applet* client en invoquant l'interpréteur d'*Applet* (figure 11):

```
appletviewer ClientImpl.html
```

À ce moment, le client (l'`Applet`) n'est pas encore connecté au serveur. L'usager réalise la connexion en inscrivant son nom et en utilisant le bouton approprié. Plusieurs clients peuvent être connectés en même temps et passer des commandes au serveur.

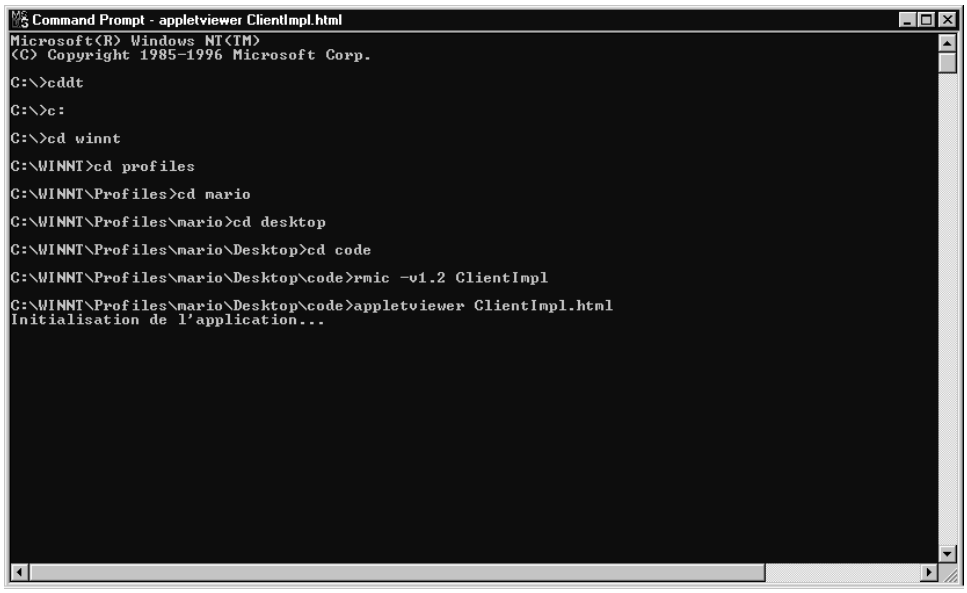

Figure 11. Compilation et lancement de l'*Applet* client.

L'*Applet* apparaît (figure 12) avec seulement trois *Widgets* actifs (le champ pour entrer le nom de la personne, le bouton pour lancer la connexion et le bouton pour effacer le champ).

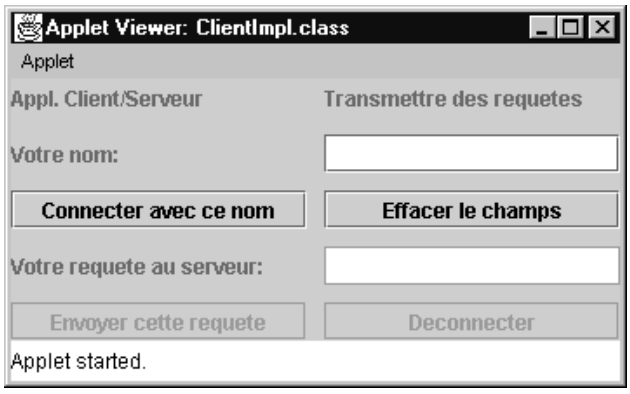

Figure 12. L'*Applet* client avant connexion au client.

Une fois connecté au serveur, les trois *Widgets* de connexion sont désactivés et les trois *Widgets* pour les requêtes sont activés (figure 13). L'usager peut alors entrer autant de commandes qu'il le désire (créant ainsi autant de requêtes).

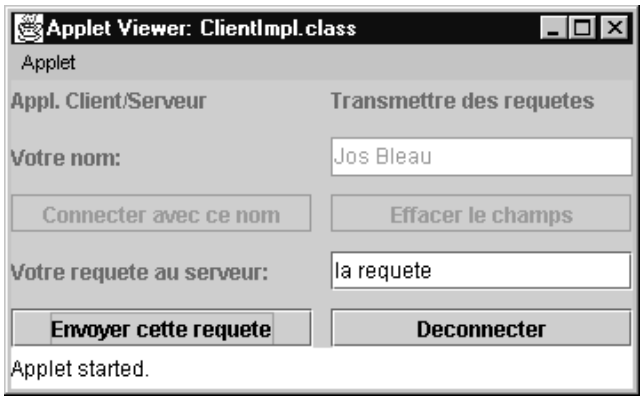

Figure 13. L'*Applet* client après connexion au client et après avoir passé une requête.

Dans la fenêtre DOS du serveur, les informations concernant la connexion du client, son nom et ses requêtes sont affichés (figure 14). Du côté client (figure 15), on peut voir la réponse du serveur aux requêtes du client (le serveur produit un *Callback* en invoquant la méthode `aviser`() du client et en y passant un objet de type Requete).

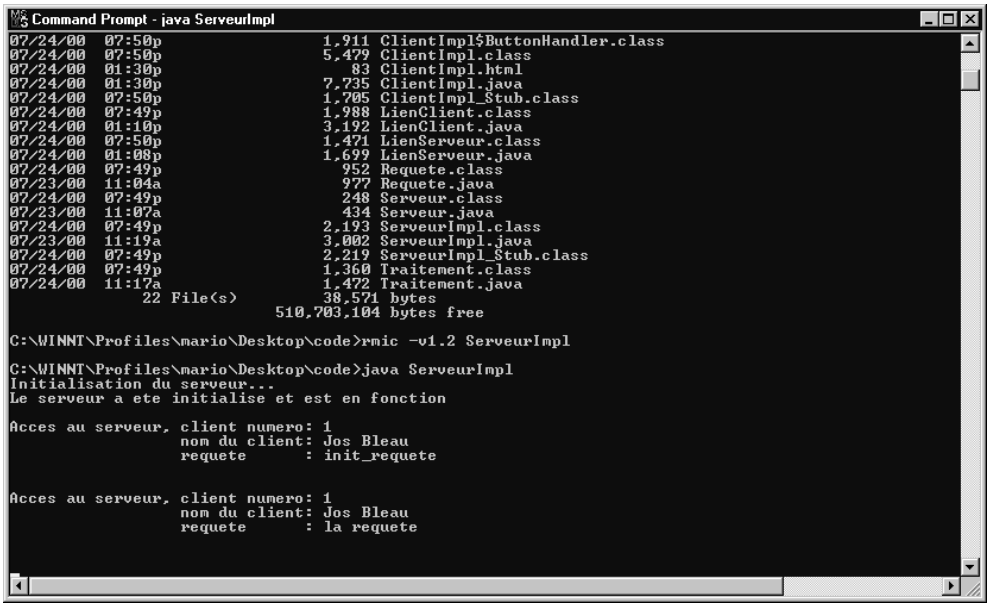

Figure 14. Fenêtre DOS du serveur après le lancement de l'*Applet*, après la connexion du client au serveur et après avoir passé la requête au serveur.

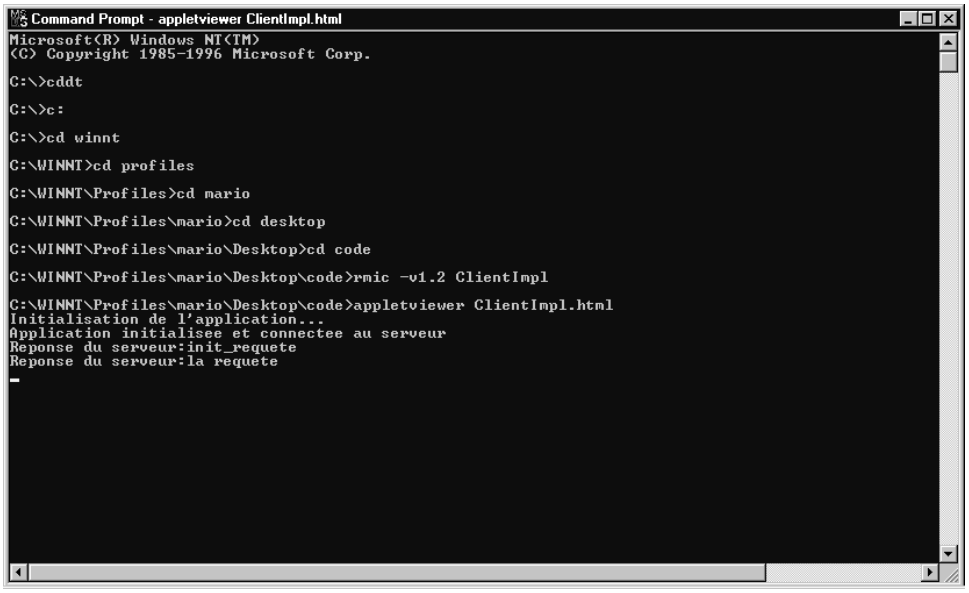

Figure 15. Fenêtre DOS du client après le lancement de l'*Applet*, après la connexion du client au serveur et après avoir passé la requête au serveur.

Le serveur ne traite pas les commandes du client, il ne fait que les ajouter à la fin d'un fichier `client.log` (figure 16). Les informations enregistrées sont la date et l'heure, le nom de l'usager client et la commande de la requête. Le fichier n'est jamais effacé par le serveur.

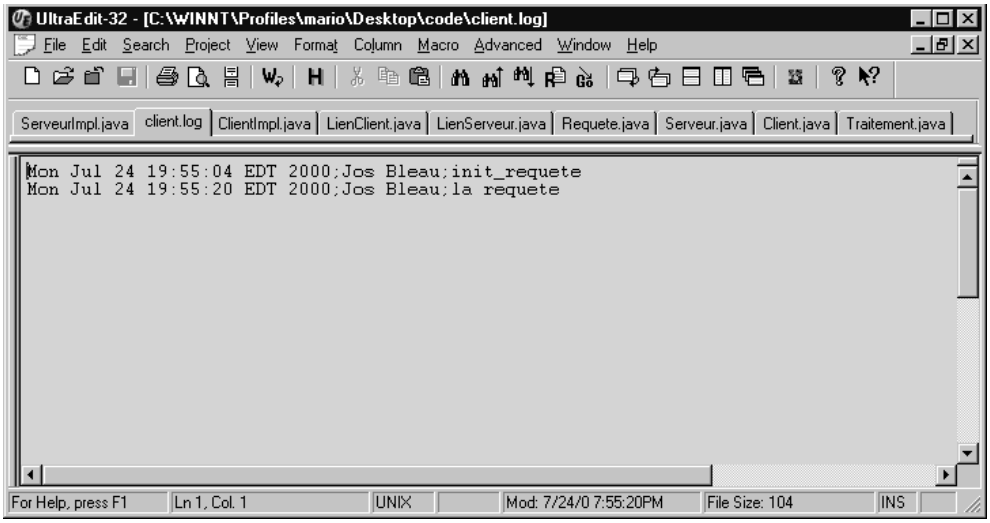

Figure 16. Fichier `client.log` après le lancement de l'*Applet*, après la connexion du client au serveur et après avoir passé la requête au serveur.

Le bouton "Deconnecter" permet à l'usager de se déconnecter du serveur. Le client peut ensuite se reconnecter en utilisant le même nom ou un autre nom. Le numéro du client

sera différent car, à chaque nouvelle connexion est associé un nouveau numéro de client. Dans l'implémentation du serveur, il serait possible de reconnaître un usager par son nom et traiter ses requêtes en conséquences.

Cet exemple est relativement générique car il ne comporte que la base d'une application distribuée. Il pourrait facilement être intégré dans plusieurs domaines d'application en y incluant un protocole de gestion des requêtes (au niveau du serveur) et en y implémentant les objets et méthodes pour traiter ces requêtes convenablement.

5.0 Conclusion

Il serait intéressant de terminer ce document en comparant les différentes technologies orientées objets qui sont disponibles sur le marché. Les questions que l'on se pose sont les suivantes: pourquoi utiliser Java RMI? dans quelles conditions est-il préférable de ne pas l'utiliser? quand doit-on utiliser CORBA ou DCOM?

Pour répondre à ces questions il faut examiner le contexte dans lequel devra opérer tant le serveur que les clients. Les paramètres qui ressortent comme étant importants dans la littérature doivent être évalués ensemble, selon leur priorité. Certains de ces paramètres sont les suivants:

- **Le niveau de performance demandé**. Il est clair que Java RMI est un interpréteur de *Bytecode* et qu'il est relativement lent comparativement à CORBA par exemple (qui est compilé). De plus, RMI permet le transfert d'objets (comprenant toute la structure de l'objet), ce qui peut devenir un facteur important de diminution de la performance. CORBA et DCOM sont compilés et donc, plus efficaces que RMI.
- **La rapidité et la facilité de développement de l'application**. Il est évident que Java RMI est la solution la plus rapide et la plus facile d'utilisation dans le développement d'une application. De plus, la courbe d'apprentissage permet aux programmeurs d'être productifs assez tôt. C'est moins le cas pour CORBA et encore moins pour DCOM.
- **Le coût associé au développement de l'application**. Il faut considérer les produits utilisés (et le temps de développement). RMI est gratuit et est disponible pour téléchargement sur le site de Sun. À notre connaissance, DCOM est aussi gratuit mais il faut considérer l'achat de logiciels satellites (VC++ par exemple). La seule version gratuite de CORBA que nous connaissons est TAO mais elle ne contient pas de système de sécurité. Une étude attentive des licences d'utilisation et de distribution (vente) peut mener à des conclusions différentes.
- **Le besoin indépendance vis à vis les langages de programmation**. Si le langage de programmation n'est pas une contrainte (le serveur et les clients seront écrits dans le même langage) alors il devient intéressant d'utiliser la solution facile et rapide; RMI. Dans le cas contraire, la solution Corba apparaît comme étant la meilleure solution. DCOM peut aussi être considéré.
- **Le besoin indépendance vis à vis les plates-formes et les systèmes d'exploitation**. Ici, CORBA et RMI s'avèrent être de bons choix. DCOM n'est pas un bon choix puisqu'il est est dédié aux plates-formes Windows. Toutefois, il faut mentionner que des passerelles existent entre DCOM et CORBA.
- **Le besoin d'utiliser une technologie basée sur l'orienté-objet**. Les trois solutions répondent à ce critère. Il faut ajouter que Java ne permet pas l'héritage multiple. L'utilisation de CORBA, de RMI ou de DCOM est donc possible.

- **Le besoin de suivre les tendances futures en informatique distribuée**. Il est difficile de faire un choix de *Middleware* en se basant sur ce critère. Il apparaît que chaque groupe de recherche fournit de grands efforts de développements pour faire avancer leurs technologies. Il faut regarder du côté des associations entre groupes de recherche pour trouver un élément de réponse. Sun a intégré les spécifications CORBA dans la version 1.3 de Java, ce qui laisse à penser que Java et CORBA auront un lien de parenté beaucoup plus fort dans les années futures. En fait, il s'avère que Java et CORBA se complètent très bien.
- **Les opérations sur les objets distants**. DCOM et CORBA retournent des références aux objets distants. RMI ajoutes en plus les fonctionnalités suivantes: 1) les API de RMI permettent d'agir sur les objets distants avec des références (*Call By Reference*) et ils permettent également le retour d'objets (*Call By Value*). 2) Si le client n'a pas accès à une classe de laquelle un objet distant a été instantié, les services RMI permettent de télécharger la classe.
- **L'expertise des membres de l'entreprise**. C'est également un facteur important lorsqu'il s'agit d'utiliser une nouvelle technologie qui est relativement complexe.
- **Le type d'application à distribuer et le champ d'application**. Ce sont des facteurs qui peuvent devenir importants lorsqu'il s'agit d'arrimer les nouveaux développements à ceux déjà en place. Dans un domaine d'application donné, il est intéressant et prudent de suivre les tendances du marché. L'idée est de promouvoir et de faciliter les développements et collaborations (internes et externes) futurs. Les passerelles entre technologies ne sont pas nécessairement les meilleures solutions.

Il ressort de la littérature qu'il est nécessaire bien considérer tous les aspects de la problématique et ceux-ci ne se limitent pas seulement aux technologies.

Les tableaux de comparaison (voir par exemple le tableau 1) sont très utiles lors du processus de prise de décision. Ils servent à mettre en évidence les paramètres les plus importants de l'organisation.

Concepts	CORBA (IIOP)	DCOM	RMI
Standard	Oui, créée par l'OMG	Non	Non
Fournisseurs	Privés + universités	Microsoft	Sun
Approche objet	Oui	Oui	Oui
Plates-formes supportées	Toutes ou presque	Windows principalement	Toutes ou presque
Langages utilisables pour la création d'objets	C++. Java, Pascal, COBOL, Ada, Smalltalk	C++, Pascal, VB, VC++	Java et autres avec Jini
Protocole de comm.	IIOP	RPC/DCE étendu	JRMP
Langage de définition d'interface	IDL, standardisé par l'OMG	IDL, créé par Microsoft, dérivé de IDL/DCE	Aucun
Coût du déploiement	Payant sauf si on utilise la version TAO-ACE	Gratuit	Gratuit
Complexité	Moyenne	Moyenne à élevée	Simple
Support des applets	Oui	Oui, limité à J++	Oui
Gestion sécurité	Oui (SSL)	Oui, limité à J++	Oui
Tolérance pannes	Oui	Non	
Répartition charge	Oui	Non	
Annuaire d'objets	Oui, service de noms	Non	`rmiregistry`
Appels statiques et dynamiques	Oui	Oui	Oui (JDK 1.2)
Héritage d'interface	Oui, multiple	Oui, simple	Oui
Identification objets	IOR	GUID	Nom et adresse
Gestion du cycle de vie des objets	Oui, simplifiée et complète	Comptage de référence et ping périodiques	Oui
Activation automat. des objets	Oui (OAD)	Oui (SCM)	Oui
Multithreading	Automatique par un pool de threads	Manuel ou automatique	Oui
Appels asynchrones	Oui	Non	Oui
Marshaling	Direct	Via RPC	Oui

Tableau 1. Comparaison entre CORBA, DCOM et RMI (partiellement tiré de [7] pour CORBA et DCOM).

Bibliographie

[1] K. Arnold, J. Gosling, D. Holmes, "The Java Programming Language," Addison Wesley, 2000, 595 p., ISBN 0-201-70433-1.

[2] T.B. Downing, "Java RMI, Remote Method Invocation," IDG Books Worldwide, 1998, 288 p., ISBN 0-7645-8043-4.

[3] E. R. harold, "Java Network Programming," O'Reilly, 1997, 422 p., ISBN 1-56592-227-1

[4] J. Farley, "Java Distributed Computing," O'Reilly, 1998, 368 p., ISBN 1-56592-206-9

[5] H. M. Deitel, P. J. Deitel, "Java: How to program series," Prentice Hall, 1999, 1355 p., ISBN 0-13-012507-5

[6] N. Wallace, "COM/DCOM Blue Book," Coriolis, 1999, 747 p., ISBN 1-57610-409-5

[7] D. Acreman, G. Moujeard, L. Rousset, "Développer avec Corba en Java et C++," Campus Press, 1999, 404 p., ISBN 2-7440-0626-2

Pour la documentation officielle relative à SDK 1.2:

[8] Java® 2 SDK, Standard Edition, Documentation Version 1.2.1 (disponible avec SDK 1.2)

Pour la documentation relative à RMI, DCOM et CORBA:

[9] Plusieurs documents du site http://developer.java.sun.com

[10] Plusieurs documents du site http://java.sun.com

[11] Quelques documents du site http://www.cs.wustl.edu/~schmidt

[12] Plusieurs documents du site http://msdn.microsoft.com

[13] Quelques documents du site http://www.omg.org

Annexe 1

Interfaces, classes et *exceptions* de Java RMI

Java RMI est divisé en 5 *Packages* différents. Nous avons extrait une partie de la documentation officielle de JDK 1.2 (du site `http://java.sum.com`) et en avons fait un bref résumé que nous présentons ici.

`java.rmi` (depuis JDK 1.1):

Interfaces (*Package*: `java.rmi`)	
`Remote`	L'interface `Remote` est utilisée pour identifier les interfaces dont les méthodes peuvent être invoquées à partir d'une machine virtuelle distante. Tous les objets distants doivent implémenter cette interface. Il n'y a que les méthodes spécifiées dans l'interface `Remote` qui sont accessibles à distante par des clients. Les classes d'implémentation peuvent implémenter autant de `Remote` interfaces que désirées et peuvent hériter d'autres classes.

Classes (*Package*: `java.rmi`)	
`MarshalledObject`	Un objet `MarshalledObject` contient un *Byte Stream* sous la forme *Serialized*. L'objet transféré est *Serialized* et *Deserialized* en utilisant le même procédé (*Marshaling*).
`Naming`	Cette classe contient les méthodes utilisées pour sauvegarder et obtenir les références aux objets distants.
`RMISecurityManager`	La classe `RMISecurityManager` contient un exemple de gestionnaire de sécurité qui peut être utilisé avec RMI. L'utilisation de cette classe est obligatoire si le client et le serveur sont distribués sur le web.

Exceptions (*Package*: `java.rmi`)	
`AccessException`	Provient de l'exécution de certaines méthodes (`bind, rebind, unbind`). Elles indiquent que le client n'a pas la permission d'effectuer l'opération.
`AlreadyBoundException`	Survient lorsqu'on tente d'effectuer un `bind` en utilisant un nom déjà utilisé.
`ConnectException`	Survient lorsqu'une connexion est refusée.
`ConnectIOException`	Survient lorsqu'une erreur I/O survient lors de la connexion avec le serveur.
`MarshalException`	Survient si une erreur est détectée lors du provessus de *Marshalling*.
`NoSuchObjectException`	Survient si la méthode invoquée n'existe pas.
`NotBoundException`	Survient si le client tente d'accéder à un nom d'objet distant n'existant pas.
`RemoteException`	Survient si une erreur est détectée lors de la communication entre le client et le serveur.
`ServerError`	Provient du serveur. Une méthode du serveur *Throws* une *Exception*.
`ServerException`	Provient du serveur. Une méthode du serveur *Throws* une *Exception*.
`StubNotFoundException`	La classe *Stub* n'a pas été trouvée pour un objet distant quand il a été exporté.
`UnexpectedException`	Survient lorsque qu'une *Exception* inconnue arrive au client.
`UnknownHostException`	
`UnmarshalException`	Survient lors du processus de *Un/Marshalling*.

`java.rmi.dgc` (depuis JDK 1.1):

Ce *Package* (*Distributed Garbage Collection*) fournit les classes et les interfaces utilisées pour la collecte de la mémoire morte dans les applications distribuées avec RMI (*Garbage Collection*).

Interfaces (*Package*: `java.rmi.dgc`)	
`DGC`	Le DGC est utilisé par la partie serveur du *Garbage Collection* de l'application distribuée.

Classes (*Package*: `java.rmi.dgc`)	
`Lease`	
`VMID`	VMID est un identificateur unique pour toutes les machines virtuelles.

`java.rmi.registry` (depuis JDK 1.1):
Un serveur enregistre ses objets distants afin qu'ils soient accessibles pour les clients. Un client localise les objets distants avant de les utiliser.

Interfaces (*Package*: `java.rmi.registry`)	
`Registry`	C'est l'interface qui est utilisée pour sauvegarder et retrouver une référence à l'objet distant.

Classes (*Package*: `java.rmi.registry`)	
`LocateRegistry`	Cette classe est utilisée pour obtenir une référence à un *Bootstrap Remote Object Registry*.

`java.rmi.server` (depuis JDK 1.1):

Ce *Package* regroupe les classes et les interfaces permettant le développement de la partie serveur de l'application distribuée RMI.

Interfaces (*Package*: `java.rmi.server`)	
`RemoteRef`	Un `RemoteRef` représente un handle à l'implémentation d'un objet distant.
`RMIClientSocketFactory`	Cette interface est utilisée par RMI pour obtenir les sockets d'un client lors d'un appel RMI.
`RMIFailureHandler`	
`RMIServerSocketFactory`	Cette interface est utilisée par RMI pour obtenir les sockets d'un serveur lors d'un appel RMI.
`ServerRef`	
`Unreferenced`	

Classes (*Package*: `java.rmi.server`)	
`ObjID`	Utilisé pour identifier un objet distant dans une machine virtuelle (numéro unique).
`RemoteObject`	La calsse `RemoteObject` implémente le comportement `java.lang.Object` pour un objet distant.
`RemoteServer`	La classe `RemoteServer` est la super classe pour l'implémentation des objets distants.
`RemoteStub`	La classe `RemoteStub` est la super classe pour le *Stub* côté client.
`RMIClassLoader`	`RMIClassLoader` fournit les méthodes statiques pour charger les classes distantes.
`RMISocketFactory`	
`UID`	
`UnicastRemoteObject`	Cette classe définit l'objet (unique) dont la référence est valide si le serveur est en fonction.

Exceptions (*Package*: `java.rmi.server`)	
`ExportException`	Erreur lors de l'exportation de l'objet.
`ServerCloneException`	
`ServerNotActiveException`	
`SocketSecurityException`	

`java.rmi.activation` (depuis JDK 1.2):

Ce *Package* fournit les interfaces et les classes nécessaires à l'activation d'objets distants.

Interfaces (*Package*: `java.rmi.activation`)	
`ActivationInstantiator`	C'est l'interface responsable de la création d'objets activables.
`ActivationMonitor`	
`ActivationSystem`	Fournit les méthodes pour enregistrer les objets distants activables.
`Activator`	Facilite l'activation d'objets distants.

Classes (*Package*: `java.rmi.activation`)	
`Activatable`	Fournit le support pour les objets distants activables demandant un accès périodique.
`ActivationDesc`	Descripteur contenant l'information nécessaire à l'activation d'un objet distant
`ActivationGroup`	

`ActivationGroupDesc`	
`ActivationGroupDesc.` `CommandEnvironment`	
`ActivationGroupID`	
`ActivationID`	

Exceptions (*Package*: `java.rmi.activation`)	
`ActivateFailedException`	
`ActivationException`	
`UnknownGroupException`	
`UnknownObjectException`	

Annexe 2

Voici le listing des fichiers constituant l'exemple dont il est question au chapitre 4 de ce document.