12, rue de la Houssinière
44322 Nantes

RMI Lite
Documentation développeur

Paul Vaillant
2009-2010

Master 2 - ALMA

Table des matières

Chapitre 1

Introduction

1.1 Objectifs

RMI Lite est une API écrite en pure java. Elle permet de partager et manipuler des objets distants de la même manière que l'API RMI de base fournie par le JDK, toutefois avec un nombre limité de fonctionnalité. L'implémentation ne requiert pas de compilateur externe pour les Stubs qui sont générés à la volés, ainsi que deux mode de communication sont proposés par défaut : java.io et java.nio. L'utilisation de java.rmi est toujours possible mais sera encapsulée dans RMI Lite.

L'avantage est de rendre indépendant l'application du middleware utilisé, en effet java.rmi n'est pas disponible sur toutes les plateformes (par exemple Dalvik).

1.2 Conventions

On considère qu'un appel distant est l'invocation du méthode sur un objet distant en incluant les arguments.

Chaque interface, en fonction de son implémentation, est suffixée par la technologie utilisée ("_RMI" pour java.rmi, "_IO" pour java.io...).

1.3 Audience

Ce dossier se destine au développeur désireux d'utiliser RMI Lite ou d'implémenter de nouvelles fonctionnalités.

1.4 Portée du document

Ce dossier a pour but de compléter la documentation déjà fournie au sein du code.

1.5 Plan

Dans un premier temps nous verrons comment se servir des différents modes disponibles. Ensuite, dans un second temps nous nous attarderons sur l'architecture globale de l'API suivie par les interactions entre les différentes classes. Enfin pour les plus téméraire, nous terminerons sur les extensions possibles à l'implémentation par défaut proposée.

Chapitre 2

How to

2.1 Classes & méthodes

Pour les applications utilisant RMI, le portage vers RMI Lite ne bouleversera pas l'implémentation déjà existante à condition de se limiter aux fonctionnalités disponibles. En effet, on peut faire un parallèle entre les classes de RMI Lite et celles de java.rmi :

- RemoteObjectProvider = java.rmi.server.UnicastRemoteObject
- NamingServer = java.rmi.registry.LocateRegistry
- Registry = java.rmi.registry.Registry

Une fois les correspondances connues, les noms des méthodes restent sensiblement les mêmes... A noter qu'il n'est pas possible de créer automatiquement un stub en héritant de la classe RemoteObjectProvider comme avec UnicastRemoteObject en java.rmi. La méthode exportObject est obligatoire.

2.2 Configuration

Par défaut, deux configuration sont proposées au sein de l'API :

- ConfigManager_RMI : encapsule java.rmi.
- ConfigManager_Socket : utilise RMI Lite avec java.io pour la communication.

Il est aussi possible de configurer le middleware :

```
Manager_IO io = new Manager_IO(); // couche de communication
NamingServer_Socket ns;
ns = new NamingServer_Socket();
RemoteObjectProvider_Socket rop;
rop = new RemoteObjectProvider_Socket();

rop.setIOManager(io);
io.setRemoteObjectManager(rop);
```

```
ns.setRemoteObjectProvider(rop);

RemoteMethodFactory.remoteObjectManager = rop; // usine d'appel distant
StubFactory.ioManager = io; // usine de stubs
```

Chapitre 3

Architecture

Voici le diagramme de classe général, celui-ci ne représente que le cadre de l'API :

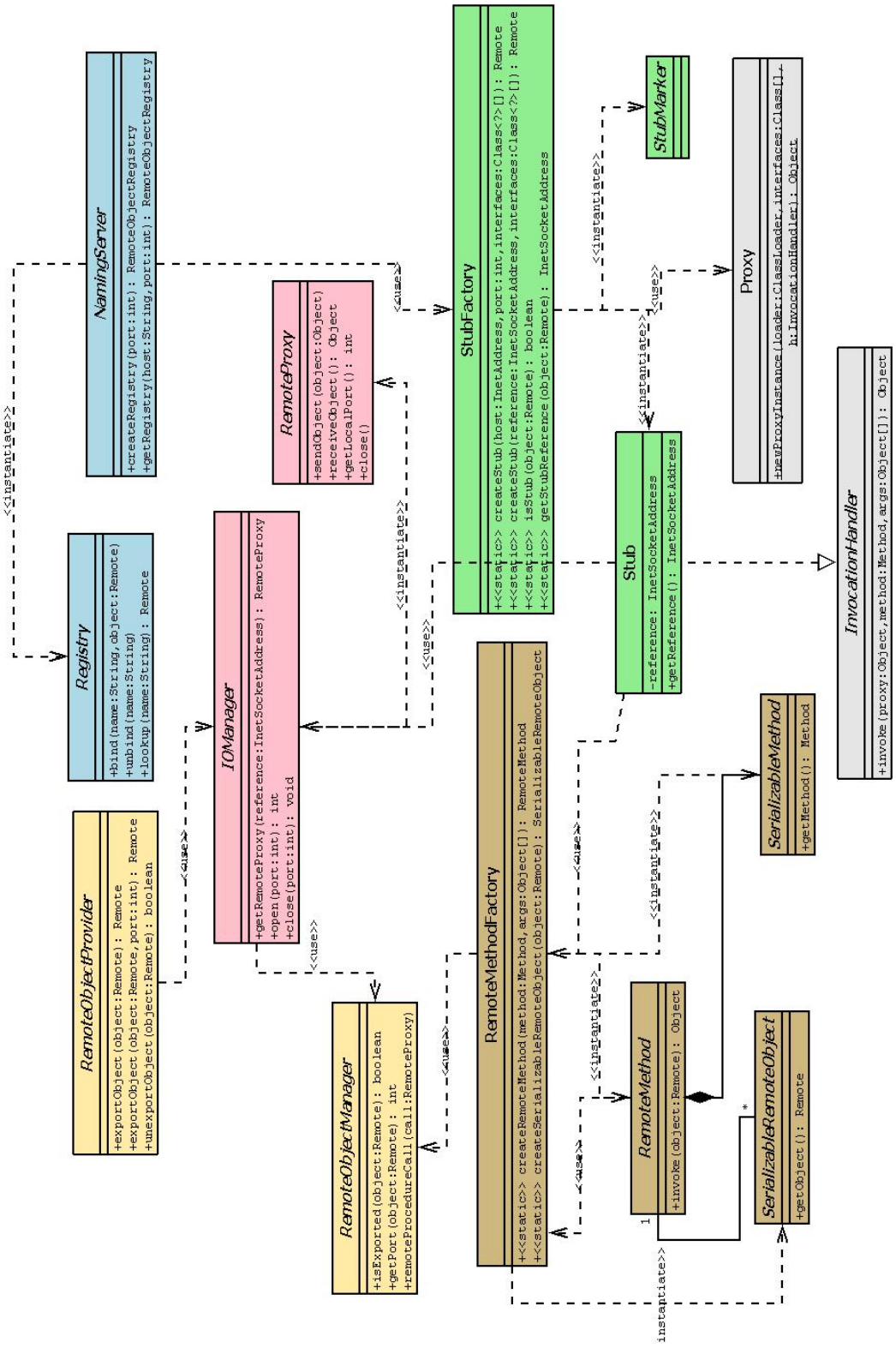

FIG. 3.1 – Diagramme UML de classes

On peut distinguer clairement 5 parties différentes :
- Serveur de nom
- Partage d'objets
- Couche réseau
- Appels distants
- Stub

On identifie un objet partagé sur le réseau grâce à son adresse IP et le port (InetSocketAddress) sur lequel il reçoit les appels distants.

3.1 Serveur de nom

Le fournisseur de registre exporte en même temps que de créer un registre distant. De même, il créé un stub quand on lui en demande un.

3.2 Partage d'objets

L'interface RemoteObjectManager est interne à l'API, elle n'est pas destinée à être utilisée par l'utilisateur lambda. Elle permet à l'usine d'appels distants de vérifier si les objets manipulés en paramètre ou en retour sont bien exportés, et si c'est le cas, d'obtenir leur port d'écoute. Elle permet aussi à la couche de communication de faire remonter un appel distant vers l'objet partagé.

3.3 Couche réseau

L'interface RemoteProxy est destinée à encapsuler un socket, fournit par un IOManager qui se charge d'ouvrir, d'écouter et de fermer des ports.

3.4 Stub

Grâce à l'API de réflexion java.reflect, il est possible de créer des stubs dynamiquement.

3.5 Appels distants

A noter qu'une instance de RemoteMethod doit être sérialisable et contenir tout les paramètres nécessaires à l'éxécution de la méthode distante (ou appel distant) sur l'objet partagé spécifié en paramètre.

Chapitre 4

Interactions

Les diagrammes de séquences suivants ne reflètent que l'idée générale des algorithmes employés dans l'implémentation par défaut... Le scénario suivant montre un client qui accède à un objet distant via un registre connecté à un serveur de nom.

4.1 Côté client

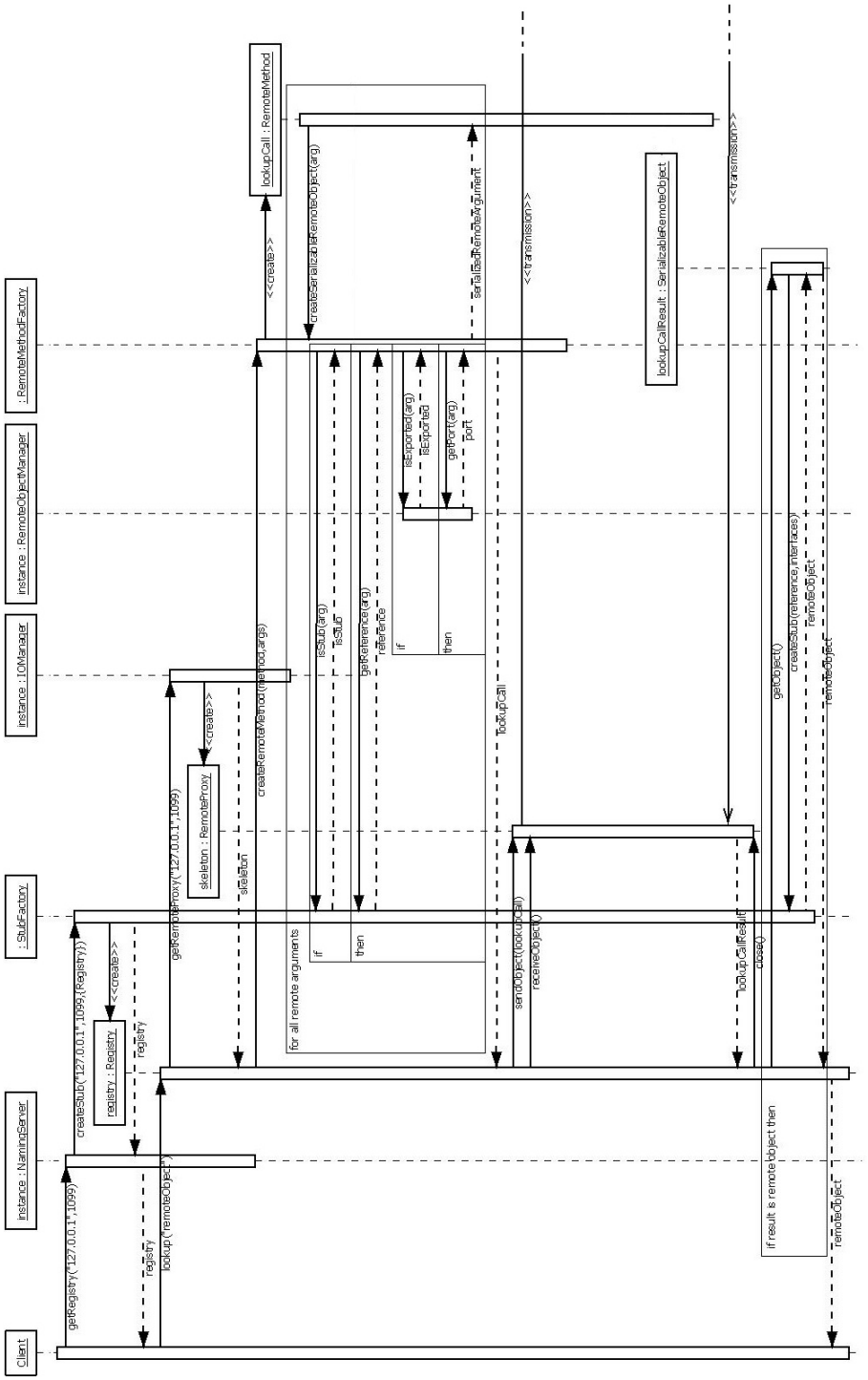

FIG. 4.1 – Diagramme de séquence

Le client récupère d'abord un stub connecté au registre (serveur de nom) distant, puis récupère un objet du serveur.

4.2 Côté serveur

Ici la méthode de createSerializableRemoteObject est simplifiée afin de rendre le diagramme plus lisible (cf. client).

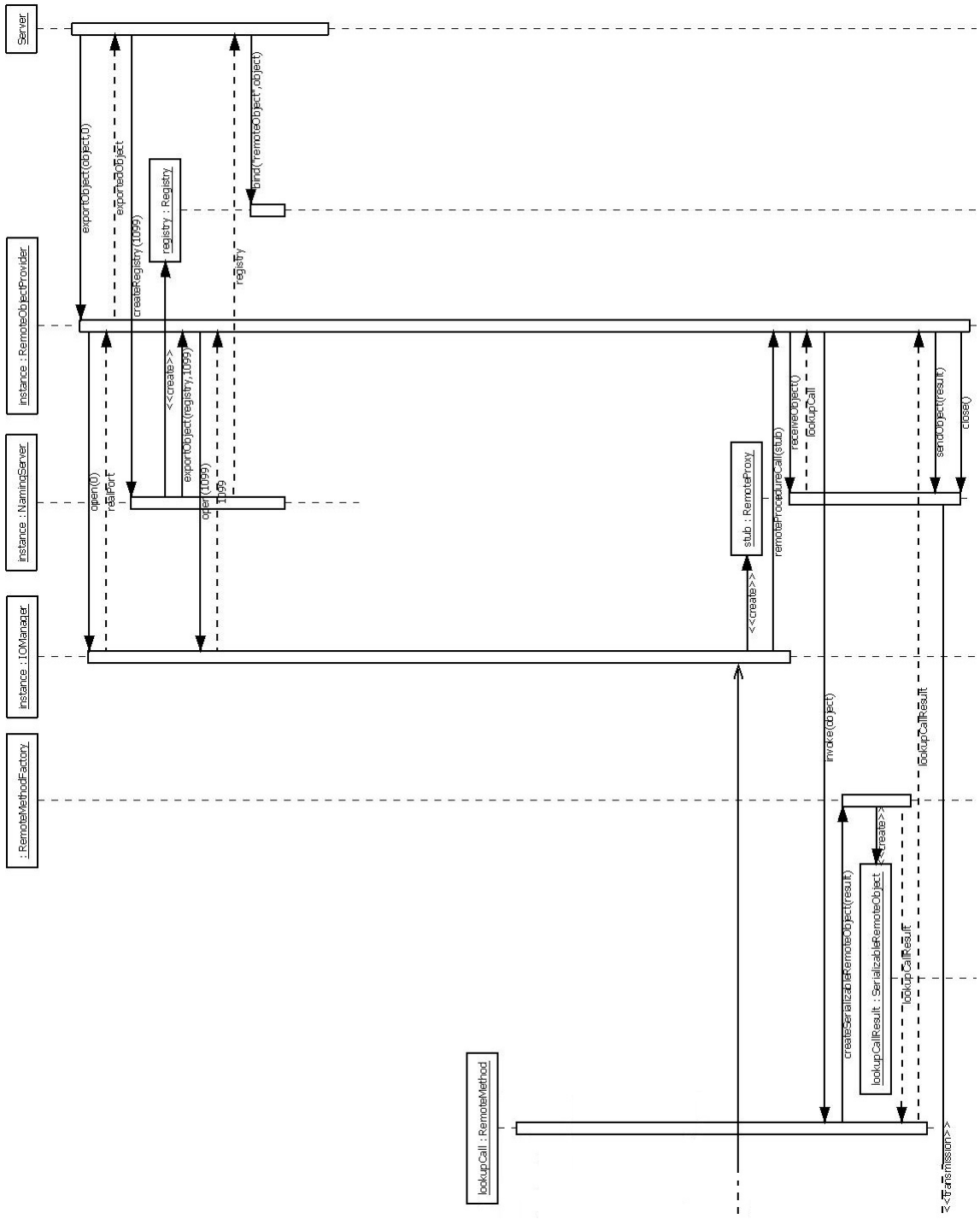

FIG. 4.2 – Diagramme de séquence

Chapitre 5

Extensions

La couche de communication actuelle n'offrant aucune sécurité, le plus évidant serait de commencer à implémenter une couche de communication cryptée en implémentant les classes IOManager et RemoteProxy.

Table des figures