12, rue de la Houssinière
44322 Nantes

RMI Lite
Developer documentation

Paul Vaillant
Brendan Le Ny
2009-2010

Master 2 - ALMA

Contents

1 Introduction **2**
1.1 Objectives . . . 2
1.2 Conventions . . . 2
1.3 Target audience . . . 2
1.4 Scope of document . . . 3
1.5 Organization of this document . . . 3

2 How to **4**
2.1 Classes and methods . . . 4
2.2 Configuration . . . 4

3 Architecture **6**
3.1 Naming server . . . 8
3.2 Object sharing . . . 8
3.3 Communication layer . . . 8
3.4 Stub . . . 8
3.5 Distant calls . . . 8

4 Interactions **9**
4.1 Client . . . 10
4.2 Server . . . 11

5 Extensions **13**

Chapter 1

Introduction

1.1 Objectives

RMILite is an API entirely written in pure Java. RMILite do not replace RMI, you can write your code using RMILite and then, make your application use RMI or our specific protocol depending of what is possible. It is similar to standard RMI available in the standard JDK but has a limited however sufficient features (think RMI Light). This implementation don't require any external compilation tool (like rmic) : stubs are generated on-the-fly. Sockets are used to permit distant communication.

By using RMILite, you can make distant call. The distant call, depending how you use configure RMILite, will be made using RMI, our new protocol based on socket programming implemented by IO or this same protocol, implemented with New IO (better performance).

1.2 Conventions

We consider that a distant call is a method call on a distant object (an instance on a different VM), this call may have arguments. Every interface may be implemented by different classes : those classes have their name following this pattern : nameOfTheInterface followed by the technology used ("_RMI", "_IO" ...).

1.3 Target audience

This document will help users and future developers of RMILite.

1.4 Scope of document

This file is intended to supplement the information already provided in the code.

1.5 Organization of this document

Fist, you will learn how to use the different available modes. Secondly, we will see the global architecture of the project and the interactions between the different classes. Finally, we will see how the proposed implementation can be extended or improved.

Chapter 2

How to

2.1 Classes and methods

For the applications already using RMI, migrating from RMI to RMILite should not raise problems if your use of RMI is limited to the features available in RMILite. See this equivalence between RMILite classes and RMI classes:

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

Once you know those classes, methods remain the same. Note that it's not possible to automatically create a stub by inherit by RemoteObjectProvider just like UnicastRemoteObject. The method exportObject is required.

2.2 Configuration

By default, two setups are proposed in the API:

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

You can configure the middleware too :

```
Manager_IO io = new Manager_IO(); // communication layer
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; // distant call factory
StubFactory.ioManager = io; // stub factory
```

Chapter 3

Architecture

Here is the main class diagram, it only represents the frame of the API:

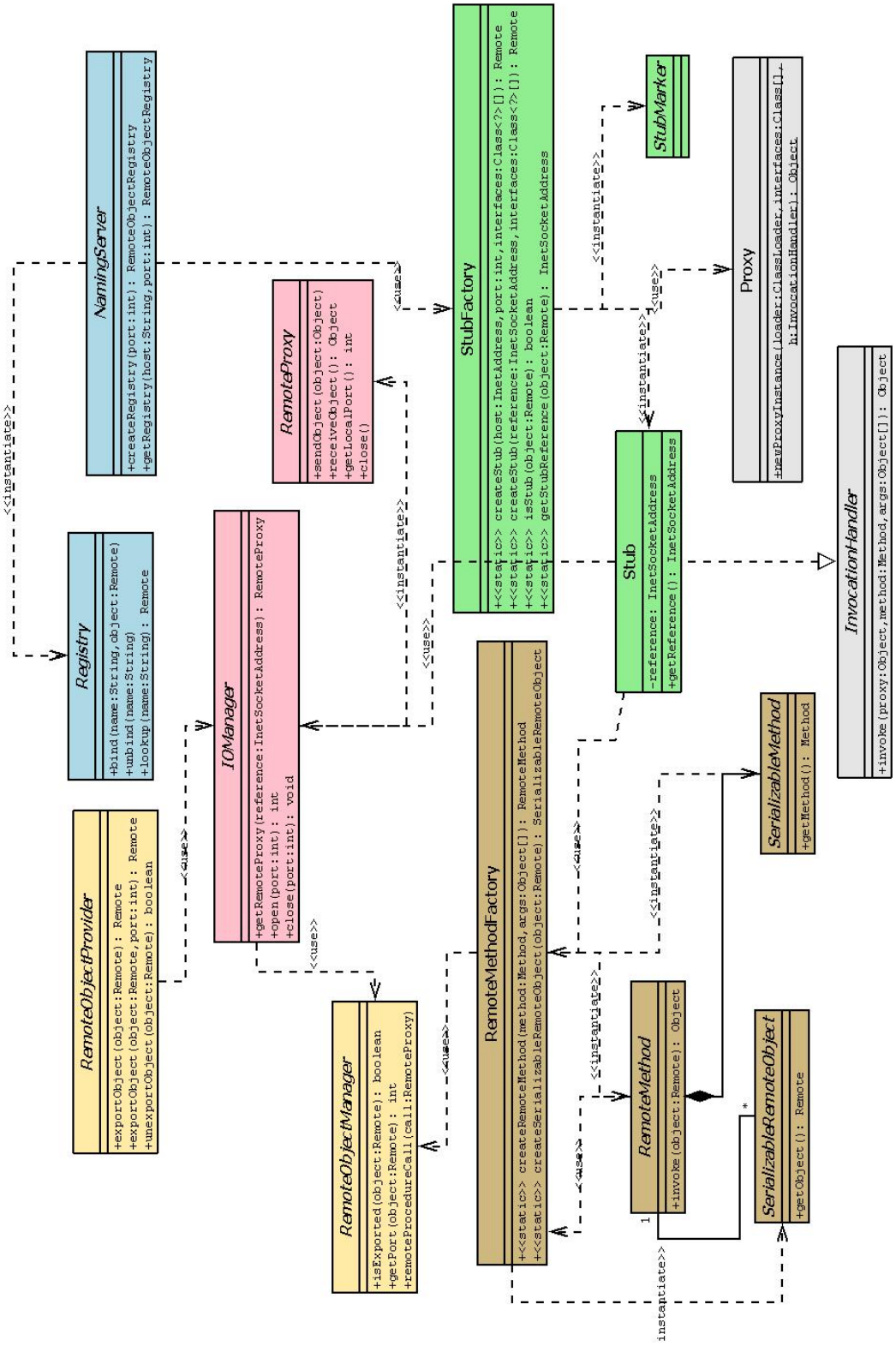

Figure 3.1: UML class diagram

You can see clearly five different parts:

- Naming server
- Object sharing
- Communication layer
- Distant calls
- Stub

We identify an object shared on the network by an IP address and the port used to receive distant calls.

3.1 Naming server

The registry provider provide a stub when asked to a distant register.

3.2 Object sharing

The RemoteObjectManager interface is inner the API, it should not be used by the end-user. It permit the distant call factory to check if the object named in the parameters or returned value are exported, and if it's not the case, to obtain the port which is listened. It allows to the communication layer to pass a distant call to a shared object.

3.3 Communication layer

The RemoteProxy interface is encapsulate a socket, provided by an IOManager which has to open, start listen to, and close ports.

3.4 Stub

Thanks to the Java.reflect API, stubs are created dynamically, on-the-fly.

3.5 Distant calls

A RemoteMethod instance has to be serializable and has to contain all the parameters needed to the execution of the distant method on the shared object used in parameter.

Chapter 4

Interactions

Following sequence diagrams only show the general idea of the algorithms used in the default implementation. This scenario show a client that access a distant object via a register connected to a naming server.

4.1 Client

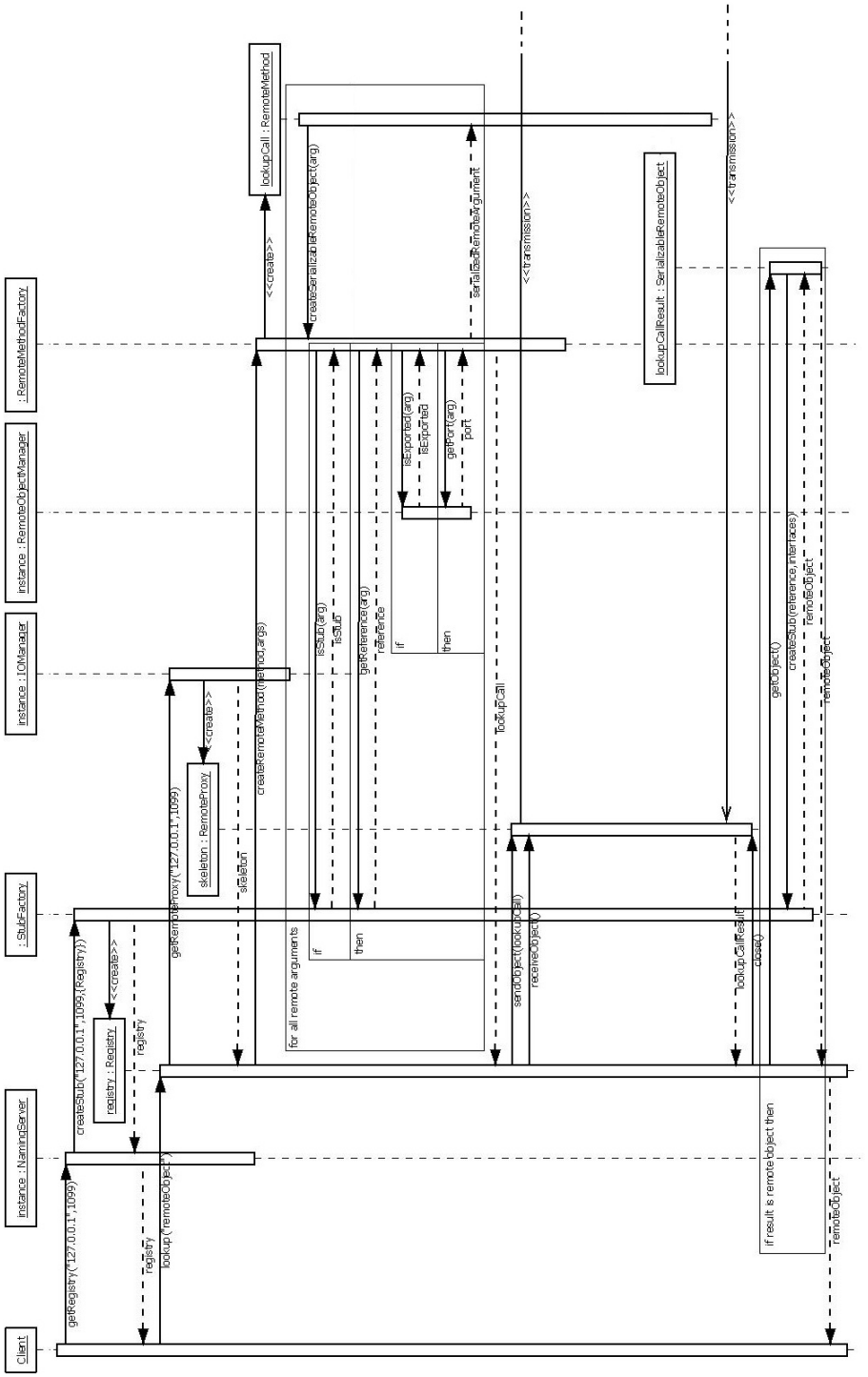

Figure 4.1: Sequence diagram

The client gets a first stub connected to the remote registry (name server), and then retrieves an object from the server.

4.2 Server

Here, the method of createSerializableRemoteObject is simplified in order to make the diagram more readable.

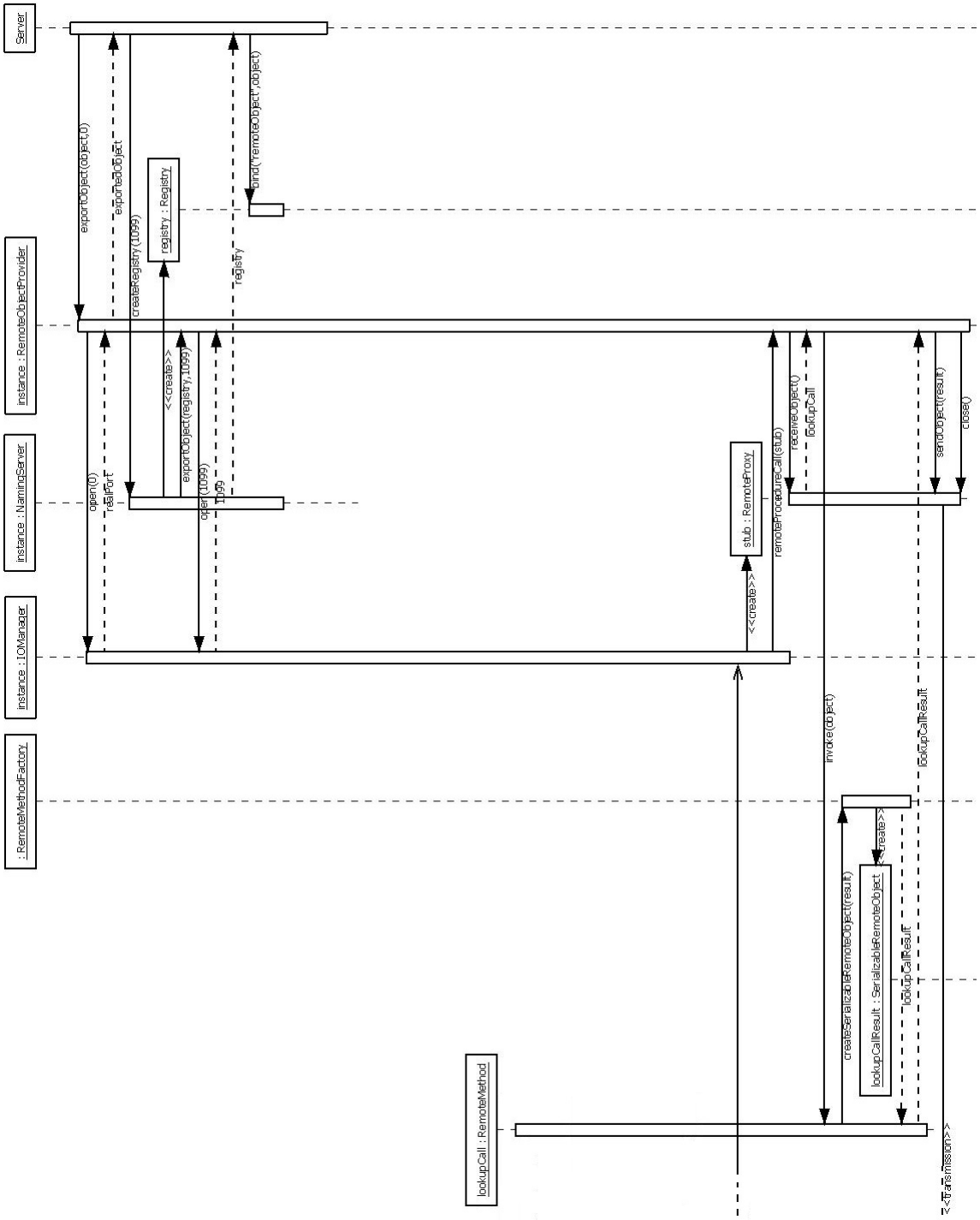

Figure 4.2: Sequence diagram

Chapter 5

Extensions

The current implementation does not consider any security issue, the use of an encrypted communication layer should be considered.

List of Figures

3.1 UML class diagram 7

4.1 Sequence diagram 10
4.2 Sequence diagram 12