TP1 - Appels de méthodes à distance

Loïc Poncet (1973621) Baptiste Rigondaud (1973586)

October 2, 2018

1 Partie 1

1.1 Question 1

La fonction que nous avons utilisée afin de créer un objet de taille 10^x octets est présentée ci-dessous. Il s'agit simplement de créer un tableau de "**byte**" (un **byte** ayant une taille d'un octet en Java) avec 10^x éléments.

```
/**
 * Generates a byte array of size 10^bytesPower.
 * @param bytesPower A power between 1 and 7.
 * @return A 10^bytesPower sized byte array.
 */
private byte[] generateBytes(int bytesPower) {
        int bytesLength = (int) Math.pow(10, bytesPower);
        byte[] byteArray = new byte[bytesLength];
        for(int i = 0; i < bytesLength; i++) {
                byteArray[i] = 80;
        }
        return byteArray;
}
```

Ainsi, la méthode d'exécution possède la signature suivante :

```
int execute(byte[] param) throws RemoteException;
```

Le tableau 1.1 présente les temps d'appels aux différentes méthodes en fonction de la taille des arguments.

La figure 1.1 représente les données des temps d'appels.

Nous pouvons voir sur la figure 1.1 que pour une taille d'argument supérieure à 10^6 octets et plus, le temps écoulé lors de l'appel RMI distant devient plus important que lors d'un appel normal, ou bien un appel RMI local qui eux, semblent conserver des temps d'exécution équivalents. Mais alors, comment expliquer ce comportement? La figure 2 présente l'architecture simplifiée d'un progamme utilisant l'API Java RMI. [1]

Taille des arguments (en octets)	Appel normal (en ns)	Appel RMI local (en ns)	Appel RMI distant (en ns)
10^1	71059	1361651	1418287
10^2	71387	1363770	1559736
10^3	74643	1300269	1708558
10^4	218932	1323662	2367643
10^5	1391098	1759514	10969500
10^6	4296573	5760072	87883959
10^7	8650327	18608809	873567223

Table 1: Coût des appels de méthodes en fonction de la taille des arguments

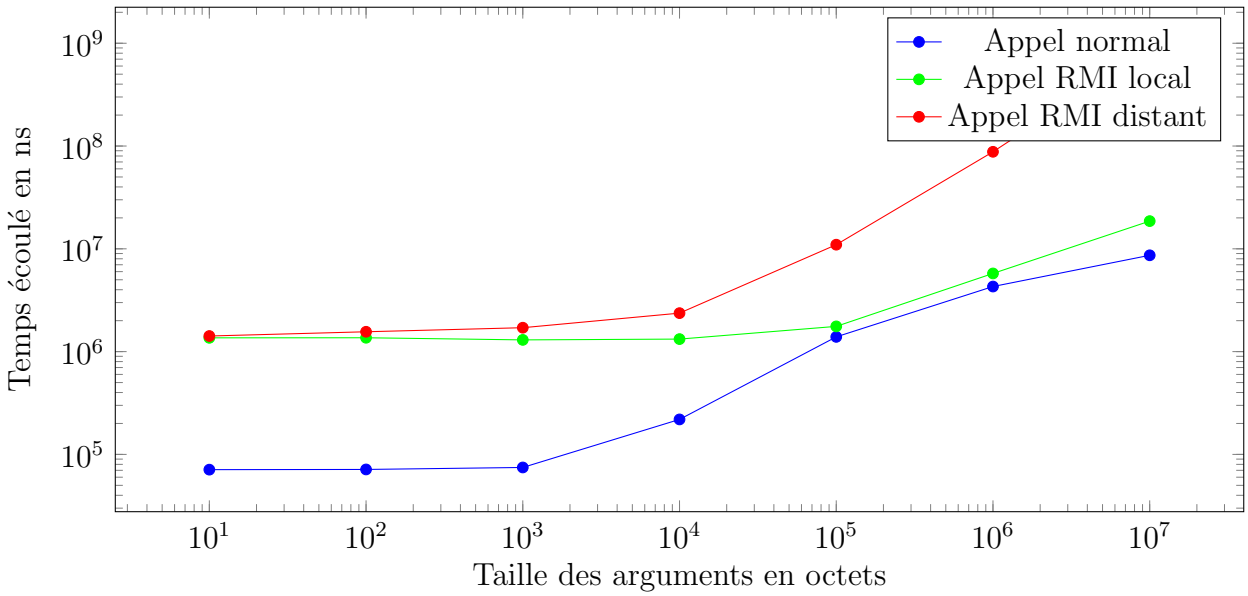

Figure 1: Graphes des temps d'appels

En utilisant RMI, le serveur va pouvoir rendre accessible aux appels distants plusieurs méthodes en liant un objet, qui aura pour but de réaliser les appels de méthodes côté serveur, à un nom spécifique au sein de l'objet que l'on appelle "RMI registry". Le client pourra alors récupérer une référence locale (que l'on appelle "stub") à cet objet distant via le "RMI registry", et effectuer des appels de méthodes RMI via cette instance. Lors d'un appel de méthode, le "stub" va se charger de transmettre la requête à l'objet distant en utilisant la couche transport et le protocole TCP/IP. Une fois la requête reçue, l'objet distant va appeler la méthode adéquate sur le serveur et va transmettre le résultat au client en empruntant le chemin inverse. Ainsi, pour chaque appel, les arguments passés lors de l'invocation de la méthode ainsi que la valeur de retour de cette dernière doivent être sérialisés et envoyés entre le client et le serveur. Cela explique les résultats obtenus dans le tableau 1.1, en effet, le temps de transmission de la requête augmente avec la taille des paramètres et le débit lors d'un appel distant est bien moins important que pour un appel local. De plus, lorsque l'on

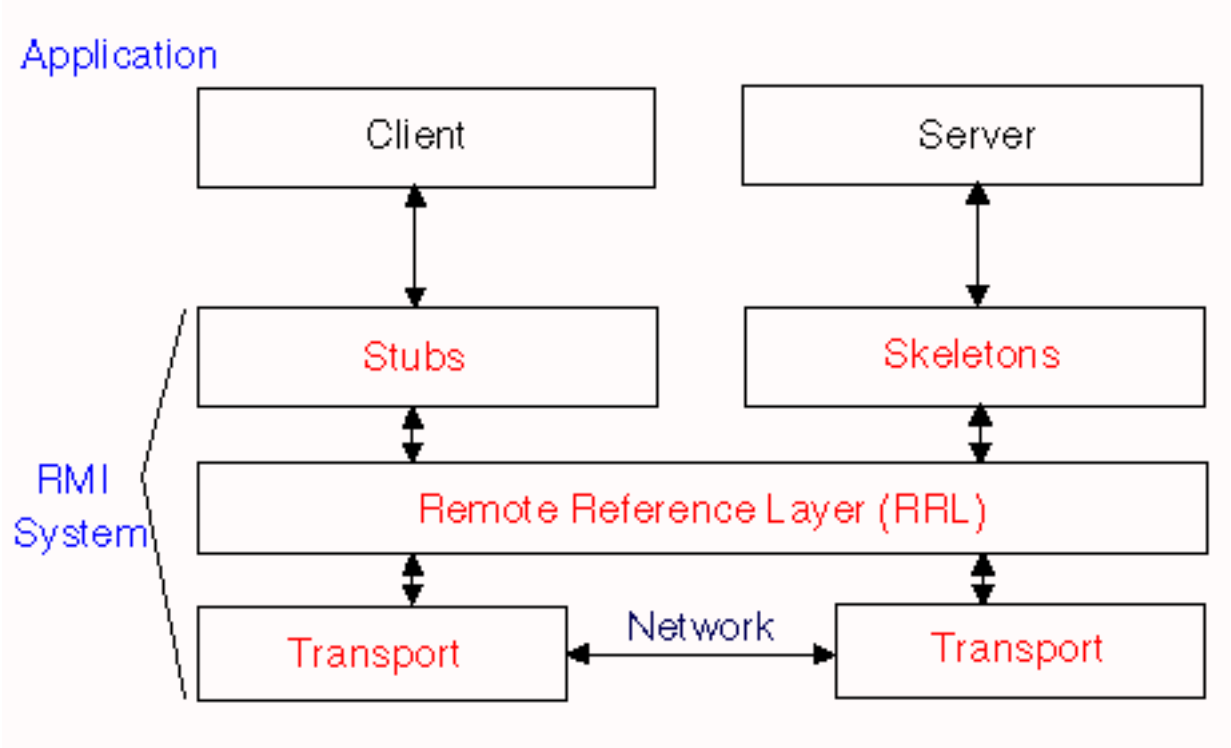

Figure 2: Architecture d'une application utilisant Java RMI [1]

travaille localement, le temps de propagation est négligeable puisque la requête n'a aucun chemin a parcourir contrairement à un appel distant.

L'avantage majeur de l'API RMI est de pouvoir bénéficier de la puissance de calcul d'une machine afin d'effectuer du travail à distance, puis d'en récupérer le résultat une fois celui-ci terminé. Néanmoins, si la latence de la requête du client vers le serveur prend le pas sur le temps nécessaire à l'exécution de la méthode, alors RMI devient désavantageux par rapport à une exécution locale. Le programme étudié dans ce TP en est un bon exemple.

1.2 Question 2

L'architecture globale de l'API RMI a déja été discutée dans la première question, voyons cela en détail avec l'exemple de l'appel RMI du TP. La première étape est de créer un registre RMI qui permettra de lier des objets distants à un certain nom, les rendant ainsi disponibles au client. Une fois ceci fait, la classe `Server`, qui implémente l'interface `ServerInterface`, attache sa propre référence à un objet distant dans le registre RMI en récupérant tout d'abord une référence au registre avec la méthode statique `getRegistry` du package `LocateRegistry` puis en utilisant la méthode `rebind` sur l'objet `registry` récupéré. Cela rendant ainsi l'ensemble de ses méthodes accessibles via des appels RMI. La classe `Client` va pouvoir récupérer une copie locale vers cet objet distant en utilisant le registre RMI et le nom associé à la classe `Server` dans ce dernier. Cette copie locale, nommée `stub` dans le programme, va en réalité agir comme un proxy entre le client et l'objet distant, en exposant l'interface

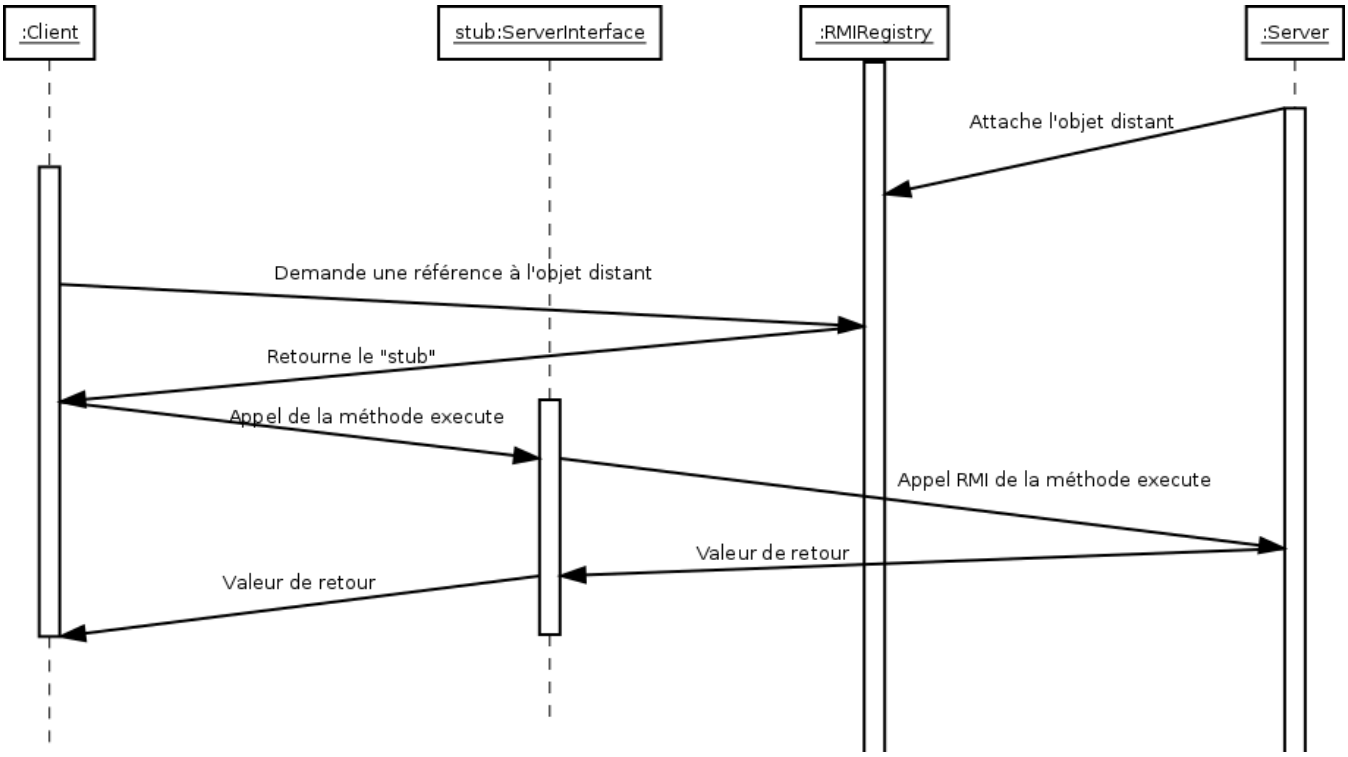

Figure 3: Diagramme de séquence d'un appel RMI

au client et en relayant les appels vers l'objet distant qui s'occupera d'exécuter les méthodes adéquates du côté du serveur. Ainsi lorsque la méthode `execute` sera appelée dans `Client`, le stub se chargera de réaliser une analyse du type des paramètres de la méthode pour vérifier que ceux-ci correspondent bien à ce qui est attendu et, le cas échéant, enverra la requête (contenant la méthode appelée ainsi que ses paramètres sérialisés) à l'objet distant. A récéption, l'objet distant exécutera enfin la méthode et retournera le résultat (qui doit donc également sérialisé) vers le stub, qui le relaiera à son tour au client.

La figure 3 présente un diagramme de séquence d'un appel RMI au sein du système du TP.

References

[1] Harald Kosch. Introduction to java remote method invocation (rmi), feb 2002. http://www-itec.uni-klu.ac.at/ harald/ds2001/rmi/rmi.html.