RAPPORT DU TP1 150237 & 1605802

Question 1 :
Explication de l'expérience mené :
Le but de cette expérience est d'observé le temps de réponse d'appel a des fonctions dans un programme java d'après différentes méthodes d'appels :
Appel normal : On appel la fonction désiré comme on appellerais une fonction d'un programme unique. La fonction appelé est situe dans une application s'exécutant sur le même système que l'application appelant la fonction. Pour cela on exécute le client et un faux serveur (contenant uniquement la fonction que l'on veux appeler) sur la même machine.
Appel RMI local : On appel la fonction a travers l'API de java RMI, mais la fonction est toujours exécuté a travers un programme s'exécutant sur la même machine que l'application appelant la fonction. Afin de réaliser ces appels on exécute un serveur mais sur la même machine que le client qui appelleras la fonction.
Appel RMI distant : On appel la fonction a travers l'API java RMI, a travers le réseaux. La fonction appelé se trouve donc sur une machine distante. On lance donc le serveur contenant la fonction sur un nuage.
Afin de tester la rapidité des appels on utilise une fonction qui n'exécute rien mais qui reçoit en paramètre un tableau de byte. On fait varier ce paramètre afin d'augmenter la taille du paramètre que l'on fait passer a la fonction.
Discussion des résultats obtenus :
En observant le graphique obtenue on remarque que :
- Pour les appel normal, le temps d'appel n'augmente pas significativement d'après la taille du paramètre.
- Pour les appel RMI local, le temps d'appel augmente significativement a partir de x=5.
- Pour les appel RMI distant le temps d'appel augmente significativement a partir de x=3.

Ces résultats s'explique facilement, le fait que le temps d'appel pour la fonction local reste constant et est moindre comparé aux autres appel viens du fait que la fonction est appelé directement, sans passé par aucune API ni par le réseau. Le fait que l'appel RMI local prenne plus de temps viens du fait que l'API RMI rajoute du temps d'exécution. Finalement les appels RMI distant prenne le plus de temps car on utilise l'API RMI mais en plus il y a le fait qu'il s'agit d'un appel a une fonction distante, donc le réseaux rajoute un délais supplémentaire.
A travers les résultats de cette expérience on peut conclure que plus la taille des donnée transmise a travers l'API RMI de java est grand plus le temps d'appel augmente rapidement. Par contre l'API RMI permet au développeur de faire des appels a des fonctions hébergé sur des serveurs distants sans trop se soucier de la couche des communications réseau car RMI permet de faire abstraction de cela et facilite donc la programmation de ce type d'applications.

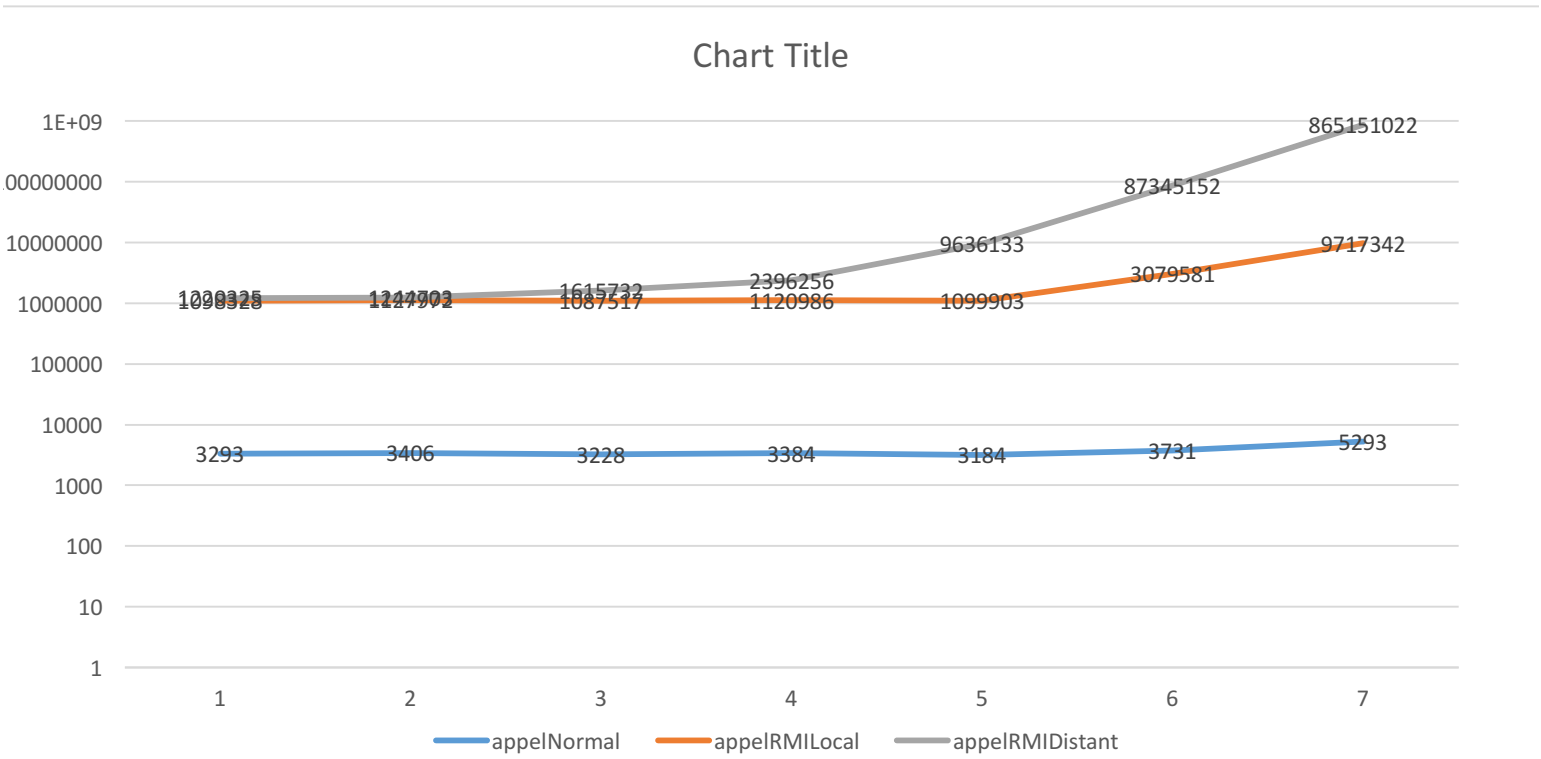

Figure 1:Graphique du temps d'appel d'après la taille du paramètre.

Question 2 :
Afin d'expliquer les interactions entre les différents acteur lors d'un appel d'une fonction RMI, un schéma a été réaliser.

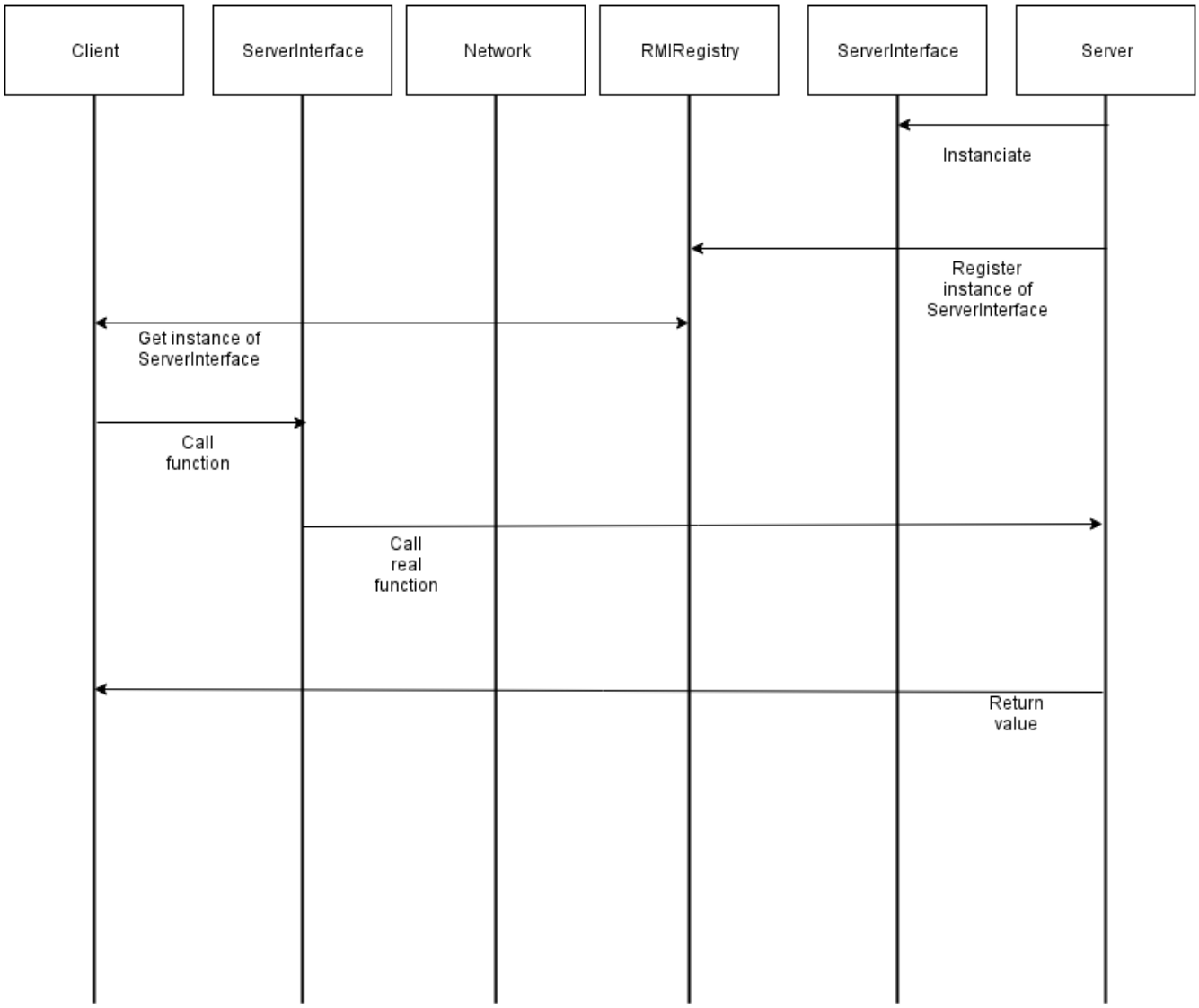

Figure 2:Interactions entre les acteurs

Dans ce schéma on peut voir que bien que le réseaux ce trouve entre le serveur et le client il n'y a aucune interaction direct avec celui ci. Toute les interactions avec le réseaux sont caché par l'API RMI. Plus en détails, voici l'explication de chaque interactions, avec les méthodes, attributs et les classes utilisé dans l'exemple de code fournis :

- Actions du serveur : Il s'agit de la classe Server dans le code fournis.

1) Instanciate : La classe Server Instancie la classe ServerInterface en appelant UnicastRemoteObject.*exportObject*(**this**, 0), cette méthode retourne un objet sur lequel on va effectuer un cast en ServerInterface. On instancie ServerInterface de cette façon afin de rendre les fonctions auxquels il donne accès accessible a travers l'API RMI.

2) Register instance of ServerInterface : la classe effectue cette action grâce a l'appel de la fonction registry.rebind("server", stub); Mais avant de pouvoir effectué cette action, on doit récupérer l'objet Registry avec l'appel LocateRegistry.*getRegistry*(); .

3) Return value : La classe Server retourne une valeur a la suite de l'appel d'une fonction. Pour cela on fait un simple return dans la fonction en question.

-Actions de ServerInterface : La classe ServerInterface est une interface, elle n'effectue donc aucune action concrète. Elle sert de liste des différentes fonctions que le serveur implémente et dont les clients peuvent accéder.

-RMIRegistry : Il s'agit du registre que l'API RMI utilise afin de lister les serveur. Afin que l'application fonctionne correctement il faut lancer le registre RMI dans le dossier /bin du projet.

- Client:Le client est représenté par la classe Client.

1) Get Instance of ServerInterface : Le client est appellé par l'utilisateur qui entre l'adresse IP flottante du serveur en argument. Cet argument est passé a la fonction `LocateRegistry.getRegistry(hostname);` qui permet de recuperer le registre grace a cette adresse. Par la suite le client récupère l'instance de ServerInterface dans le registre avec la fonction `(ServerInterface) registry.lookup("server");` .

2) Call function : Le client appel les fonctions distantes a travers l'instance de ServerInterface récupérer plus tôt. Un simple appel de fonction est effectué. On peut stocké la valeur de retour dans une variable, si la fonction distantes retourne une valeur.