TP 4
RMI et Architectures

3 heures

Prérequis : TP3 terminé

1 RMI

1.1 Passage de paramètres

Dans cet exercice nous allons mettre en évidence le fonctionnement du passage de paramètres en RMI.

1. Écrivez une interface distante *Server*, possédant deux méthodes, la première prenant un type primitif en paramètre, la deuxième prenant un objet de type *MyObject* ;
2. Écrivez la classe *MyObject* avec au moins un attribut de type int ;
3. Quelle interface doit implémenter *MyObject* ?
4. Implémentez un serveur et écrivez un client qui montre que le passage de paramètres se fait par copie profonde.

1.2 Retour de méthode

Dans le cas où une méthode distante retourne une valeur, elle est également transmise par copie profonde.

1. Ajoutez à votre serveur deux méthodes distantes retournant un type primitif pour l'une et un *MyObject* pour l'autre ;
2. Vérifiez que les valeur sont retournées par copie profonde.

1.3 Référence RMI

Il reste un dernier point à vérifier, le cas où les références manipulées sont distantes.

1. Ajoutez une méthode distante à votre serveur prenant un objet de type *Server* ;
2. Ajoutez une méthode non distante à votre serveur prenant un objet de type *Server* en paramètre ;
3. Finalement, ajoutez une méthode distante nommée *test* dont l'implémentation fait appel aux deux méthodes précédentes avec *this* en paramètre ;

4. Testez avec votre client et affichez le type réel des paramètres des méthodes. Que constatez vous ?

5. Ajoutez une méthode distante de signature `Server returnServer() throws ...` dont l'implémentation retourne *this* ;

6. Appelez cette méthode du coté du client et affichez le type réel de l'objet retourné par l'appel de méthode. Que constatez vous ?

2 Annuaire distribué

Dans cet exercice nous allons étudier deux manières de construire une application simulant un annuaire. L'application consiste à pouvoir stocker un numéro de téléphone pour un nom donné et de retrouver le numéro de télephone associé à un nom.

2.1 Version 1 : Architecture client/serveur

La première version que nous allons réaliser repose sur une architecture de type client/serveur. Le principe est que plusieurs clients peuvent se connecter à un serveur, qui maintient les entrées de l'annuaire, pour en ajouter de nouvelles ou pour retrouver un numéro à partir d'un nom.

1. Créez une interface distante *Directory* contenant deux méthodes :
   - une nommée *put* qui renvoie rien mais prend deux paramètres de type String correspondant au nom et au numéro de téléphone ;
   - une nommée *get* qui prend un paramètre de type String (nom recherché) et renvoie une String contenant le numéro trouvé ou bien *null* si aucune entrée ne correspond au nom spécifié.

2. Créez et implémentez l'interface *Directory* dans une classe nommée *DirectoryImpl* ; Les entrées doivent être stockées dans une *Map* en mémoire ;

3. Ajoutez une méthode *main* à la classe *DirectoryImpl* pour créer une instance du serveur et l'enregistrer dans le RMI registry ;

4. Testez votre application en simulant plusieurs clients faisant des appels distants aux méthodes *put* et *get* de manière séquentielle ;

5. Faites de même en simulant cette fois ci plusieurs clients faisant au moins 10^4 put avec des noms différents et de manière concurrente. Pour cela vous pouvez utiliser un mécanisme de haut niveau nommé les *Executors* qui permettent de créer un pool de threads. Utilisez par exemple un nombre de threads égale au nombre de coeurs disponibles sur votre machine plus un ;

6. Modifiez le code pour afficher le numéro associé à chacun des noms ajoutés dans l'annuaire ;

7. Que faut il faire pour que *DirectoryImpl* supporte des accès concurrents sans aucune situation de compétition ? Corrigez et vérifiez.

2.2 Version 2 : Architecture P2P structurée

Cette seconde version consiste à mettre en place une version distribuée d'un annuaire. Pour cela on va s'appuyer sur une version simplifiée du protocole *Chord* [1]. Les entrées de notre annuaire ne sont plus forcemment stockées sur une seule et même machine mais sur plusieurs. Chacune des machines qu'on nomme pair à un identifiant unique représenté par la classe *Identifier*. Cet identifiant est unique dans un anneau de taille pré-définie 2^m. Si $m = 31$, il est alors possible d'avoir au maximum $2^{31} - 1$ pairs dans l'anneau.

Dans cette version simplifiée chacun des pairs a uniquement une référence vers sont successeur et son prédécesseur. Cependant, il gère un ensemble de clés. Une clé est représentée par la classe *Key* et correspond à une valeur de hachage d'une ressource qu'on souhaite indexer dans un anneau *Chord*. Cette valeur permet d'identifier le pair qui va indexer notre ressource. Un pair p gère l'ensemble des clés se trouvant entre l'identifiant de son prédecesseur *p.predecessor* exclu et son identifiant inclus.

Pour connaître quel pair doit indexer une clé k il suffit de trouver le successeur de k. Pour cela il faut tout d'abord connaître une référence vers un pair du réseau. On suppose cela possible grâce à un tracker qui maintient des références vers certains des pairs faisant parti d'un réseau *Chord*. Une fois le tracker interrogé et la référence récupérée, l'algorithme consiste à parcourir les liens successeurs jusqu'à trouver le pair gérant k.

Bien que simplifiée, la version du protocole *Chord* que l'on souhaite implémenter doit pouvoir supporter l'arrivée de nouveaux pairs en continue. Pour cela, *Chord* possède un algorithme dit de stabilisation qui permet de mettre à jour les liens successeurs et prédecesseurs périodiquement.

Pour simplifier certaines étapes, les classes *Identifier*, *Key* ainsi que les interfaces nécessaires vous sont fournies dans une archive disponible à l'addresse `http://goo.gl/ayzb4b`.

1. Prennez une feuille et essayez de lister les différentes Classes, Interfaces, et méthodes nécessaires pour réaliser un annuaire distribué avec un réseau P2P structuré de type Chord ;
2. Téléchargez, dézippez et analysez le contenu du code fourni ;
3. Comparez les interfaces fournies avec celles que vous avez noté sur votre feuille et essayez de comprendre ce qui est manquant ;
4. Créez une classe nommée *PeerImpl* qui implémente l'interface *Peer* fournie.
   - Un pair à un champs final de type *Identifier* représentant sa position dans l'anneau, une *Map<String,String>* permettant de stocker les entrées de l'annuaire et une référence vers son successeur et son prédecesseur ;
   - La classe *PeerImpl* a un constructeur qui prend en paramètre un identifiant. Ce constructeur doit initialiser les champs nécessaires et démarrer un thread qui va périodiquement (par exemple toute les 500 *ms*) faire appel à la méthode *stabilize*. Pour cela vous pouvez utiliser la méthode *Executors.newScheduledThreadPool(1)* pour créer un *ScheduledExecutorService* ;
   - La méthode *create* doit être appelée uniquement par le premier pair qui initialise le réseau *Chord* afin d'affecter son lien successeur sur lui même ;
   - La méthode *findSuccessor* permet de trouver le pair gérant l'identifiant ou la clé passée en paramètre. Cette méthode doit être implémentée suivant le pseudo code suivant :

```
n.find_successor(id):
    if n = successor:
        return n
```

```
if id ∈ (n, n.successor]
    return n.successor
else
    // forward the query around the circle
    return successor.find_successor(id)
```

— La méthode *join* doit uniquement trouver le successeur du pair qui joint le réseau et l'affecter en tant que lien successeur ;

— Les méthode *put* et *get* doivent respectivement ajouter et récupérer une entrée de l'annuaire dans la *Map* contenu par le pair ;

— Les méthodes *stabilize* et *notify* doivent être implémentées telles que décrites dans l'article original de Chord :

```
// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize():
    x = successor.predecessor;
    if x ∈ (n, successor):
        successor = x
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n'):
    if predecessor is nil or n' ∈ (predecessor,n):
        predecessor = n';
```

— Implémentez les accesseurs, mutateurs et redéfinissez les méthodes *equals* et *hashCode* en vous appuyant sur l'identifiant du pair ;

5. Quelle sont les méthodes devant être synchronisées dans la classe *PeerImpl*. Modifiez votre code en conséquence ;

6. Créez une classe nommée *Main* avec une méthode *main* qui crée un réseau de 5 pairs à l'aide d'une boucle for, chaque pair ayant un identifiant unique entre 0 et $2^{31} - 1$;

7. Dans cette même classe ajoutez une méthode statique nommée *turnAround* qui à partir d'une référence d'un pair fait un tour complet de l'anneau en utilisant les liens successeurs. Cette méthode pourra par exemple produire une sortie similaire à :

```
Started turn around from 300
Visited 400 that has 300 as predecessor and 0 as successor
Visited 0 that has 400 as predecessor and 100 as successor
Visited 100 that has 0 as predecessor and 200 as successor
Visited 200 that has 100 as predecessor and 300 as successor
Visited 300 that has 200 as predecessor and 400 as successor
```

8. Faites appel à la méthode *turnAround* puis faites un sleep de deux secondes deux fois d'affilé puis faites un sleep de 6 secondes avant de terminer par un *turnAround*. Vous devriez remarquer qu'après le premier et/ou second appel à *turnAround* le réseau est pas encore stabilisé et tous les liens ne sont pas corrects. Il faut en effet un certain temps pour que le réseau se stabilise ;

9. Créez une classe *TrackerImpl* qui implémente *Tracker* et qui maintient les références aux pairs qui sont enregistrés. Le tracker doit prendre dans son constructeur un numéro de port qu'il utilisera pour créer un registre RMI et s'enregistrer dedans ;

10. Modifiez votre classe *Main* afin de créer un tracker avant de créer un réseau Chord. Chaque pair doit être enregistré dans le tracker et chaque join effectué en utilisant le tracker pour récupérer aléatoirement une référence d'un pair à joindre ;

11. Finalement, créez une classe *DirectoryImpl* qui implémente *Directory*. Le but de cette classe, qui prend en constructeur une référence vers un *Tracker*, est de cacher à l'utilisateur final le système P2P sous-jacent. Implémentez les méthodes de l'interface. Pour exemple, effectuer un *put* consiste à créer une *Key* pour le nom qu'on souhaite ajouter à l'annuaire puis à trouver le pair gérant cette clé. Une fois la référence du pair trouvé il reste plus qu'à ajouter l'entrée sur le pair en question ;

12. Modifiez votre classe *main* pour créer une instance de *DirectoryImpl* et tester les méthodes *put* et *get*.

Références

[1] Stoica, I. and Morris, R. and Karger, D. and Kaashoek, M.F. and Balakrishnan, H., *Chord : A scalable peer-to-peer lookup service for internet applications*. ACM SIGCOMM Computer Communication Review, 2001