ANDRE Julien
BUISSON Kévin
ROLLIN Antoine

04/05/2016
SI4

RMI-Spork

Applications Réparties – Mini-Projet

L'objectif de ce projet est de mettre en œuvre une application répartie représentant un portail générique de partage de données, à l'aide des différents outils Java vus en cours. Un client pourra alors utiliser ce portail pour déposer ou récupérer des données ou des services.

Pour ce faire nous aurons besoin de :

- Un serveur utilisant la technologie RMI et capable de stocker des données et des services
- Une application cliente qui permet d'interagir avec le serveur
- Un serveur de classes capable de mettre à disposition des fichiers java compilés

Table des matières

1. Serveur de collection universel

1.1. Description

Le serveur de collection est l'entité qui permet de collecter et distribuer les données et services. Il utilise la technologie JNDI pour annuariser des objets Java qui seront mis à disposition des clients. Le serveur représente le cœur de notre application.

La classe CollectionServer représente ce serveur qui sera accédé par les clients. Il est unique et étend donc la classe UnicastRemoteObject. De ce fait les clients ne récupèrent pas une copie du serveur mais une référence à cet objet.

1.2. Collection

La base de données est représentée par une HashMap pour pouvoir stocker des objets et les référencer par des clés uniques.

La collection enregistre également les actions (get, put) client à l'aide de Queues représentant un historique. Pour des raisons de mémoire, ces unités de stockage disposent d'une certaine capacité. Si ces Queues atteignent la limite de stockage, alors l'élément le plus vieux laisse sa place au nouvel élément (principe de la FIFO).

Dans notre projet, nous utilisons un objet nommé Gateway pour interagir avec la base de données. A chaque émission ou réception d'objet, les historiques sont mises à jour et sont accessibles par diverses méthodes de la classe.

1.3. Réception d'un objet

Lorsqu'un objet est réceptionné par le serveur de collection (méthode put), il est stocké dans la base de données et référencé par une clé (un objet ne peut être référencé par une clé qui existe déjà). Ensuite, l'historique des clés reçus est mis à jour.

1.4. Emission d'un objet

Lorsqu'un objet est demandé par un client (méthode get), le serveur de collection va vérifier l'existence de l'objet souhaité. S'il existe, il caste cet objet en DistantObject de façon à le rendre Serializable, et le retourne. Le client recevra donc une copie de l'objet. Ensuite, l'historique des clés demandées est mis à jour.

1.5. Service d'information

Le serveur met à la disposition des clients un service d'information permettant d'obtenir des statistiques sur la collection. Ce service d'information utilise l'objet Gateway notamment pour :

- Récupérer la liste des objets récupérables
- Récupérer l'ensemble des N dernières clés enregistrées
- Récupérer l'ensemble des N dernières clés utilisées
- Récupérer l'ensemble des N clés les plus utilisées

1.6. Gestion des abonnements

Le serveur dispose également d'un service d'abonnement, permettant à ses clients de se tenir informés des modifications sur une donnée ou un service donné. Pour cela, nous avons mis en place une architecture MOM de type centralisée (Spoke and Hub).

Lorsqu'un client modifie un objet, le serveur va notifier tous les subscribers qui sont abonnés à cet objet par le biais d'un topic et grâce à la technologie JMS et ActiveMQ.

2. Client

2.1. Description

Le client est l'entité qui va envoyer ou recevoir des données ou services depuis le portail générique.

2.2. Connexion

La connexion d'un client est représentée par la récupération de l'objet distant CollectionServer à l'aide du JNDI. Ainsi, il peut effectuer toutes actions qu'autorise le serveur, à savoir l'envoi ou récupération d'une donnée ou d'un service et l'utilisation du service d'information.

2.3. Déposer une Donnée ou un Service

Après s'être « connecté » au serveur, le client peut déposer (méthode put) une donnée ou un service. Les objets déposables sur le serveur doivent implémenter l'interface DataInterface pour les données ou l'interface ServiceInterface pour les services. Ces interfaces permettent aux clients qui récupéreront l'objet de pouvoir l'exploiter.

Pour déposer un objet, le client doit spécifier une clé (chaine de caractères) qui sera utilisé pour référencer l'objet dans la collection du serveur.

Notons que dès lors qu'un client pose un objet, il déclenche la publication d'un message dans le « topic » et tiens ainsi les abonnés au courant des changements de l'objet.

2.4. Récupérer une Donnée ou un Service

Après s'être connecté au serveur, le client peut également récupérer (méthode get) une donnée ou un service à l'aide de la clé qui le référence. L'objet distant étant récupéré, le client va d'abord tenter de le caster en service à l'aide de l'interface ServiceInterface. Si le cast échoue, il le caste alors en donnée à l'aide de l'interface DataInterface. L'objet est alors utilisable à travers les méthodes définies par les interfaces.

2.5. Récupération des informations sur la collection

Lorsque le client se connecte au serveur, il récupère la collection distante et donc le service d'information qu'elle embarque. Ce service est alors utilisable par le client et propose un panel de fonctions offrant la récupération d'informations concernant la collection (voir plus haut).

2.6. S'abonner

Le client a la possibilité de s'abonner à un objet via son identifiant au sein de notre serveur. Il pourra alors effectuer une action en fonction du MessageListener créé. Deux listeners sont présents en guise d'exemples dans notre serveur. Un qui informe simplement l'abonné que l'objet a été mis à jour en affichant que tel objet a été mis à jour par tel publieur, et un autre qui recharge automatiquement l'objet dès qu'un autre client publie sur cette clé.

3. Serveur de classes

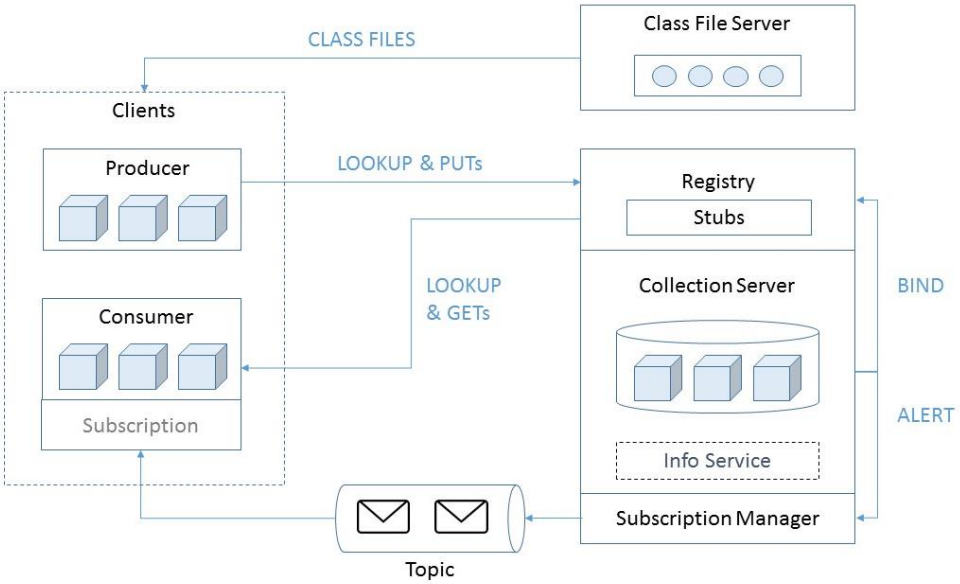
Avec l'architecture actuelle, le problème est le que registre RMI n'a pas accès aux interfaces qu'implémente le serveur de collection et son service d'information, ou les classes de données/services des clients. Il ne peut donc fonctionner que si celui-ci est lancé sur la même machine que le serveur… dans le bon dossier… avec en plus les bonnes classes clients, etc.

Pour pallier ce problème, nous avons mis en place un serveur de classes, qui se chargera de partager les classes nécessaires en utilisant le partage de classes RMI. Celui-ci, configuré correctement, permettra au RMIRegistry de venir télécharger les classes dont il a besoin durant son exécution.

Le serveur de classes a donc été mis en place avec succès, et fonctionne correctement pour le partage des classes serveurs (interfaces). Cependant, pour une raison mystérieuse, et alors que le procédé est le même, le partage ne fonctionne pas pour les classes clients.

/!\ Cf. le mail (Kévin Buisson) que je vous avais fait parvenir, où je vous expliquais le problème avec plus de détails. Vous m'aviez répondu que vous aviez déjà était confronté au même problème, sans pour autant parvenir à le résoudre.

4. Architecture globale



Architecture globale de notre application

5. Prérequis

Les prérequis nécessaires pour la bonne utilisation de notre projet sont les suivants :

- Java 8
- ActiveMQ

6. Utilisation

Pour utiliser notre application, il suffit de :

1) Lancer le serveur de classes : xxxx.sh
2) Lancer ActiveMQ
3) Lancer le serveur : runServer.sh
4) Lancer les clients : runClient.sh (2 clients : producteur et consommateur)
5) Arrêter le RMI Registry : cleanPort.sh (seulement sous Linux)

7. Exécution

Nous détaillerons le scénario d'exécution en 2 parties, une première sans utilisation du serveur de classes, et une seconde avec.

Afin de simplifier les exécutions, des scripts Shell ont été écrits afin d'effectuer de manière automatique et simplifiée le déploiement et le démarrage des différents services.

Dans les deux cas, le registre RMI seront lancés automatiquement sur le port 8082 par défaut.

Une trace d'exécution normale du serveur et des clients est disponible dans les fichiers « output » associés dans le répertoire « SANS class_server » du projet.

7.1. Sans serveur de classes

Se rendre dans le dossier « SANS class_server », puis appliquer l'ordre suivant :

1. Lancer ActiveMQ
2. Exécuter runServer.sh
3. Exécuter runClient.sh

7.2. Avec serveur de classes

Se rendre dans le dossier « AVEC class_server », puis appliquer l'ordre suivant :

1. Exécuter deploy.sh
2. Lancer ActiveMQ
3. Exécuter runServerPart.sh
4. Exécuter runClientPart.sh