Projet: Une plateforme pour objets Java répliqués

Ce projet doit être réalisé en binôme, i.e. au plus deux élèves, et remis sur Campus. La date limite de remise est indiquée dans l'espace correspondant.

Préambule

Remote Method Invocation (RMI) est une plateforme qui facilite le développement d'applications Java réparties. Elle permet à du code Java s'exécutant dans une JVM d'appeler les méthodes d'un objet résidant dans une autre JVM, éventuellement localisée sur une autre machine. RMI assure ainsi la *transparence d'accès* aux objets : l'appel de méthode sur un objet distant se fait (presque) de la même manière que pour un objet local.

Sujet

Il s'agit d'enrichir les fonctionnalités de RMI pour permettre le développement d'applications réparties composées d'objets *répliqués*. Il s'agit notamment d'assurer la transparence de *localisation* et de *réplication* des objets de l'application. En fonction de leurs besoins, les objets doivent pouvoir choisir entre plusieurs *politiques de réplication* : active, passive, semi-active. Il faut également pouvoir identifier les méthodes qui *ne modifient pas l'état* de l'objet répliqué. La plateforme doit en tirer parti pour diminuer leur durée d'exécution et répartir la charge sur les différentes répliques d'un objet.

Le sujet n'est pas davantage détaillé pour laisser libre cours à votre imagination. Un plan de travail est néanmoins proposé ci-dessous.

Important : pour simplifier les développements, on ne gère pas les pannes dans le cadre de ce projet.

Plan de travail suggéré

Les paragraphes ci-dessous décrivent les premières étapes que vous pouvez suivre pour mener à bien le projet.

1. Transparence de localisation

Pour des raisons de sécurité, l'annuaire (registry) standard RMI n'accepte que les appels `bind()` venant du calculateur sur lequel il s'exécute. En conséquence, (1) dans une application RMI, il y a autant d'annuaires que de calculateurs hébergeant l'application, (2) lorsqu'un client veut interagir avec un serveur, il doit connaitre la machine sur laquelle il s'exécute : la transparence de localisation n'est pas assurée.

Développez un annuaire global, appelé `GlobalRegistry`, qui lève cette restriction. `GlobalRegistry` enregistre tous les serveurs constituant l'application, quelle que soit leur calculateur d'origine. Seul le calculateur sur lequel s'exécute l'annuaire global a besoin d'être connu de l'application répartie. Développez également la classe `LocateGlobalRegistry`, qui est à `GlobalRegistry` ce que `LocateRegistry` est à `Registry`.

2. Réplication – Gestion d'objets sans état

Développez une nouvelle version de l'annuaire global qui autorise l'enregistrement par `bind()` de plusieurs répliques d'un même objet. En supposant qu'on ne gère que des objets sans état, développez une implémentation de `lookup()` qui assure la répartition de charge entre les répliques d'un objet.

Envisagez différents *grains* de répartition. Par exemple, un client peut se voir associé à une nouvelle réplique à chaque `lookup()` seulement, ou plus finement à chaque appel de méthode sur l'objet. Envisagez également différents *critères* de répartition. Par exemple, les répliques seront distribuées aux clients selon un simple ordre circulaire, ou plus finement en fonction de la charge des machines qui les hébergent.

3. Réplication – Gestion d'objets avec état

Modifiez votre architecture pour traiter maintenant les objets avec état.

Envisagez différentes politiques de réplication, par exemple active, passive et semi-active. Il faut transposer à un monde d'objets communiquant par appel de méthode les exemples étudiés en cours dans le cadre de processus communiquant par messages.

Distinguez également les méthodes qui modifient l'état de l'objet, et celles qui ne le modifient pas. La plateforme doit en tirer parti pour alléger la charge de traitement sur les répliques, et raccourcir le temps d'exécution perçu par les clients.

Objectifs

La plateforme développée doit être générique. En particulier, elle doit pouvoir gérer, *simultanément* :

- plusieurs services et plusieurs clients,
- des objets répliqués et d'autres non répliqués,
- des objets répliqués sans état et d'autres avec état,
- pour un objet répliqué avec état, des méthodes modifiant son état et d'autres non.

La plateforme doit fournir des services correctement encapsulés. Son fonctionnement doit être proche de RMI pour ne pas perturber les développeurs d'application. Elle doit respecter les différents niveaux de *transparence* :

- transparence d'accès : un client accède à un objet distant comme à un objet local,
- transparence de localisation : un client ne sait pas où est localisé l'objet qu'il accède,
- transparence de réplication : un client ne sait pas si l'objet qu'il accède est répliqué ou non, si l'objet est avec ou sans état, si la méthode qu'il appelle modifie l'état de l'objet ou non.

Barème

Le barème *prévisionnel* est donné ci-dessous.

Item	Marks
GlobalRegistry	**2**
Stateless Replication	**6**
Basic	*2*
Advanced	*4*
Stateful Replication	**6**
Active	*2*
Get/set	*2*
Passive	*2*
Demo Application	**2**
Report	**4**

Remise

Le projet doit être remis sur Campus sous la forme d'un fichier archive, nommé **NOM1.Prénom1[.NOM2.Prénom2].zip** ou .rar pour identifier les membres du binôme. Si vous déposez votre projet plusieurs fois, faites-le toujours svp sous le même utilisateur Campus.

L'archive doit contenir les éléments suivants :

- Le projet Eclipse contenant votre plateforme (dans le package `framework`) et une application exemple (dans le package `application`),
- Un rapport décrivant votre implémentation pour chacune des étapes. Attachez-vous svp à décrire l'architecture, les principes, les choix effectuées. Vous pouvez aussi fournir des copies d'écran commentées montrant des exemples d'exécution.