SCIENCES SORBONNE UNIVERSITÉ

Travaux pratiques : Zookeeper

Jonathan Lejeune

Objectif

Ce TP vous exerce à programmer des mécanismes distribués en Java grâce à des clients Zookeeper.

Prérequis

Vous devez avoir installé la Zookeeper et avoir configuré votre espace de travail correctement pour coder et compiler vos programmes java avec l'API cliente de Zookeeper. Vos programmes seront validés avec JUnit en mode distribué : chaque test configure et démarre en local un ensemble de serveurs Zookeeper (paramétré à 3 dans les tests), test une fonctionnalité et arrête l'ensemble des serveurs.

Informations complémentaires

Deux classes vous sont fournies :

- `datacloud.zookeeper.util.ConfConst` qui ne contient que des constantes utiles au déploiement des serveurs et à la réalisation de certains exercices de ce TP.
- `datacloud.zookeeper.ZkClient` qui permet créer une connexion vers Zookeeper, maintenir un identifiant unique parmi l'ensemble des clients connectés à Zookeeper. Cette classe fera également office de `Watcher` par défaut, mais la méthode `void process(WatchedEvent event)` restera abstraite et sera implantée dans les classes filles que nous programmerons dans la suite de ce TP. Vous noterez que l'ensemble des identifiants des clients connectés se fait grâce à la création de znodes éphémères et incrémentaux dans un znode de référence */ids*.

N'oubliez pas qu'au sein d'un même client vous serez dans un contexte multi-threadé. Il est donc nécessaire de gérer une synchronisation locale sur les accès concurrents des variables partagés.

Exercice 1 – Membership

L'objectif de cet exercice est que chaque zkclient connecté à Zookeeper puisse maintenir dans une liste locale (en attribut), les identifiants des zkclients qui sont actuellement connectés sur le système. Ainsi à chaque départ d'un zkclient ou à chaque arrivée d'un nouveau zkclient, la liste locale de chaque zkclient connecté doit être mise à jour.

Question 1

Codez la classe `datacloud.zookeeper.membership.ClientMembership` qui étend la classe `ZkClient` décrite en introduction pour réaliser cet exercice. Votre solution se basera sur une surveillance des nœuds enfants du znode */ids*. Cette classe offrira la méthode `getMembers()` qui renvoie la liste locale des clients connectés du point de vue de l'instance.

Question 2

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.zookeeper.tests.ZkClientMembershipTest`

Exercice 2 – Barrière de synchronisation

Une barrière de synchronisation permet de bloquer un ensemble de processus (dans note cas les zkclients) tant qu'une condition n'est pas respectée. Nous allons dans un premier temps programmer une première version d'une barrière où la condition de blocage sera l'existence d'un znode donné.

Question 1

Programmer une classe `datacloud.zookeeper.barrier.SimpleBarrier` dont les spécifications sont :

- offrir un constructeur qui prend en paramètre un `ZkClient` et un chemin absolu de znode qui caractérisera la barrière. Ce znode est créé si celui-ci n'existe pas afin d'initialiser correctement la barrière.
- offrir une méthode `sync()` qui permet à l'appelant de se mettre en attente tant que le znode caractérisant la barrière existe.

Question 2

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.zookeeper.tests.SimpleBarrierTest`

Nous allons à présent programmer une barrière qui permet d'assurer que N zkclients aient tous passé un point spécifique de leur programme. Ainsi un appel à `sync()` devra être bloquant tant que N clients existants n'ont pas atteint la barrière, c.-à-d. fait appel à `sync()`.

Question 3

Programmer une classe `datacloud.zookeeper.barrier.BoundedBarrier` dont les spécifications sont :

- offrir un constructeur qui prend en paramètre un `ZkClient`, un chemin absolu de znode qui caractérisera la barrière et un entier pour la taille de la barrière (i.e. N). Si le znode n'existe pas, il est alors créé et l'entier est alors pris en compte pour initialiser correctement la barrière. Si le znode existe déjà , cela signifie qu'il y a déjà un zkclient qui a initialisé la barrière, l'entier n'est alors pas pris en compte.
- offrir une méthode `sync()` qui est bloquant tant que N clients existants n'ont pas fait appel à `sync`. Il faudra assurer que tout znode ayant servi au bon fonctionnement de cette synchronisation soit supprimé une fois que tous les zkclients ont été réveillés et que la barrière n'est plus utilisée.
- offrir une méthode `sizeBarrier` qui renvoie la taille réelle de la barrière. On notera que la valeur renvoyée par cette méthode n'est pas forcément celle qui a été passée en paramètre du constructeur si la barrière avait déjà été initialisée.

Question 4

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.zookeeper.tests.BoundedBarrierTest`

Exercice 3 – Diffusion de messages

Nous souhaitons mettre en place un mécanisme de pusblish/subscribe entre les différents zkclients. Ce mécanisme distingue deux types de zkclient :

- les subscribers : ils peuvent s'abonner à des topics.
- les publishers : ils peuvent envoyer un message à un topic. Chaque message envoyé sur un topic t par publisher, devra être reçu une seule et unique fois par l'ensemble des subscribers abonnés à t.

Question 1

Programmer une classe `datacloud.zookeeper.pubsub.Publisher` qui étend `ZkClient` et qui offre la méthode `void publish(String topic, String message)` qui permet d'envoyer un message sur un topic donné.

Question 2

Programmer une classe `datacloud.zookeeper.pubsub.Subscriber` qui étend `ZkClient` et qui offre la méthode `void subscribe(String topic)` qui permet de s'abonner à un topic donné. Le subscriber maintient pour chaque topic auquel il est abonné, la liste des messages reçus depuis le début de l'abonnement. La classe `Subscriber` offre une méthode `List<String> received(String topic)` qui renvoie la liste des messages reçus associés au topic donné.

Question 3

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.zookeeper.tests.PubsubTest`