INF4402 – Systèmes répartis sur Internet

TP3 – Systèmes *peer-to-peer*

Simon Marchi

20 novembre 2012

École Polytechnique de Montréal

Introduction

Ce troisième et dernier travail pratique a pour thème les réseaux de communication poste-à-poste, mieux connus sous le nom *peer-to-peer*. Vous aurez à concevoir un système de clavardage décentralisé qui permet aux clients qui y sont connectés de discuter dans différentes salles de discussion virtuelles. L'utilisation de tels systèmes pourraient être envisagés dans des cas où l'on veut contourner la censure, où l'utilisation de serveurs centraux rend le système trop vulnérable. On peut aussi penser aux situations d'urgences, comme après un ouragan ou un tremblement de terre. Souvent, des réseaux de types *mesh* sont rapidement mis en place afin de permettre la coordination des opérations de secours. Là aussi, l'utilisation d'un serveur central peut rendre le système vulnérable si l'alimentation en électricité n'est pas fiable, par exemple.

Remise

- Méthode: Par Moodle, un seul membre de l'équipe doit remettre le travail mais assurez vous d'inclure les deux matricules dans le rapport et le nom du fichier remis.
- Échéance: mardi le 4 décembre 2012, avant 16h, voir le plan de cours pour les pénalités de retard.
- Format: Une archive au format .tar.gz contenant votre code et votre rapport contenant vos réponses aux questions. Pour le code, la remise d'un projet Eclipse (le ou les projets seulement, pas le *workspace*) est préférable.

Vous pouvez faire le laboratoire seul, mais le travail en binôme est encouragé. On vous demande d'inclure un *README* expliquant comment exécuter votre TP. Après la remise, il se peut que le chargé de laboratoire vous demande de faire une démonstration de votre travail, donc gardez votre code.

Barème

Implémentation: 8 points
Questions: 12 points (3 points par question)
Total: 20 points, valant pour 10% de la note finale du cours. Jusqu'à 4 points peuvent être enlevés

pour la qualité du français.

Documentation

Les librairies utilisées:

http://code.google.com/p/openkad/

http://code.google.com/p/dht/

Openkad utilise *Guice* pour gérer l'injection de dépendances. Bien que ce soit une bonne pratique de développement logiciel, ça rend le code un peu mêlant pour les non-initiés. Vous pouvez jeter un coup d'oeil au début du guide d'utilisateur de *Guice* pour vous familiariser avec les bases de cette technique.

http://code.google.com/p/google-guice/

http://code.google.com/p/google-guice/wiki/Motivation?tm=6

Le protocole:

http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html

Description du travail demandé

En vous basant sur le code fourni, vous devez concevoir un système de clavardage décentralisé. La bibliothèque sur laquelle nous allons construire est openkad[1], une implémentation facile d'approche du protocole Kademlia[2]. Ce protocole, généralement destiné aux logiciels de partage de fichiers, identifie chaque client et chaque ressource à l'aide d'une clé unique. Une distance peut être calculée entre deux clients ou un client et une ressource en prenant le résultat du ou exclusif (XOR) entre les deux clés. Une ressource est stockée chez les quelques clients dont les clés sont les plus rapprochées de celle de la ressource selon cette métrique. Pour trouver une ressource ou un client dans le réseau, un client a absolument besoin de la clé correspondante. Il interroge quelques uns des autres clients dont il a déjà les informations (suite à ses actions antérieures) et qui sont plus près que lui de la clé recherchée. Ceux-ci lui renverront une liste de clients qui sont encore plus près de la cible afin que la recherche puisse continuer et éventuellement converger.

1 http://code.google.com/p/openkad/

2 http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

À plus haut niveau, le protocole Kademlia agit comme une table de hachage distribuée et décentralisée. N'importe quel client peut faire des insertions et des consultations et les données sont distribuées aussi équitablement que possible entre les pairs, en tenant compte du fait que ceux-ci peuvent disparaitre à tout moment. La librairie que nous utilisons permet d'associer plusieurs valeurs à la même clé. Lorsqu'un client fait une requête pour une clé, il récupère l'union des ensembles de valeurs trouvés sur les différents clients susceptibles d'être responsables de cette clé.

ChatClient

Le client devra être capable d'effectuer les opérations décrites ici. Pour nos besoins, nous considérerons la clé du client (ou plutôt son *toString()*) comme étant son nom.

- Montrer la liste des salles existantes sur le réseau.

  La clé “rooms” devra contenir la liste des noms des salles existantes. Cette opération revient donc à aller chercher les valeurs associées à “rooms” dans la table de hachage distribuée et les afficher.

- Abonner le client à une salle.

  Le nom de la salle servira de clé pour la liste des clients qui y sont abonnés. S'abonner à une salle correspond donc à ajouter son nom (ou un autre objet identifiant le client) dans cette liste. Comme la valeur qu'on place dans la table peut être n'importe quel *Serializable*, vous pouvez placer un objet qui permettra de contacter directement ce client.

- Montrer la liste des clients membres d'une salle.

  Cette opération correspond à montrer la liste des valeurs associées à la clé du nom de la salle.

- Envoyer un message aux membres d'un salle.

  Pour ce faire, le client doit d'abord obtenir la liste des clients de la salle grâce à la méthode précédente. Puis, il envoie le message à chaque client de façon indépendante.

Pour cette dernière fonction, openkad offre la possibilité d'envoyer des messages de notre choix en utilisant l'infrastructure de transmission de messages du protocole. Un exemple d'enregistrement de *handler* de message et d'envoi est montré sur la page d'accueil du projet (http://code.google.com/p/openkad/). Lorsqu'un message est reçu, il devrait être affiché de façon claire dans la console. Par exemple:

```
Message reçu de <client> sur <salle>: <contenu>
```

Questions

Répondez aux questions suivantes. Mentionnez les références que vous utilisez s'il y a lieu.

Question 1

Dans notre programme, lorsqu'un client envoit un message à une salle de discussion, il envoie un message séparé à chacun des membres de cette salle. On peut imaginer que cela devient lourd pour des salles avec plusieurs milliers de personnes. Proposez une amélioration qui permettrait de réduire le nombre total de messages transmis.

Question 2

Dans ce travail, nous n'avons pas parlé de l'aspect sécurité. Rien n'est fait pour assurer l'authenticité des messages (le fait qu'ils viennent bien du client prétendu) ni la confidentialité. Aussi, comment est-ce qu'un client devrait savoir si une requête de modification ou de suppression est légitime ? Après tout, un client malicieux pourrait s'inviter dans le réseau et envoyer des messages de suppression à tous les clients. Quels moyens ou techniques pourrions nous utiliser afin d'améliorer ces aspect ?

Question 3

Dans Kademlia, chaque fois qu'un client a accès aux informations d'un autre, il les sauve dans sa liste de contacts. Toutefois, cette liste a une taille maximale, généralement paramétrable. Lorsqu'un nouveau client est rencontré mais que notre liste de contacts est pleine, est-il préférable de simplement ignorer cette nouvelle information, ou de remplacer un ancien contact par le nouveau ? Pourquoi ?

Question 4

À quoi correspond l'étape du *bootstrap* dans Kademlia et pourquoi est-elle nécessaire ? Comment est-ce que cela se traduit dans le monde réel, comment est-ce qu'un client qui veut se connecter au réseau pourrait-il obtenir l'adresse d'un client existant ?

Annexe

Voici les instructions pour l'utilisation du client.

```
Usage: ./client localPort [bootstrapIp boostrapPort]
```

- localPort: port auquel s'attacher
- bootstrapIp: adresse IP d'un client existant
- bootstrapPort: port d'un client existant

Les deux derniers arguments doivent être omis pour le premier client du réseau. Par exemple

```
$ ./client 5000
(dans une autre console)
$ ./client 5001 localhost 5000
(dans une autre console)
$ ./client 5002 localhost 5001
```

En supposant que le code a été complété correctement, cela crée un réseau avec trois participants.