Réseaux pair à pair

Le but de ce TD est de mettre en place un réseau pair à pair. Cet énoncé de TD est très proche du TD de [3], lui même librement inspiré de Chord [2]. Un canevas de programme vous est fourni dans l'archive `src-td4.tgz`, disponible sur le compte `/home/lmarchal/` (et sur la page web des TDs [1]). Nous utiliserons le langage java ; vous pouvez trouver de la documentation sur l'API de java à l'adresse `http://java.sun.com/j2se/1.4.2/docs/api/`.

Aujourd'hui nous nous contentons d'implémenter les fonctions de bases d'un réseau pair à pair comparable à Chord. Dans les prochaines séances, nous introduirons des méchanismes plus complexes de routage rapide, et de départ de nœuds.

1 Se joindre au réseau pair à pair

Un pair est caractérisé par son ID, son adresse IP et un numéro de port où le contacter. On utilisera pour cela la classe minimaliste `PeerInfo.java`. Chaque membre du réseau créera un socket Datagram sur ce port pour recevoir des messages et en envoyer. Cela vous permet de créer plusieurs membres d'un réseau sur la même machine en leur donnant des numéros de ports différents.

Pour communiquer, les nœuds utiliseront 6 types de messages :
- PING, PING_REP et JOIN sont utilisés pour se joindre au réseau,
- PUBLISH, GET et GET_REP sont utilisés pour insérer et rechercher des données dans la table de hachage et sont décrits dans l'exercice suivant.

Une classe `Message.java` vous est donnée pour manipuler les messages facilement et vous consacrer sur l'opération pair à pair. On pourra se contenter de l'annexe A pour comprendre son utilisation. Chaque message aura l'en-tête suivante (une ligne correspond a quatre octets) :

```
+----+----+----+----+
|  type   |   TTL   |
+----+----+----+----+
|      destID       |
|                   |
+----+----+----+----+
|      fromID       |
|                   |
+----+----+----+----+
|       fromIP      |
+----+----+----+----+
|fromPort |   ZERO  |
+----+----+----+----+
|      succID       |
|                   |
+----+----+----+----+
|      succIP       |
+----+----+----+----+
|succPort |  ZERO2  |
+----+----+----+----+
```

- type est le type du message (1 pour PING, 2 pour PING_REP,...)
- TTL est un "Time To Live", garde fou pour éviter que des paquets ne circulent à l'infini : il est décrémenté chaque fois qu'un message est forwardé et un paquet de TTL 0 n'est jamais forwardé.
- destID dépend du type de message et contient généralement un ID recherché.
- fromID est l'ID du pair à l'origine du message.
- fromIP est l'adresse IP du pair à l'origine du message.
- fromPort est le port du pair à l'origine du message.
- ZERO est inutilisé (ou disons plus glorieusement qu'il est réservé pour un usage futur et doit être mis à zéro)
- succID, succIP et succPort désignent de même l'ID, l'IP et le port du successeur du pair à l'origine du message.

Décrivons les messages de types PING, PING_REP et JOIN :
- PING sert à rechercher qui est en charge d'un identificateur. Le champs destID contient alors cet identificateur.
- PING_REP sert à répondre à un PING. Un nœud qui reçoit un PING pour un identificateur dont il est en charge doit répondre un message PING_REP dans lequel destID a même valeur que dans le PING reçu. Un PING_REP est directement envoyé au pair à l'origine du PING (son adresse IP et son port sont indiqués dans le PING).

– JOIN sert à un nouveau nœud pour s'insérer dans le réseau, destID doit indiquer l'ID du nœud à qui est envoyé le message.

Le but de cet exercice est de gérer l'insertion de nouveaux nœuds dans un réseau pair à pair. On pourra partir du squelette `Peer.java`. Le premier nœud du réseau est seul et son successeur est lui même. La commande `java Peer new 1024` permettra par exemple de créer le premier nœud d'un réseau sur le port 1024.

Un nouveau nœud tire un ID au hasard, émet un PING pour rechercher le nœud x en charge de cet ID. Quand il reçoit la réponse, il prend pour successeur le successeur de x. Il envoie ensuite directement à x un message JOIN pour lui indiquer son arrivée. x prend alors ce nœud comme successeur. La commande `java Peer join 1025 localhost 1024` permettra par exemple de créer un pair sur le port 1025 et de le joindre au réseau du pair tournant sur `localhost` sur le port 1024.

Les fonctions suivantes vous sont fournies dans `Peer.java` :

– `Peer.randomId()` renvoie un `long` aléatoire.
– `Peer.inInterval(long id, long id1, long id2)` renvoie vrai si `id` est dans l'intervalle `[id1, id2[` de l'anneau.
– `Peer.forwardMessage(Message m, Peer p)` décrémente le TTL de m et l'envoie à p si le TTL est positif.
– `Peer.run()` est une suggestion pour gérer la réception des messages.

Il est conseillé de ne rattraper qu'un minimum d'exceptions et de laisser les autres s'échapper pour pouvoir déboguer plus facilement. Pour mettre au point votre programme, lancer plusieurs fois le programme dans plusieurs fenêtres `xterm` pour former un petit réseau pair à pair sur votre machine. (Tous les nœuds auront même adresse IP, mais des ports différents, par exemple 1024, 1025, 1026,...)

2 Publier et retrouver une donnée

Rajouter une table de hachage (`java.util.Hashtable`) dans chaque membre du réseau. Un message PUBLISH est routé comme un PING et contient de plus des données (une chaîne de caractères ici) associées à la clé destID. Le nœud en charge de destID rajoute une entrée (destID, données) dans sa table de hachage. Un message GET est similaire à PUBLISH sauf qu'il ne contient aucune donnée ; le nœud en charge de l'identificateur répond directement au pair à l'origine du message par un GET_REP qui contient le même destID que le GET plus la donnée associée. (Une donnée vide indique que l'identificateur n'était pas associé à une donnée.) La donnée est directement codée à la suite de l'entête des messages. Le champs ZERO2 est utilisé pour coder la longueur (en octets) de la donnée. Les méthodes `Message.setData` et `Message.getData` permettent d'accéder à la donnée contenue dans un message.

3 Test d'un réseau

Écrire une classe `Visitor` similaire à `Peer` pour envoyer des messages dans un réseau pair à pair dont on connaît un contact (c'est à dire son adresse IP et son port).

La commande `java Visitor ping IP port destID` permettra notamment de lancer un PING sur destID au contact d'adresse IP et de port spécifiés, puis d'afficher le PING_REP reçu en retour. Offrir de manière similaire la possibilité de publier ou de rechercher une donnée associée à une clé.

Proposer de plus un "traceroute". L'idée de traceroute est de découvrir le chemin suivi par un message PING en envoyant un message PING avec même destID et TTL 1, puis un autre avec même destID et TTL 2, et ainsi de suite jusqu'à atteindre le nœud en charge de destID. Pour en déduire le chemin suivi pour atteindre ce nœud final, il faut qu'un nœud intermédiaire (qui n'est pas en charge de destID) réponde au PING si toutefois le TTL vaut 0 (le fonctionnement prévu plus haut prévoyait qu'il jette le message sans le forwarder dans ce cas).

A Utilisation de Message.java

Un objet de la classe `Message` est simplement constitué d'un `DatagramPacket` (champ packet). La classe possède des fonctions pour modifier ou lire les champs dans la partie données du `DatagramPacket` (les autres champs de `DatagramPacket` ne sont pas considérés par cette classe).

- `Message()` crée un nouveau message vide (utile pour recevoir un message).
- `Message (short type, short ttl, long destID, PeerInfo from, PeerInfo succ)` permet de créer un PING_REP ou un JOIN.
- Les constantes `Message.PING`, `Message.PING\_REP`,... sont définies dans cette classe.
- `Peer getFromPeer()` lit les champs fromID, fromIP et fromPort, et renvoie le pair associé. `getSuccPeer()` fait la même chose pour succID, succIP, succPort. que le message est un PING_REP ou un JOIN.
- `void setFromID (long l)` permet de mettre `l` dans le champ fromID. De manière similaire, les méthode `set`... permettent d'écrire dans un champ du message.
- `long getFromID()` renvoie la valeur codée dans le champ fromID du message. De manière similaire, les méthode `get`... permettent de lire dans un champ du message.
- La classe `BytesTools` possède des méthodes pour coder et décoder des `short`, des `int`, des `long`, des `InetAddress` et des `String` dans un tableau de byte. Les méthodes de la classe `Message` sont faites à partir de ces fonctions élémentaires.

Sources et références

[1] Loris Marchal. TDs et TPs du cours d'algorithmique des réseaux et des télécoms. http://graal.ens-lyon.fr/~lmarchal/ART.html.

[2] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[3] Laurent Viennot. Algorithmique des réseaux. http://www.enseignement.polytechnique.fr/profs/informatique/Laurent.Viennot/.