Réseaux pair à pair

Le but de ce TD est de mettre en place un réseau pair à pair, en s'inspirant fortement 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'implanter les fonctions de bases d'un réseau pair à pair comparable à Chord. Dans les prochaines séances, nous introduirons des mécanismes 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 DatagramSocket 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. Elle est décrite dans l'annexe A, ainsi que les fonctions utiles pour manipuler des messages. Un message contient les champs suivants :

- **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 transmis et un paquet de TTL 0 n'est jamais transmis.
- **contentID** dépend du type de message et contient généralement un ID recherché ;
- **fromID**, **fromIP** et **fromPort** sont l'ID, l'adresse IP et le port du pair à l'origine du message ;
- **succID**, **succIP** et **succPort** désignent de même l'ID, l'IP et le port du successeur du pair à l'origine du message.
- Un message possède également des champs **ZERO** et **ZERO2** inutilisés pour l'instant, et mis à zéro.

On décrit précisément le protocole pour les messages PING, PING_REP et JOIN. À vous de compléter `Peer.java` pour implanter ce protocole, de façon à ce que vos pairs puissent rejoindre un réseau utilisant ce protocole.
- PING sert à rechercher qui est en charge d'un identifiant. Le champ contentID contient alors cet identifiant.
- PING_REP sert à répondre à un PING. Un nœud qui reçoit un PING pour un identifiant dont il est en charge doit répondre un message PING_REP dans lequel contentID 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).
- Lorsqu'un nouveau pair veut rejoindre le réseau, il commence par rechercher le pair x dont il doit devenir le successeur à l'aide de messages PING. Il envoie alors un message JOIN à x, avec contentID=x.

Premier nœud La commande `java Peer new 1024` permettra par exemple de créer le premier nœud d'un réseau sur le port 1024. Le premier nœud du réseau est seul et son successeur est lui même.

Nouveau nœud 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.

Implantation et tests 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 la même adresse IP, mais des ports différents, par exemple 1024, 1025, 1026,...)

2 Publier et retrouver une donnée

Rajoutez une table de hachage (`java.util.Hashtable`) dans chaque membre du réseau. En vous inspirant du routage des messages PING, déterminez un protocole pour permettre à nœud de publier une chaîne de caractères associée à son identifiant à l'aide d'un message PUBLISH. De même, implantez la gestion des messages GET et GET_REP qui permettent de rechercher une chaîne de caractères publiée et l'identifiant du nœud qui l'a publié.

Pour ajouter une chaîne de caractères à un message, vous pouvez par exemple utiliser les méthodes `Message.setData` et `Message.getData` (la chaîne est alors stockée à la suite des champs du message, et la longueur de la chaîne de caractères est dans le champ ZERO2).

3 Tester un réseau

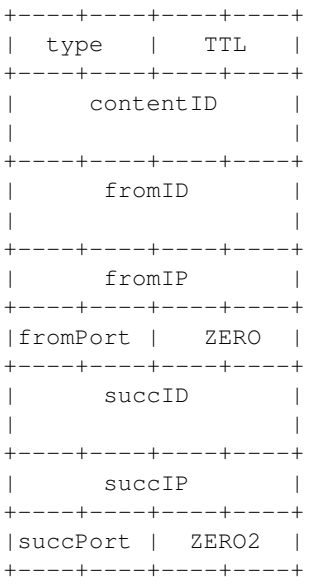
Écrire une classe `Visitor` similaire à `Peer` pour envoyer des messages dans un réseau pair à pair dont on connaît un contact (c'est à dire son adresse IP et son port).

La commande `java Visitor ping IP port ID` permettra notamment de lancer un PING sur ID 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.

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

A Utilisation de Message.java

Chaque message a l'en-tête suivante (une ligne correspond a quatre octets) :



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 contentID, PeerInfo from, PeerInfo succ)` permet de créer un message quelconque.
- 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.
- `void setFromID (long i)` permet de mettre `i` 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 méthodes `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 d'octet. 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/.