Réseaux pair à pair – suite

Ce TD est la suite du TD 4. On repart du réseau pair à pair minimal construit à la dernière séance, et on l'enrichit pour fournir un routage efficace et une tolérance aux pannes minimale. Vous pouvez soit repartir de votre travail (si vous avez terminé la partie 1 du TD 4), soit utiliser les fichiers `Peer.java` *et* `Visitor.java` *disponibles sur le site web des TDs [1]. On rappelle que la documentation de java est disponible en ligne [2].*

1 Routage efficace

Pour router efficacement les messages PING, PUBLISH et GET, il faut rajouter des raccourcis à la structure d'anneau qui sert de base au routage. Pour cela, chaque nœud devra connaître non seulement son successeur, mais également une liste de voisins longs. Un nœud d'identifiant x va stocker et mettre à jour les informations (ID, IP, port) sur les nœuds gérants les IDs $v_0 = x + 1$, $v_1 = x + 2$, $v_2 = x + 4$, $v_3 = x + 2^3$,..., $v_{m-1} = x + 2^{(m-1)}$. On rappelle qu'un nœud x gère un identifiant y si y est dans l'intervalle entre x et succ(x) :

$$x \leqslant y < \text{succ}(x)$$

Notez que les premiers ID de cette liste seront gérés par le nœud x lui-même. Il faut donc également calculer l'indice P tel que v_P est le premier voisin long différent du nœud courant.

À vous de rajouter les structures de données nécessaires au stockage de ces nœuds, et aussi de faire en sorte qu'un nœud découvre ses voisins lorsqu'il rejoint le réseau. Ces voisins longs peuvent devenir obsolètes par la suite si d'autres nœuds rejoignent le réseau, mais on garde leur mise à jour pour une future amélioration. Il vous faut ensuite modifier le routage pour utiliser ces liens longs. Comment choisir le meilleur lien pour router efficacement sans dépasser le nœud cible ?

Je vous conseille de mettre une option pour pouvoir router des messages soit avec le mécanisme des liens longs, soit uniquement par les successeurs. Cela vous permettra de comparer les deux approches.

Enfin, comment établir expérimentalement que le routage obtenu avec ces liens longs est plus efficace qu'avant ? Calculez le temps de routage attendu (en nombre de sauts) en fonction de la taille n du réseau (nombre de nœud), et vérifiez ce résultat sur votre code avec les tests appropriés.

2 Gérer les départs de nœuds

Pour que le réseau pair à pair fonctionne toujours, il est primordial que l'anneau formé par les liens de nœud à successeur ne soit jamais brisé. Pour partir, un nœud d'ID x doit donc d'abord informer son prédécesseur (c'est à dire le nœud dont il est le successeur) de son départ imminent. Pour ceci, il vous faudra probablement créer un nouveau type de message, et trouver un moyen pour qu'il soit routé exactement au prédécesseur (si possible sans rajouter de liens). Détaillez le protocole et codez-le.

Les informations qui était stockées par le nœud qui quitte le réseau sont perdues. Pour faire persister une donnée publiée dans le réseau, il faudrait la republier régulièrement. On

ne traitera pas la gestion des données pour se concentrer sur les problématiques directement liées au routage.

Pour signifier à un nœud qu'il doit quitter le réseau, on créera de plus un type de message KILL. Quand un nœud reçoit un message de type KILL avec son ID comme `contentID`, il doit quitter le réseau.

3 Les départs de voisins

Il nous faut désormais gérer le départ et les pannes de nos voisins, car un nœud quittant le réseau ne le signale qu'à son prédécesseur, mais pas aux nœuds l'ayant choisi comme voisin long. Pour ce faire, on doit envoyer régulièrement des messages PING afin de s'assurer que nos voisins sont toujours en vie. Pour le cas particulier du successeur, il faudra également s'assurer qu'on peut conserver la structure du réseau même dans le cas de son départ. Quelle information supplémentaire doit-on alors stocker ?

Pour gérer des actions régulières, il faut rendre la *socket* non bloquante en réception (méthode `DatagramSocket.setSoTimeout()`). Il faudra alors s'assurer qu'on envoie un PING régulièrement (fréquence à définir) et que la réponse PING_REP est reçue avant un certain temps (*time-out* à définir).

Pour aller plus loin, essayez de rendre votre protocole de réseau pair à pair robuste vis à vis de certaines situations incongrues qui pourrait se produire. On peut par exemple imaginer qu'un nouveau nœud émet un PING pour s'insérer dans l'intervalle géré par un nœud n qui lui renvoie un PING_REP, et juste à ce moment là (avant le JOIN), le successeur de n décide de partir. Identifiez et réglez ce bug possible. Il est primordial qu'un nœud ait toujours un successeur valide. En dernier recours, si un nœud n'a plus de successeur, il peut essayer de prendre un voisin comme successeur, voire n'importe quel nœud dont il peut avoir connaissance et dont l'ID lui semble le plus proche possible du sien.

Une approche générale consiste à supposer que de toute manière il risque d'arriver des situations complexes qui risquent de détruire l'anneau. On peut donc essayer d'introduire des mécanismes pour réparer l'anneau si on détecte un problème. Par exemple, on peut faire en sorte qu'un nœud puisse détecter s'il y a deux nœuds qui le prennent pour successeur. Dans ce cas il pourra envoyer un message à l'un pour qu'il prenne l'autre comme successeur (par exemple un faux message JOIN de l'autre nœud).

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] Sun Microsystem. Java™ 2 platform, standard edition, v 1.4.2 API specification. http://java.sun.com/j2se/1.4.2/docs/api/.