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 le fichier `Peer.java` disponible sur le site web des TDs [1]. On rappelle que la documentation de java est disponible sur `http://java.sun.com/j2se/1.4.2/docs/api/`.

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 va tenir à jour une table de voisins. Si x est son ID, le nœud va stocker les informations d'ID, d'adresse IP et de port des nœuds gérants les IDs $x+1$, $x+2$, $x+4$, $x+2^3$,..., $x+2^{(m-1)}$. Les premiers IDs sont gérés par le nœud lui-même. On pourra utiliser l'indice indicePremVois du premier indice tel que $x + 2^{\text{indicePremVois}}$ n'est pas géré par le nœud lui-même. Les nœuds gérant les IDs $x + 2^k$ sont appelés voisins. Le nœud gérant $x + 2^k$ est celui dont l'intervalle contient cet ID. Les voisins sont découverts grâce à des messages PING. On pourra stocker ces voisins dans un tableau de Peer de longueur m. Seuls les indices supérieurs à indicePremVois correspondront à de réels voisins.

Pour router un message vers le nœud en charge de destID, un nœud doit maintenant le passer à son voisin de plus grand ID inférieur à destID. Le choix des voisins assure qu'un message transitera en moyenne par $O(logN)$ nœuds (où N est le nombre de nœuds dans le réseau pair à pair). Cependant, si destID est inférieur à l'ID du premier voisin du nœud, on applique la règle du TD précédent.

Une fois qu'un nœud s'est inséré dans l'anneau, il émet un PING pour chacun des IDs définissant ses voisins (sauf pour les IDs inférieurs à celui de son successeur). Si des PING_REP sont perdus, on n'essayera pas de réémettre les PINGs correspondants, mais il faudra garder la table des voisins cohérente de sorte que le routage fonctionne toujours. Remarquons que l'insertion de nouveaux nœuds par la suite peut rendre la table de voisinage d'un nœud obsolète mais ne remettra pas en cause la validité du routage. (L'efficacité peut toutefois en pâtir.)

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 cela, créer un nouveau type de message : LEAVE (codé par la valeur 7). Le champ destID est mis à $x - 1$ et le message est routé comme un PING. Le nœud est donc assuré que son message va arriver à son prédécesseur même s'il ne le connaît pas. Le prédécesseur, quand il reçoit le message, prend alors le successeur du nœud comme nouveau successeur.

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 plus 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 destID, il doit quitter le réseau. Pour éviter des attaques malveillantes, vous pouvez prendre un nombre secret plus grand que 256 pour coder ce type de message dans le champ type.

3 Les départs de voisins

Pour gérer le départ et les pannes de ses voisins, un nœud doit "pinguer" régulièrement les IDs définissant ses voisins. Pour ceci, un nœud devra envoyer un PING à chaque voisin toutes les PING_VOISIN_PERIOD=3 secondes. Si aucun PING_REP n'a été reçu d'un voisin dans PING_VOISIN_TIMEOUT=9 dernières secondes, le voisin est considéré comme parti. Pour le cas particulier du successeur, il faudra toujours stocker le successeur de celui-ci pour pouvoir réagir à la disparition du successeur.

Pour gérer ces actions régulières, il faut rendre la socket non bloquante en réception (méthode `DatagramSocket.setSoTimeout()`). Pour chaque voisin et pour le successeur, on stockera la date du dernier envoi d'un PING vers lui et la date de réception d'un PING_REP de sa part (utiliser `java.util.Date.getTime()`). A chaque itération de `run`, il faut comparer toutes ces dates à l'heure courante pour voir s'il ne faut pas envoyer un PING ou supprimer un voisin.

Pour aller plus loin, essayez de rendre votre protocole de réseau pair à pair robuste vis à vis de toutes les 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. Identifier et régler 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/.

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