Mathias CROMBEZ, Ramzi RAIS, Yongzhi ZHANG
Jeudi 23 Décembre 2010

Projet Systèmes Répartis

Contruction d'une Table de Hachage Distribuée

Présentation

Le but de ce projet est de nous confronté au problème des systèmes répartis à travers un exemple relativement abordable. Il s'agit de la construction d'une table de hachage contenant des couples clé/valeur qui est situé sur un ou plusieurs serveurs et parcouru par un ou plusieurs clients.

Pour cela, on va faire un groupe de serveurs qui se partagent plus ou moins équitablement la table pour améliorer la performance, et des clients qui se connectent à un serveur afin d'effectuer des opérations :

PUT permet de rajouter un couple dans la DHT.

GET permet d'obtenir la valeur associée à une clé.

REMOVE permet de supprimer un couple de la DHT.

STATE permet de lister le contenu d'un serveur et d'obtenir celui qui le suit.

QUIT permet de faire quitter le serveur proprement.

Ces primitives ne sont pas les seuls car nous en rajoutons d'autres pour simplifier le fonctionnement.

Chaque serveur s'occupe d'une partie de la DHT, qui est défini par un entier "h" qui donne la première clé dont s'occupe le serveur, et la taille de cette partie de la DHT.

Pour la connexion réseau, le choix a été fixé sur les sockets qui permettent de communiquer facilement pour un couple serveur/client et, pour le serveur, d'isoler les communications de chaque client.

Les choix d'implémentation

Contrairement à ce qui été demandé, nous avons qu'une version du programme car nous avons rapidement choisi le module de l'équilibre de charge.

La méthode choisi est un parcours de la liste des serveurs par le nouveaux serveurs par des connexions et déconnexions qui permettent d'obtenir la table la plus chargé (DHT plus grande) et de la découpé en 2. Le nouveau prends la deuxième moitié. Cela se fait uniquement à chaque arrivée d'un nouveau serveur.

Un serveur crée un "pthread" à chaque arrivée de client et lui envoi la socket qui lui est dédié. Ainsi il est quasiment indépendant du programme principal. La routine permet de recevoir le type du message et d'appliquer la fonction.

Le message est composé de plusieurs parties :

- L'origine qui est utilisé uniquement pour le premier message pour savoir si cela vient d'un client ou d'un serveur et d'appliquer une routine différente selon le cas.
- Le type du message qui est l'une des primitives définies (PUT, GET ...), chaque message commence par lui (sauf le premier)
- Les données qui sont des uint32_t, uint64_t, des octets ou des chaînes de caractères (précédés de leur taille)

– L'octet de synchro qui est interne dans les fonctions d'envoi et de réception (chaque envoi à une réception d'abords et chaque réception un envoi).

Les principales primitives rajoutées sont CONNECT (qui est quasiment inutile) et DISCONNECT (qui permet de libérer les allocations de structure dédié au client au niveau du serveur).

Les problèmes et leur solution

Le principal problème est ce que nous pensions être de la synchronisation. En effet, certaines données étaient envoyés à des moments improbables et nous recevions des données sur des sockets qui étaient censées être fermé et après une deconnexion de l'un des clients, les clients non connectés se débloquaient et quitter sans bien appliqué la connexion et sans se déconnecter.

Après de nombreuses réflexions, nous avons plus ou moins localisé le problème : plusieurs "pthread" ont la même socket dédié (même si la socket donnée est la bonne). En fait, quand on crée une variable "sockClient" pour passer la socket à la routine, nous envoyons une adresse sur la valeur qui peut être écrasé par l'arrivée d'un autre client avant qu'on la récupère. Ce qui fait que plusieurs "pthread" avait la même socket. Après avoir corrigé ce problème en envoyant non pas une adresse mais la valeur (la socket et l'adresse sont tout deux des entiers), nous nous sommes aperçu que la synchronisation rajouté était inutile c'est-à-dire l'octet de synchronisation, nous l'avons donc supprimé (le code a été commenté).

Ce bug nous a coûté énormément en ressources et s'est surtout compliqué par un manque d'informations sur sa localisation car son apparition provoqué un problème de synchronisation sur toute la suite du programme.

Conclusion

Pour conclure, le projet était intéressant car il permettait de joindre les sockets et les pthreads et de créer son propre protocole de communication (de haut niveau).

Cependant, il est particulièrement difficile à tester et debuggé car les cas sont différents selon le nombre de clients qui se connectent à un serveur, la façon de se connecter (séquentielle ou parallèle), le nombre de serveur, les opérations utilisés ... En effet, le système possède des limitations (nombre de sockets ouvertes, de pthread, de connexions en attente), ainsi les tests ne permettent pas de mettre en jeu des grands nombres de clients (de façon séquentiel, le nombre est limité à environ 350 sur machine personnelle et 1000 sur ensisun).

De plus, les tests réels (avec adresse internet entre deux ou plusieurs machines) sont difficiles à effectuer pour des problèmes matériels tels que les pare-feux des réseaux hôtes qui refusent les ports utilisés en connexions entrantes. Ainsi la majorité des tests se font par boucle hôte ou "localhost" (adresse : "127.0.0.1") : les tests peuvent donc se montrer insuffisant.