Université de Strasbourg
UFR Mathématiques et Informatique

Licence 3 Informatique
Année 2017/2018

Réseaux et Protocoles

Projet : table de hachage distribuée

Objectif : appréhender de manière simple ce qu'il se fait dans des outils de la vie quotidienne en recréant par nous-mêmes une version simplifiée.

Le projet est de créer une bibliothèque de gestion d'une table de hachage distribuée (DHT). Cette bibliothèque sera accompagnée d'un lot de programmes de test, afin de s'assurer du bon développement de chaque partie, sans régression. Vous ferez les tests que vous jugerez pertinents (à justifier). Les différents programmes ainsi créés devront être documentés (voir section 2).

Modalités et consignes

- Les exercices sont à faire en groupe de 2 personnes.
- Le rendu doit se faire dans une archive au format `nom1-nom2_projet-rp.tar.gz`. L'archive devra être déposée sur la plateforme Moodle.
- L'évaluation du travail se fera au dernier TP où vous devrez présenter votre travail.
- Le format des messages échangés n'est pas imposé. Une piste est disponible en annexe.
- Tous les programmes doivent être documentés (une page de manuel par programme).
- Le code est en langage C (tests en shell), doit être lisible et commenté.
- Tous les programmes se baseront exclusivement sur UDP.
- Les programmes doivent fonctionner en IPv6 (et IPv4 si vous avez le temps).
- Justifiez vos choix là où c'est pertinent (ex : choix d'implémentation dans le code).

1 Table de hachage distribuée

Une table de hachage distribuée (DHT) est un élément qui permet de s'affranchir d'un unique serveur qui aurait toute l'information et dont tout le monde serait dépendant. Une DHT permet donc de décentraliser une information.
Par exemple, lorsque vous téléchargez des fichiers torrents, une DHT est présente pour stocker une liste de hash (un résumé des fichiers que vous partagez) avec des valeurs associées (les IP des personnes qui possèdent ou veulent ces fichiers). Lorsque vous souhaitez un fichier (disons une image Linux), vous téléchargez un fichier « torrent » qui possède un hash. Ce hash est utilisé pour demander à une DHT la liste des IP qui ont le fichier souhaité. Finalement, lorsque vous avez la liste des IP des gens qui possèdent le film, vous leur demander le contenu.
Nous allons implémenter une version très simple d'une DHT.

1.1 Table de hachage : premiers pas

Le premier exercice consiste à faire un serveur unique gérant une table de hachage. Pour cela, il doit déjà répondre à deux types de requêtes : **GET** et **PUT**.

Le message PUT permet d'envoyer un hash et une valeur associée, et GET permet de la récupérer. Créez un serveur utilisable comme ceci `./serveur IP PORT`, exemple :

```
$ ./server ::1 9000
```

Le programme doit aussi comprendre les noms de domaine, exemple :

```
$ ./server localhost 9000
```

Puis faites un client qui saura envoyer ou demander un hash, `./client IP PORT COMMANDE HASH [IP]`, exemple :

```
$ ./client localhost 9000 put 394c8a052d ::1
$ ./client localhost 9000 get 394c8a052d
IP disponibles pour le hash 394c8a052d :
::1
```

Un hash a une taille d'au moins 65 octets.

1.2 Connexion et déconnexion entre deux serveurs

Nous avons désormais un serveur gérant une table de hash. Notre objectif est maintenant de distribuer cette table de hash entre plusieurs serveurs pour améliorer la redondance, commençons avec 2 serveurs. Voici le déroulé des évènements :

1. serveurA démarre
2. serveurB se connecte à serveurA
3. serveurB envoi ses hash à serveurA
4. quand serveurA reçoit un hash d'un client, il le transfère à serveurB (et inversement)
5. un des serveurs se déconnecte (= dire à l'autre qu'on s'arrête)

1.3 Keep Alive entre serveurs

Nos deux serveurs savent désormais se connecter, partager des hash et leur valeur (à la connexion et au fil de l'eau) puis se déconnecter. Le problème c'est qu'un de ces serveur peut s'arrêter sans prévenir l'autre, comme par exemple quand il lui arrive une erreur de segmentation mémoire (`n'est-ce pas ?;)`). L'idée ici est que les deux serveurs s'envoient périodiquement un message dit `keep alive` pour maintenir leur connexion. Si l'un des deux ne répond pas depuis un certain temps, alors il est considéré déconnecté.

1.4 Obsolescence des données

Cette fois-ci on considère les données comme périssables : un serveur doit supprimer un hash de sa table s'il devient obsolète. Un hash est considéré obsolète s'il n'a pas reçu de message PUT depuis plus de 30 secondes.

Attention : n'oubliez pas qu'il y a deux serveurs, et qu'un client n'en contacte qu'un.

1.5 DHT à plus de 2 serveurs

À ce stade nous avons 2 serveurs qui savent se connecter entre eux, s'échanger des hash et se déconnecter, et si un serveur s'arrête de fonctionner normalement il est considéré déconnecté. Généralisez ceci à de multiples serveurs. Faites ceci comme bon vous semble, puis justifiez votre choix.

1.6 Des pistes d'amélioration

Donnez des pistes d'amélioration de ces programmes qui vont dans le sens de la simplification, de l'amélioration des performances (gain en empreinte mémoire, nombre ou taille de messages...), de la résistance à une coupure réseau ou un crash d'un des programmes...et implémentez ! Points bonus à la clé.

Bon courage !

2 Consignes supplémentaires et conseils

Tout d'abord, vos programmes peuvent être vérifiés par Wireshark, utilisez cet outil autant que possible pour être sûr du contenu de vos messages.

Sur un système d'exploitation, un programme ne doit pas être fourni sans un minimum d'explications. En guise d'entraînement, faites des pages de manuel pour chaque programme qui vous est demandé. Documentez également le format des messages qui sont échangés (dans un README).

2.1 Écrire une page de manuel

Les pages de manuel de vos programmes comporteront au minimum :
- le nom du programme avec une petite description de son utilité
- le nom de l'auteur de la page de manuel
- la date de création de la page de manuel
- une description claire des différentes options possibles
- le synopsis pour savoir comment utiliser le programme
- les différents numéros de retour
- les erreurs et limitations de vos programmes
- et tout autre élément qui vous semblera pertinent

Faites `man 7 mdoc` pour apprendre à créer des pages de manuel. Vous pouvez prendre exemple sur les pages de manuel déjà présentes sur votre système (dans `/usr/share/man/man1/` par exemple). Le nom de votre page de manuel sera le nom de votre programme suffixé par « .1 », le numéro « 1 » correspondant à la section dédiée aux manuels d'utilisation d'un programme (`man man` pour vous rafraîchir la mémoire).

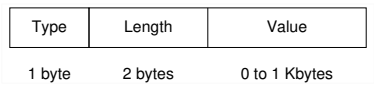
2.2 La lisibilité

Le code doit être lisible et commenté. Un code non indenté vaut 0. Un code non commenté vaut 0. Les commentaires doivent être pertinents, et montrer que vous comprenez ce que vous faites. Un saut de ligne de temps en temps ne peut pas faire de mal. Vous êtes libre d'organiser votre code comme bon vous semble, s'il vous paraît judicieux de séparer le code en plusieurs fichiers pour améliorer la lisibilité ou pour rendre votre code plus modulaire, n'hésitez pas ! Vous avez explicitement le droit de rendre vos sorties d'exécution plus jolies si vous le souhaitez, tant que cela ne nuit pas à la lisibilité du code et que c'est pertinent.

2.3 Trouver de l'aide

Les pages de manuel peuvent vous aider, de même que de nombreux sites. N'utilisez un moteur de recherche que si vous ne trouvez pas dans le manuel, vous en apprendrez plus ! ;) Si vous avez des questions, n'hésitez pas à venir soliciter vos enseignants.

3 Annexes - Format des échanges de données



Format Type Length Value, simple et efficace, vous pouvez vous en inspirer