Projet - Réseaux 1
Réalisation d'un Proxy de cache en C

UMONS
Faculté des Sciences
BAC 2 Science-Informatique.

Delplanque Julien
Charlier Maximilien

Année académique
2013-2014

i Introduction

Il nous a été demandé d'implémenter un serveur proxy de cache en `C` supportant le protocole `HTTP 1.1`. Un client lui adressera des requêtes de type `GET` (une requête `GET` consiste à demander des données à un serveur : *page web*, *image*). Le serveur proxy, devra vérifier si l'objet de la requête est dans son cache, si ce n'est pas le cas, il le téléchargera, le mettra en mémoire et le renverra au client. Si l'objet est dans le cache, il demandera au serveur destination si l'objet a été modifié depuis la mise en cache, si oui, il le retéléchargera, le remplacera dans le cache et le renverra au client, si non, il renverra l'objet présent dans son cache.
Le serveur devra gérer plusieurs clients simultanément.

ii Fonctionalités

- **Caching/Sauvegarde du cache (proxy.c)** :
  Lorsqu'une requête est reçue par le proxy, celui-ci vérifie si le fichier existe en local, si c'est le cas, il vérifie si ce fichier est à jour ou non en envoyant une requête au serveur hébergeant cette page. Si le fichier est à jour, le proxy envoie le fichier stocké localement au client, si le fichier est périmé, il le redemande au serveur, le transmet au client et remplace l'ancienne sauvegarde de ce ficher par la nouvelle. Si le fichier n'existe pas en local, celui-ci est téléchargé par le proxy, transmis au client et sauvé dans le cache du proxy. S'il n'y a pas de connection internet ou que l'hôte n'existe pas, le proxy envoie un message d'erreur au client. Le dossier contenant le buffer s'appelle `.buffer` et se situe dans le dossier courant.
- **Pipelining (client.c & client-test.c)** :
  Le pipelining est géré par le client de test, l'envoi des requêtes doit être espacés d'environs 1/5 de secondes pour permettre à la fonction `send()` d'envoyer correctement les données (un délais plus court entraine la disparition de paquet). Les réponses envoyées au client sont donc observées à l'aide d'un descripteur de fichier (de manière non bloquante).
  Nous avons rencontré des problèmes concernant la fin de transmissions des paquets, en effet, lors de la réception des données concernant les 3 sites de tests nous avons constaté que le descripteur de fichier recevait les données des deux premiers en un bloc, puis un second bloc pour `stackoverflow.com` (on entend par là le fait qu'il n'y ai pas de fin de transmission après avoir reçu les données de `perdu.com` étant donnée que les données de `google.be` suivent juste après).
- **Persistance (client.c & proxy.c)** :
  La persitance est gérée par le proxy, ainsi que la non persistance, on peut choisir de lancer le client en mode persitant ou non via modification d'une macro dans le code (voir rubrique `Utilisation`).
- **Connection concurrents (proxy.c)** :
  Le proxy gère les connections multiples grâce au système de descripteurs de fichiers.
- **Communication client-proxy (client.c & proxy.c)** :
  Le client est configurable, on peut choisir son port de communication et son type de communication (persistant ou non), un client de test est disponible, et implémente le piplining.
- **Communication proxy-server (proxy.c)** :
  Le proxy communique avec les serveurs hébergeant les fichiers requis de façon non persitante. Les réponses de ces serveurs sont directement transmises au client et sauvées en mode ajout au fichier dans le cache. C'est donc le serveur hébergeur qui ferme la connection avec le proxy.
  Dans le cas où la connection avec le serveur n'est pas possible et que le fichier demandé

n'est pas en cache, le proxy envoie un message explicatif au client.

iii Utilisation

Un fichier bash contenant les fonctions utilisées lors des tests est livré avec le projet, celui-ci est nommé `build.sh`. La commande : `$ source build.sh` permet de charger ces fonctions.
Parmi ces fonctions, celles permettant d'executer le projet sont :

- **$ build** Compile le proxy et le client
- **$ run_proxy [port]** Lance le proxy pour écouter le port passé en paramètre. Si aucun port n'est passée, le proxy écoute sur le port 4242.
- **$ run_client proxyip [proxyport]** Lance un client qui se connecte au proxy sur le port proxyport. Si le port n'est pas spécifié, c'est le port 4242 qui sera utilisé.
  Surveille l'entrée clavier pour envoyer des requettes au proxy.
- **$ run_test proxyip [proxyport]** Lance un client qui se connecte au proxy sur le port proxyport. Si le port n'est pas spécifié, c'est le port 4242 qui sera utilisé.
  Lance des requêtes automatiquement au proxy.

Remarque : Il est possible de modifier le type de connection entre le client et le serveur en modifiant la definition de la macro `PERSISTANT` définie dans le header `src/client-tools.h`, (sans oublier de recompiler le code).

iv Répartition des tâches

Julien Delplanque a développé le serveur proxy, Maximilien Charlier a développé le client et le client de test.

v Remarques

Voici les outils avec lesquels le proxy a été développé :

Nom	`OS`	Version `GCC`	Éditeur de texte	Version `GIT`
Julien Delplanque	`Archlinux` 3.13.8	4.8.2	`Sublime text 2`	1.7.2.5
Maximilien Charlien	`Ubuntu` 13.4	4.7.3	`Sublime text 3`	1.7.2.5

Remarque : Un outil qui a été utile pour faire des tests hors ligne et voir quelles requêtes le serveur reçoit du proxy est la commande /bin/python2 -m SimpleHTTPServer. Cette commande crée un serveur http observant le dossier courant. Ce serveur écoute sur le port local 8000 et affiche, dans le terminal où il est lancé, la requête http reçue.

vi Source

- *RFC 2616 HTTP 1.1*
- *Information sur le Pipeline*
- *Les sockets sur developpez.com*
- *Taille requette HTTP et longeur max d'une URL*
- *SimpleHTTPServer*