Rapport projet Un server HTTP

Lavigne Alexandre - Vallade Vincent

04/01/2017

Introduction

Ce rapport décrit notre raisonnement et les solutions apportées pour répondre aux exercices donnés par le projet de construction d'un mini-serveur HTTP pour l'UE Programation Répartie.

Ce serveur suit le protocole HTTP 1.1. par ailleurs, il n'accepte que la requette de type GET (dont le détail sera décrit par la suite)

Le projet est élaboré en langage C et la constante : XOPEN_SOURCE est définit avec la valeur : 700

1 Structure du Server

Le serveur HTTP prend en argument à son lancement :

— un port d'écoute

— le nombre de clients que l'on peut accepter à la fois

— un nombre d'octets qu'un client peut demander au maximum par minute

Pour cette première partie nous avons essayé de concevoir un serveur HTTP en anticipant les questions 2 et 3.

Nous avons conçu la partie principale du serveur pour que sa seule tâche soit de recevoir les requêtes des clients sur le socket d'écoute et de lancer un thread tout de suite après.

Le serveur établit donc toute l'initialisation puis lance une boucle infinie durant laquelle il écoute pour une connection de type TCP sur le port donné.

Le serveur redéfinit le *handler* pour le signal SIGINT afin de fermer le socket, détruire le sémaphore, et toutes autres ressources allouées, ce qui nous permet de fermer le serveur proprement via un CTRL+C. Pour faciliter le transfert d'informations entre le serveur principal et le thread qui traite avec un client, nous avons créer une structure "client" contenant les informations suivantes :

— le socket de communication avec le client

— le descripteur de fichier de journalisation

— la structure sockaddr_in du client

— le semaphore pour gérer l'accès en concurrence du fichier de journalisation

La suite du document décrit le code du thread contenut dans le fichier "requete.c"

Le thread lit les caractères un par un sur le socket de communication jusqu'a trouver un caractère '\r\n'. Ensuite il extrait le premier mot pour vérifier qu'il corespond bien à 'GET'. Il extrait le deuxième mot pour vérifier le chemin que le client demande.

Sous unix pour demander une ressource à partir du dossier courant il suffit que le nom de cette ressource ne commence pas par '/', nous vérifions alors sa présence et le supprimons du chemin s'il existe. Dans une requete HTTP, le client client demande des ressources situé à partir de la racine du server web donc qui commence par '/'. Nous vérifions si la ressource existe et si le server à les droits en lecture dessus.

Ensuite le thread vérifie le type mime du fichier grâce à la fonction "get_mime" permetant de parser le fichier '/etc/mime.types' et de trouver si un mime-type existe, si oui alors il l'envoie au serveur sinon le serveur envoie 'text/plain', de cette manière nous protégeons le client contre toute exécution de code alors que le type du fichier est indéterminé.

Pour rechercher et vérifier le mime-type du ficheir demandé, nous utilisons une expression régulière en C. cette expression cherche l'extention du fichier sur chaque ligne du fichier mime-type sans lire la première colone (donc uniquement en comparant les colones de 2 jusqu'a la fin de la ligne).

Si la fonction ne trouve pas mime-type, elle renvoie une chaine vide. Dans ce dernier cas le thread sait que le mime-type n'a pas été trouvé.

Une fois toutes les vérifications effectuées le thread envoie sa réponse au serveur, donc les headers contenant le code de retour, puis le type mime du fichier. Ensuite une fonction dédiée ouvre, lit et envoie le fichier au client. Une vérification des erreurs est éffectuée durant l'exécution et renvoie s'il le faut un code de retour 404 ou 403 ou 500 (voir questions suivantes)

2 Journalisation

Après l'envoie d'une réponse à un client un appel à la méthode write_log est effectuée. On écrit dans une chaine de caractères à l'aide de sprintf toutes les informations que l'on veut sauvegarder dans le log. La date de fin de traitement de la requête est récupérée grâce à un appel à la méthode get_time(char *result) qui récupère la date grâce à la fonction localtime et la transmet dans la chaîne passée en argument. Nous utilisons un sémaphore pour gérer l'accès en concurrence au fichier de journalisation. Ce fichier est ouvert qu'une seule fois au lancement du serveur et le descripteur de ce fichier est transmis aux threads par la structure "client".

3 Fichier exécutable

Dans la méthode "process_request", nous vérifions le type du fichier demandé par le serveur grace à la méthode "check_file", si ce fichier est accessible en lecture et executable : alors nous créons un processus fils via la méthode fork()

et appelons la methode “execl” qui permet d’écraser le code de ce fils avec celui de l’exécutable.

Avant de lancer la methode “execl” nous prenons soin de rediriger la sortie standards du fils vers un fichier temporaire. Cette solution nous permet de suivre l’exécution du fils, si celui ci ce termine normalement alors nous n’avons qu’a lire et envoyer le fichier temporaire sur le socket, ainsi nous connaissons la taille du résultat à envoyer.

En cas d’erreur ou si le fils ne s’est pas terminé au bout de 10 secondes, nous renvoyons au serveur le code “500 Internal Error”.

4 Requêtes persistentes

Le thread créé par le serveur principal pour gérer les connections avec le client va se contenter de lire tous les headers, puis va creer un sous thread pour chaque requêtes du client. Une structure “thread_fils” a été créée pour transmettre des informations à ce sous thread.

- counter : l’adresse d’un compteur qui indique le nombre de sous thread terminé
- id : l’id du thread, sa valeur est sa position de création.
- mutex : un mutex pour l’accès au socket du client
- cond : condition sur laquelle les threads s’endorment si ils ne peuvent pas accéder au socket client.
- cli : un pointeur vers une structure client
- get : une chaine de caractères contenant la première ligne lue par le thread de gestion de connection, cette ligne est sensée être la première ligne recu par le serveur (GET ...)

Le sous thread va verifier que la chaine get transmise commence bien par le mot clé “GET” et que le chemin est valide. Puis pour la gestion de la concurrence entre les threads, il va verifier que l’id qu’on lui a transmis est égal au compteur passé dans la structure. Si c’est le cas le thread s’exécute, sinon il s’endort.

à la fin de son exécution le thread réveille tout les autres thread en attente sur la condition, ce qui nous évite les problème de famine.

5 Contrer le déni de services

Pour contrer le déni de service nous lançons un thread juste après le démarage du serveur pour contrôler et vérifier que chaque client à le droit de recevoir au plus N octets.

Ce thread particulier que nous avons surnommé “Vigilante” contient la structure suivante :

- threshold : le seuil qu’un client ne doit pas dépasser.
- mutex : le vérou pour évité les accès mutuels
- clients : une liste chainé contenant les informations de chaque client qui tente de ce connecter au serveur

La liste chainé est composé d'une structure client_data_count :

— track : un tableau de 60 case, chaque case représente une seconde, elle contient le nombre d'octets envoyer à cette instant
— ip : l'addresse ip du client
— flag : un drapeau permetant de savoir si le client est bloqué ou non
— last : la dernière seconde à la quelle le client à effectué une demande
— timeleft : le nombre de secondes restante avant que le client ne soit plus bloqué s'il est bloqué

Ce thread parcours ,chaque seconde (donc toutes les secondes moins le temps de traitement nécessaire pour parcours la liste et vérifier tout les clients), la liste chainé de clients, et met à zero la case du tableau pointé par la seconde actuelle, si jamais un client à fait une demande durant cette seconde, le thread ignore le client et n'efface pas cette case de son tableau.

Lorsqu'un client ce connecte et envoie sa/ses requêtes, le thread traitant avec ce client ajoute le client à la liste chainé s'il n'existe pas encore, puis vérifie que ce client ne soit pas bloqué. Si c'est le cas, alors il répond par un code 403 et ferme la connection. Sinon il traite la demande du client. Juste avant de lui répondre, il fait un appel à la méthode "incremente_size" qui permet d'ajouter cette demande au total que ce client à déjà potentiellement demandé. Si cette demande dépasse le seuil maximum alors le client est banis pour 10 secondes, et recoit un code de retour 403. Ceci tant que le client emmet des requêtes et qu'il est banis.

Pour identifier nos client dans la liste chainé nous utilisont leurs addresse IP sous le type "long", ce qui nous permet d'identifier parfaitement un client. Cette information nous la prenons dans la structure socckaddr_in que le système remplis à chaque nouvelle connection.