Projet Transfert de Fichiers (FTP) Compte-Rendu

Dylan Grousson
Axel Deuleuze-Dordron

Mars 2024

1 Fonctionnement Global

1.1 File Transfer Protocol

Nous dénombrons trois parties principales dans le protocole FTP:

- Connexion au serveur: Le client va en premier temps envoyer une requête de connexion au serveur. Si ce dernier peut la traiter, un processus accepte cette demande.
- Emmision de la requête: Une fois la connexion effectuée, le client va envoyer la taille du nom du fichier qu'il souhaite télécharger, puis le nom de fichier en lui-même. Il va également vérifier si de son côté le fichier n'est pas vide, auquel cas il envoie également la taille du fichier stocké localement (0 octet si ce fichier est vide).
- Réception du fichier: À partir des informations reçues, le serveur va calculer la taille restante à envoyer. Il envoie au client le nombre d'octets qu'il va lui transmettre, puis par paquet de 512 octets maximum le contenu du fichier, qu'il lit dans sa mémoire à partir du point manquant au client. Une fois le fichier entièrement réceptionné, le client peut ré-émettre une requête ou fermer la connexion.

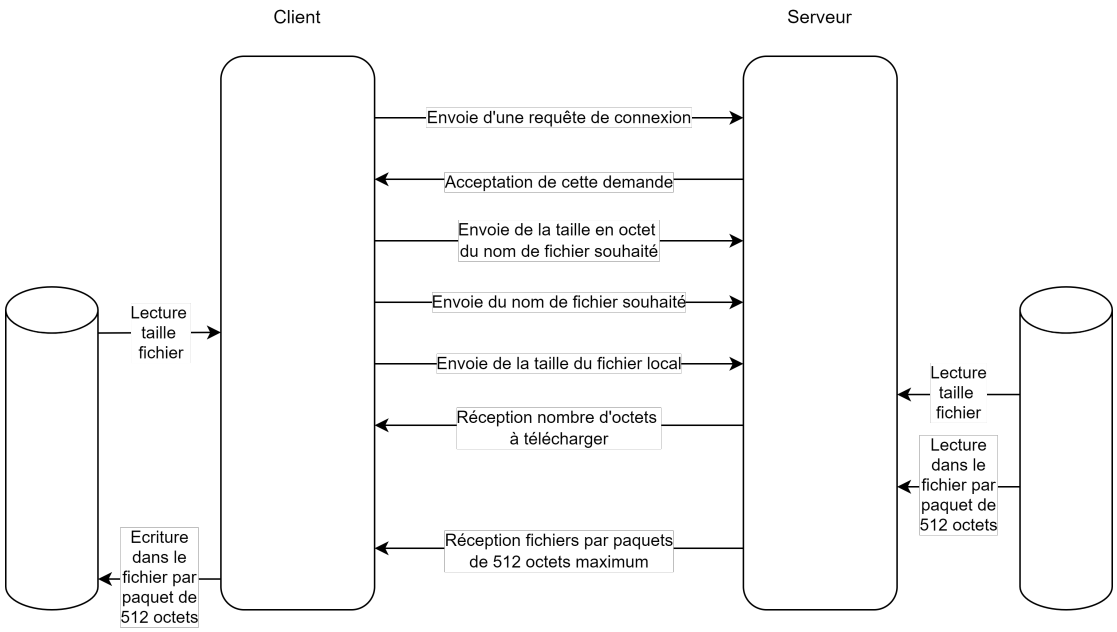

Figure 1: Shéma de fonctionnement du protocole (point de vue du client)

1.2 Description du Fonctionnement

Lorsqu'un client souhaite récupérer un fichier présent dans la base de données de notre serveur, il s'y connecte. Si un processus fils est libre, il accepte cette demande de connexion et ouvre un socket de communication. Le client peut alors entrer un nom de fichier. Si il correspond à un fichier côté serveur, le téléchargement se lance et le client verra s'afficher une barre de progression. Une fois celui-ci terminé, il peut au choix demander un autre fichier ou partir.

Dans le cas où le client rencontre un problème lors du transfert, le terminant abruptement avant d'avoir pu récupérer l'entièreté du fichier, il peut se reconnecter et réitérer sa demande au serveur comme la première fois. Seulement, le transfert reprendra là où il s'était coupé la première fois.

2 Choix d'Implémentation

2.1 Lecture Bufferisée/Non Bufferisée

La bibliothèque csapp offre l'accès à des fonctions de lectures robustes, bufferisées ou non. La lecture des sockets ne demandant pas plus 512 octets et aucun accès mémoire, nous avons choisis d'utiliser des lectures non bufferisées, pour gagner en efficacité. Par contre, pour les accès mémoire, nous utilisons des lectures bufferisées, afin de limiter l'utilisation du processeur.

2.2 Gestion d'Erreur

Toutes les fonctions de csapp ont un wrapper éponyme commençant par une majuscule, s'occupant de la gestion d'erreur. Cependant, nous avons fait le choix de ne pas les utiliser, car ils font un exit en cas d'erreur, ce qui nous empêche de faire une fermeture propre et même de garder notre pool de fils en vie. Aussi, nous nous occupons nous même de la gestion d'erreur, en affichant des messages d'erreurs personnalisés sur stderr. Côté serveur, le fils dans lequel l'erreur survient ne continue pas l'exécution de la requête, il ferme le socket de communication avec le client et retourne dans un état d'acceptation. Côté client, ce n'est pas un problème si le processus en cours meurt, aussi nous faisons simplement un exit.

2.3 Fermeture Propre du Serveur

À la réception d'un `SIGINT`, le père ne fait rien, mais les fils eux ferment les fichiers ouverts et les sockets de communication, leur descripteurs étant stockés. Une fois cela fait, ils meurent. Cela provoque l'envoie d'un signal `SIGCHLD`, qui est géré par `sigchildhander`. Le père `wait` tant qu'il le peut et à chaque `wait` effectué, il décrémente le nombre de fils encore en vie, qui est stockée dans la variable globale `nb_proc_` restant. Si ce nombre tombe à 0, cela signifie que tous les fils sont morts proprement et qu'il n'y a aucun zombie et donc que le père peut à son tour exit.

3 Tests

Afin de pouvoir tester le fonctionnement de notre programme, vous allez avoir besoin de récupérer l'archive que nous vous avons fourni ou bien de cloner notre dépôt git (seule Mme Marangozova y aura accès car notre repository est en privé).

Ensuite, pour pouvoir compiler notre programme vous n'aurez qu'à effectuer la commande make, cela va vous créer l'exécutable `./server` dans le répertoire `server_app` ainsi que `./client` dans `client_app`.

Pour lancer le serveur vous aurez besoin d'effectuer la commande `cd server _app` puis la commande `./server`, sur l'autre machine pour leclient vous aurez besoin d'effectuer un `cd client_app` suivi d'un `./client <adresse_ip>`, dans le cas où vous auriez déjà lancé le serveur, la connexion entre le serveur et le client a été établi et le processus fils du serveur responsable de ce client attend désormais que le client initie la procédure de transmission de fichier (tel que nous vous l'avons décrite plus haut).

Du côté du client, on a plus qu'à entrer le nom d'un fichier que l'on souhaite télécharger depuis le serveur, il s'ensuit toute la procédure de transmission de fichier qui apparaît finalement dans le sous dossier files du côté du client.

Nous vous avons préparé plusieurs fichiers côté serveur que vous retrouverez dans le dossier fichiers.