SR Projet serveur FTP

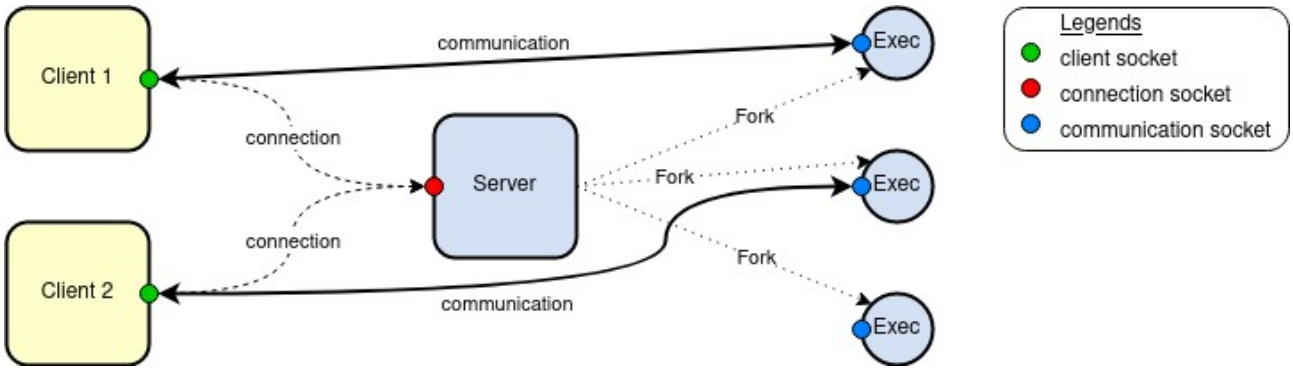
ISAAC--CHASSANDE Sacha, LANNEAU Théo

But du projet :

Le but du projet est de créer un serveur FTP ainsi qu'un moyen de récupérer des fichiers sur celui-ci. Il faut pour ceci créer deux exécutables, un qui va lancer le serveur et permettre des connexions et un autre qui va servir de client permettant de se connecter au serveur FTP et faire des requêtes à ce serveur.

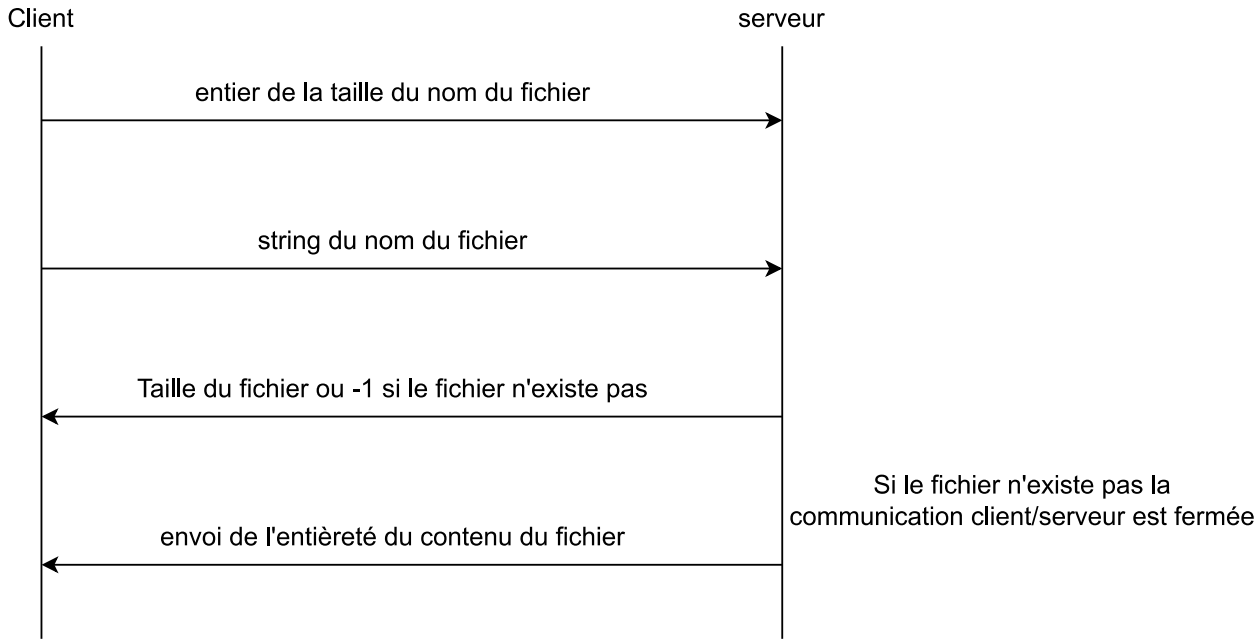
Étape 1 :

Le but de cette étape est de mettre en place un serveur FTP concurrent simple comme illustré sur le schéma ci-dessous. Le serveur aura un certain nombre d'exécutants qui vont accepter les connexions des clients et communiquer avec eux. Si le serveur reçoit un signal SIGINT chaque socket ou fichier ouvert sera fermé et le serveur ainsi que tous ses exécutants seront terminés.



Comme le serveur et le client sont sur la même machine on utilise 2 répertoires différents pour représenter le stockage côté client et côté serveur

Pour que le client et le serveur communiquent il faut mettre en place un protocole de communication. Nous avons mis en place le protocole suivant pour l'étape 1.



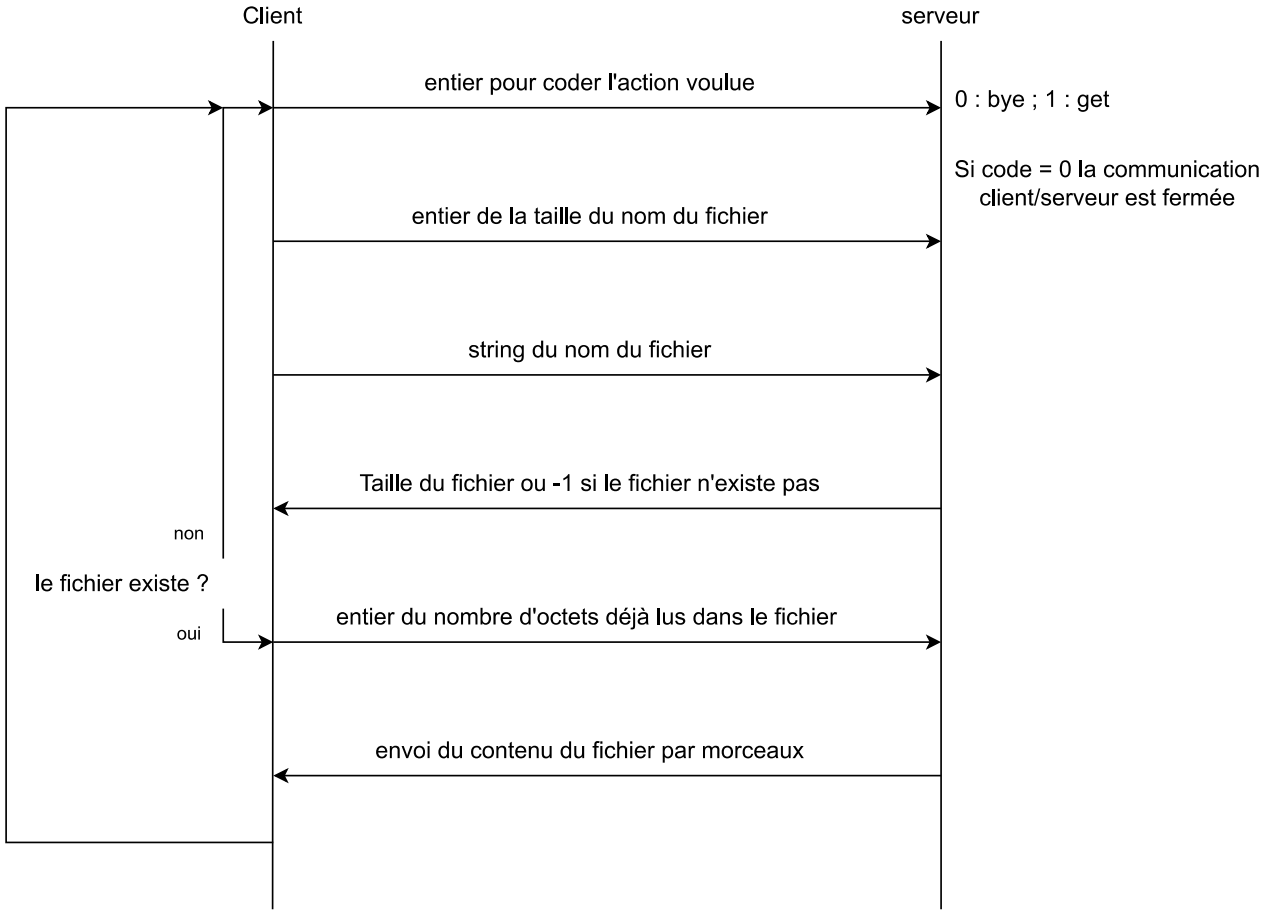
Comme les données qu'on envoi sont des octets le client peut récupérer tout type de fichier (image, texte etc..).

Cependant on ne peut faire qu'une seule requête par lancement du client et on a aucun moyen de reprendre un téléchargement la ou il s'est arrêté si le client se termine de manière impromptue.

Étape 2 :

Nous allons maintenant ajouter la possibilité de faire plusieurs requêtes à la suite. L'envoi de fichiers se fera également par morceaux plutôt qu'entièrement, ce qui permettra de ne pas monopoliser la mémoire lors de l'envoi. Finalement nous allons ajouter un moyen de gérer le cas ou le client bug et la communication se termine sans que le fichier soit téléchargé dans son intégralité.

Nous avons donc décidé de changer le protocole comme vu ci-dessous.



Multiples requêtes et découpage de l'envoi :

Le client envoi maintenant un code représentant l'action qu'il veut effectuer (quitter ou récupérer un fichier). Si le client demande à récupérer un fichier, son contenu est envoyé par morceaux. À la fin du téléchargement le client peut à nouveau faire une requête.

Gestion de crash du client :

Pour gérer les cas où la communication entre le client et le serveur est interrompue au milieu d'un téléchargement nous avons choisi d'utiliser la même fonction get. Lorsque le client demande à récupérer un fichier il envoi d'abord le nombre d'octets déjà téléchargés de celui-ci (soit 0 s'il n'y a pas eu d'interruption de téléchargement) puis le serveur recommence l'envoi du fichier à partir du premier octet non lu.

Pour connaître le nombre d'octets lus dans un fichier nous avons choisi d'utiliser un fichier log qui contient, s'il y a eu une interruption de téléchargement, la commande utilisée, le nom du fichier ainsi que le nombre d'octets déjà lus dans celui-ci. S'il n'y a toujours pas eu d'interruption alors le fichier log n'existe pas. Nous créons ce log à chaque fois que l'on démarre un téléchargement et on le met à jour dès qu'on arrive à télécharger un morceau du fichier. Le log est supprimé si le téléchargement se termine sans problèmes.

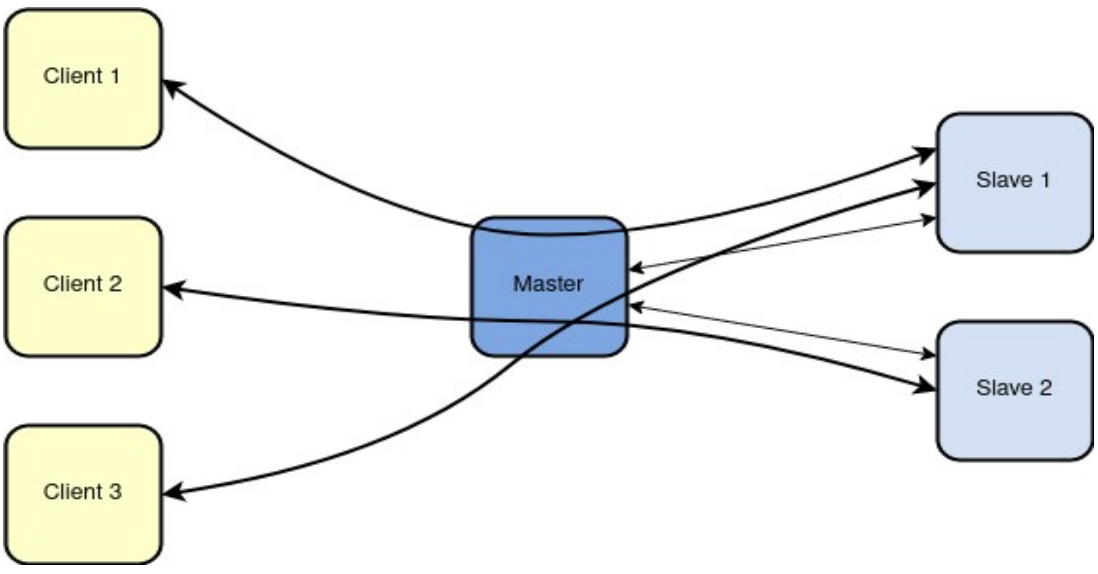
Lorsqu'on lance le client il regarde si un log existe, si c'est le cas il va regarder le contenu de celui-ci et continuer le téléchargement là où il a été interrompu.

Gestion de fermeture du serveur :

Lorsque le serveur reçoit un signal SIGINT il va d'abord envoyer ce signal à ses fils, qui vont fermer toutes les communications en cours (fermetures de tous les sockets ouverts) et se terminer, puis le père va a son tour fermer les sockets ouverts et se terminer.

Étape 3 :

Nous voulons maintenant éviter de surcharger le serveur avec trop d'utilisateurs, pour cela on peut changer l'architecture de manière à avoir un serveur maître et des serveurs esclaves qui vont s'occuper des communications (voir schéma ci-dessous). Le serveur maître attribuera aux clients qui veulent se connecter un serveur esclave avec qui ils pourront communiquer. Ce serveur sera choisi en suivant une politique round robin.



Traces de recherches :

Mise en place de l’architecture :

Le serveur maître doit connaître l’ensemble des serveurs esclaves et être capable de communiquer avec eux dans les deux sens. Le maître (resp. chaque esclave) est donc client et serveur des esclaves (resp. du maître).

Lorsqu’un client se connecte au maître, il va lui attribuer un esclave en suivant une politique round robin. Le client pourra alors communiquer avec cet esclave. Lorsqu’un changement est détecté sur un des esclaves (ex : ajout ou suppression d’un fichier) cet esclave doit informer le maître qui informera a son tour les autres esclaves qui devront mettre à jour les fichiers qu’ils contiennent.

Nouveau protocole de communication :

Une idée de protocole pour communiquer entre ce nouveau serveur et le client est d’ajouter au début du protocole précédent deux messages du serveur maître vers le client qui contiennent un nom d’hôte puis un port d’un serveur esclave qui permettent au client de se connecter à celui-ci. Il est ensuite possible de suivre le protocole précédemment établi avec le serveur esclave attribué.

Annexe :

répartition et utilisation des fonctions dans les différents fichiers :

client_FTP.c : Main de l'application côté client.

-Ouvre un socket client, appelle client_body.c puis ferme le socket client.
-Contient le handler de SIGINT pour le client

serveur_FTP.c : Main de l'application côté serveur.

-Ouvre un socket d'acceptation
-Créé des fils qui vont accepter les connexions en créant un socket de communication, lancer serveur_body puis fermer le socket de communication
-En tant que père, ferme le socket d'acceptation et attend sans rien faire.
-Contient les handlers de SIGINT et SIGCHLD pour le père et le handler de SIGINT pour les fils

client_body.c : Corps de l'application côté client.

-client_body() : Envoi une requête avec client_get() pour continuer les téléchargements interrompu s'il y en a puis demande à l'utilisateur une entrée pour effectuer une requête.

-client_get() : S'occupe de demander au serveur si un fichier existe et le télécharge si c'est le cas. Écrit également le contenu téléchargé dans un fichier et la progression du téléchargement dans le log. Supprime le log si le téléchargement s'est terminé sans problèmes.

serveur_body.c : Corps de l'application côté serveur.

-serveur_body() : Lis le code de la commande voulue et appelle la fonction serveur_get() si le code pour get est reçu.

-serveur_get() : S'occupe de lire le fichier demandé et d'envoyer son contenu au client.

Les étapes 1 et 2 ont été implémentées. L'étape 3 n'a pas été implémenté et contient les mêmes fichiers que l'étape 2.