# Projet Serveur FTP

GÖRÜR Selim
FIORI Victoria

## I. Présentation

Durant ce projet, notre objectif était de créer un serveur FTP (*File Transfer Protocol*).

Un serveur FTP permet l'échange de fichiers entre un serveur, contenant les fichiers, et un client. Le serveur peut fournir des fichiers au client et le client peut également en fournir au serveur qui les stockera dans un répertoire.

Dans un premier temps, nous avons implémenté le serveur FTP en version concurrente avec un pool de processus. C'est à dire que le serveur crée X processus fils qui vont tous interagir avec un client. Le processus père interagit également avec un client. Donc pour **X** processus fils, on permet à **X**+1 clients de se connecter au serveur.

Dans un second temps, on s'est intéressé à une autre architecture d'implémentation avec répartiteur de charge.

Un client se connecte à un serveur maître, qui va « rediriger » le client vers un des serveur esclave disponible dans un ordre type « tourniquet ».

Par exemple, pour 2 serveurs esclave et 3 clients, le client 1 est servi par le serveur esclave 1, le client 2 est servi par le serveur esclave 2 et le client 3 sera servi par le serveur esclave 1, ainsi de suite..

## II. Implémentation « concurrente »

Tout d'abord, nous avons mis en place un « protocole » afin de faciliter les communications entre le serveur et le client. Ce protocole est défini dans le fichier *ftpprotocol.h.*

```
typedef struct {
    int statusCode;
    char msg[MAXLINE];
    // -- for data transfer
    int fileSize;
    char *data;
} Response;
```

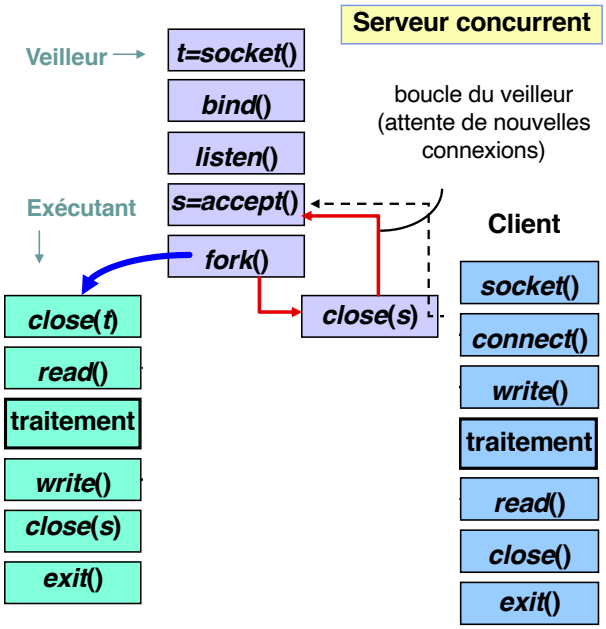
Typiquement, lors d'une demande « GET » par le client, le client envoi d'abord une demande de code **100** avec le nom du fichier souhaité. Le serveur répond par **104** en cas d'erreur (fichier inexistant etc) ou par **110** si la demande est acceptée.

Surviennent quelques vérifications avant le début du transfert (code **112**) et les données sont envoyées par le serveur et reconnues par le client avec le code **113**. Enfin, le client vérifie bien la bonne fin du transfert avec le code **111**.

À chaque message, un code d'état est défini (via *statusCode*).

| Code | Signification |
|---|---|
| 100 | Demande « GET » (demande de fichier par le client, utilisé uniquement par le client) |
| 104 | Refus de transfert « GET » (erreur, fichier inexistant, etc) |
| 110 | Demande « GET » acceptée par le serveur |
| 111 | Transfert terminé (envoyé par le serveur). Permet de détecter les erreurs de transfert. |
| 112 | Début du transfert |
| 113 | Données (durant le transfert) |
| 120 | Informations sur le transfert (indique également au serveur s'il y a une reprise du transfert après un crash client/serveur par exemple) |
| 404 | Erreur serveur/client |

Cette implémentation concurrente fonctionne en créant **N_PROC** fils (prédéfini dans *ftpprotocol.h*) qui vont contenir la primitive ***Accept()*** et le processus père également.




Son fonctionnement est décrit dans le schéma ci-joint.

La partie **traitement** correspond à l'appel de la fonction **echo(&connfd)**.

On réalise donc ici **N_PROC** fork().

Pour le client, son fonctionnement est décrit à côté.

Un des avantages de ce serveur concurrent par rapport à un serveur itératif est qu'il peut servir **N_PROC** clients à la fois (+1 si on considère que le processus père est aussi traitant).

Une des « optimisations » qui a été faite entre l'étape 1 et l'étape 2 est le transfert des données en blocs de taille définie ce qui permet d'éviter les surcharges en terme de mémoire (comme observées lors de la première étape).

La gestion d'erreurs se fait via les codes de statut précédemment énoncés. Par exemple, le client vérifie qu'il a bien reçu le message **111** à la fin d'un transfert pour s'assurer la réception s'est bien effectuée. Le client vérifie aussi que le nombre d'octets reçus correspond à la taille du fichier qui a été préalablement envoyée par le serveur.
Également, en cas de crash du serveur et/ou du client, il y a une reprise du transfert automatique.

Cette implémentation fonctionne plutôt correctement, nous n'avons pas constaté de défauts ou de soucis majeurs.

---

## III. Implémentation avec répartiteur de charge

Dans cette seconde partie, on s'est intéressé à implémenter le serveur FTP suivant une architecture de répartition de charge.

Le principe est celui-ci: nous avons un serveur maître qui joue le rôle d'un répartiteur de charge (*load balancer*), et **N** serveurs esclaves. Le serveur maître va se charger d'assigner à chaque nouveau client un serveur esclave. La répartition suit une politique de « tourniquet » plus communément appelée politique *Round-Robin*.

Par exemple, pour **N = 2**, on a alors 2 serveurs esclaves. Pour 3 clients, le client 1 est servi par le serveur esclave 1, le client 2 est servi par le serveur esclave 2 et le client 3 sera servi par le serveur esclave 1, ainsi de suite..

Cette implémentation suit toujours le même protocole décrit dans la partie précédente, mis à part l'ajout d'un code de reconnaissance d'esclave **1**.

Pour l'implémentation, nous avons choisi de définir dans *ftpprotocol.h* une variable **N_SLAVES** qui correspond au nombre de serveurs esclaves.

Premièrement, le serveur maître est lancé. Les serveurs esclaves sont crées via des **fork()** et vont se connecter au serveur maître afin d'être reconnus par celui-ci: ils envoient une requête de code **1** suivi de leur identifiant (numéro) et de leur port défini (un port est assigné automatiquement à chaque serveur esclave).

Ensuite, un client se connecte d'abord au serveur maître via le port **2121**, ils effectuent d'abord une « demande d'esclave » auprès du serveur maître qui leur renverra le port d'un des serveurs esclaves. Enfin, le client se connectera au serveur esclave via le port reçu. Le fonctionnement ensuite est identique aux parties précédentes.

---

## IV. Lancement des programmes

Les fichiers du serveur FTP sont présents dans le répertoire **_FTPROOT** et les fichiers reçus par le client sont envoyés dans le répertoire **_CLIENTROOT.**
**Pour chaque étape, les commandes sont à exécutées dans leur répertoire respectif.**

**POUR L'ÉTAPE 1 & 2 (dossier « Étape 1 » et dossier « Étape 2 »):**

1. faire **make** dans le répertoire « Étape 1 »
2. faire **./ftpserver &** pour lancer le serveur en arrière-plan
3. faire **./ftpclient localhost** pour lancer le client

**POUR L'ÉTAPE 3 (dossier « Étape 3 »):**

1. faire **make** dans le répertoire « Étape 1 »
2. faire **./ftpserver &** pour lancer le serveur en arrière-plan
3. faire **./ftpslaves localhost &** pour lancer les serveurs esclaves en arrière-plan
4. faire **./ftpclient localhost** pour lancer le client

Pour le client, les commandes disponibles sont: **get** suivi d'un nom de fichier et **bye** pour terminer la connexion.

---

## V. Tests

**POUR L'ÉTAPE 1 & 2 & 3:**

Pour les tests, nous avons testé les situations suivantes:

- Test du traitement des clients par les pools (lancement de plusieurs clients en même temps)
- Test de l'attente du client lorsque tout les pools sont occupées
- Test de la reprise de transfert en cas de crash serveur et/ou client
- Test de la gestion des erreurs en cas de fichier inexistant, ou de transfert erroné
- Test du multiformat, demande de fichiers de types différents (texte, photo, vidéo, PDF)

- Test de la bonne extinction des processus fils lors d'un CTRL+C sur le serveur (kill avec SIGINT), vérification des zombies

**POUR L'ÉTAPE 3:**

- Test de l'alternance type « tourniquet » des serveurs esclaves
- Test du bon fonctionnement des transferts
- Test de la reconnaissance des esclaves par le serveur maître