SAINDOU Soidrine G4

Pichon Kalvin-Giang Documentation/CR FTP

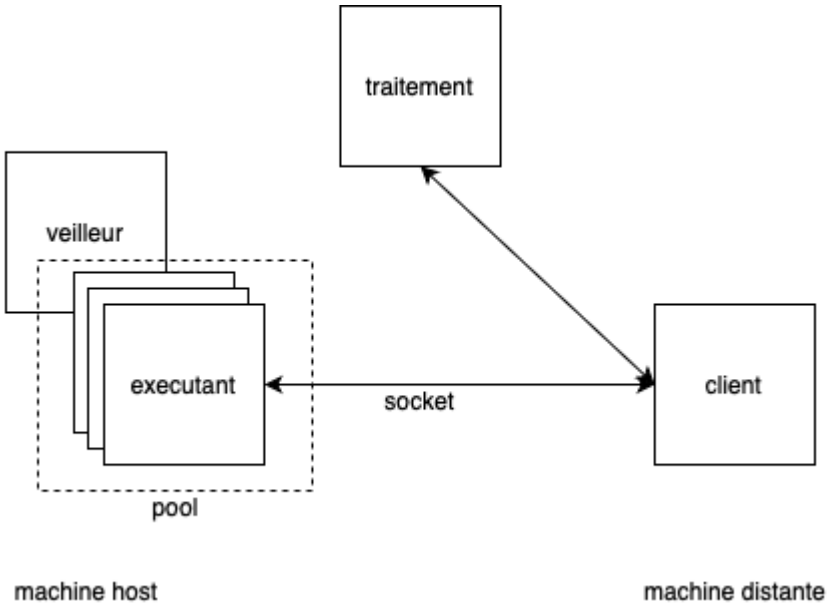
Documentation TP File Transfer Protocol

Dans ce mini projet, il s'agit de développer un FTP, en se basant sur un squelette de client-serveur.

Sommaire

1. Principe général

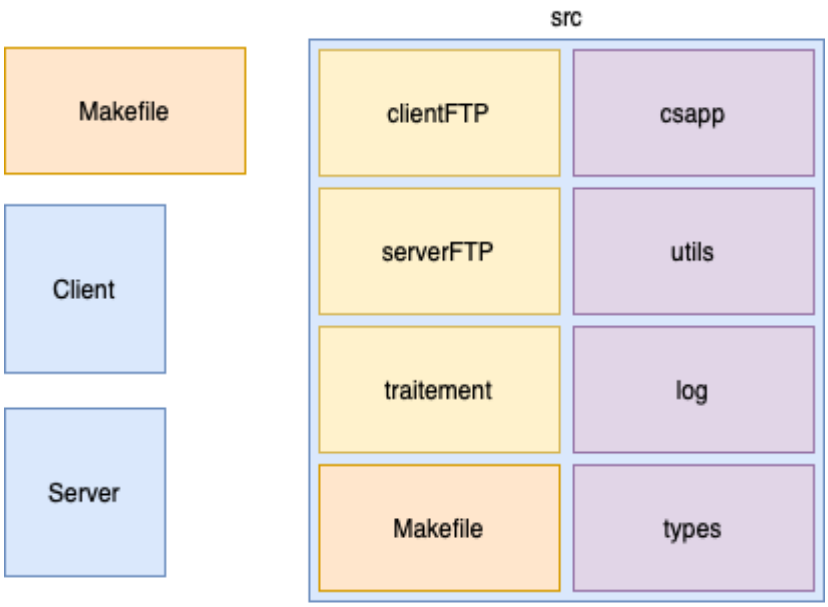
Il s'agit de développer un petit client-serveur FTP. Celui qui a été développé dans ce projet répond à la structure suivante :



Le serveur veilleur crée une socket d'écoute via la primitive : Open_listenfd(), il délègue ensuite la communication à un des exécutants. L'exécutant se met en attente de connexion d'un client avec Accept(). Lorsqu'un client se connecte, l'exécutant appelle la fonction traitement(), qui se charge d'échanger les informations du serveur vers le client, au moyen des primitives Rio_readn et Rio_writen. De la même manière pour le serveur.

2. Structure

Au fil du développement, nous avons ressenti le besoin de factoriser et d'organiser le code dans des fichiers à part entière.



Le dossier Client sert à stocker les fichiers reçus par le serveur, dont les fichiers sont stockés dans Server. Les deux exécutables clientFTP et serverFTP utilisent les types response_t, request_t et log_t.

request_t
int request_type
char filename[256]

response_t
int status
int filesize

log_t
int request_type
char filename[256]
int filesize
int paquets_recus

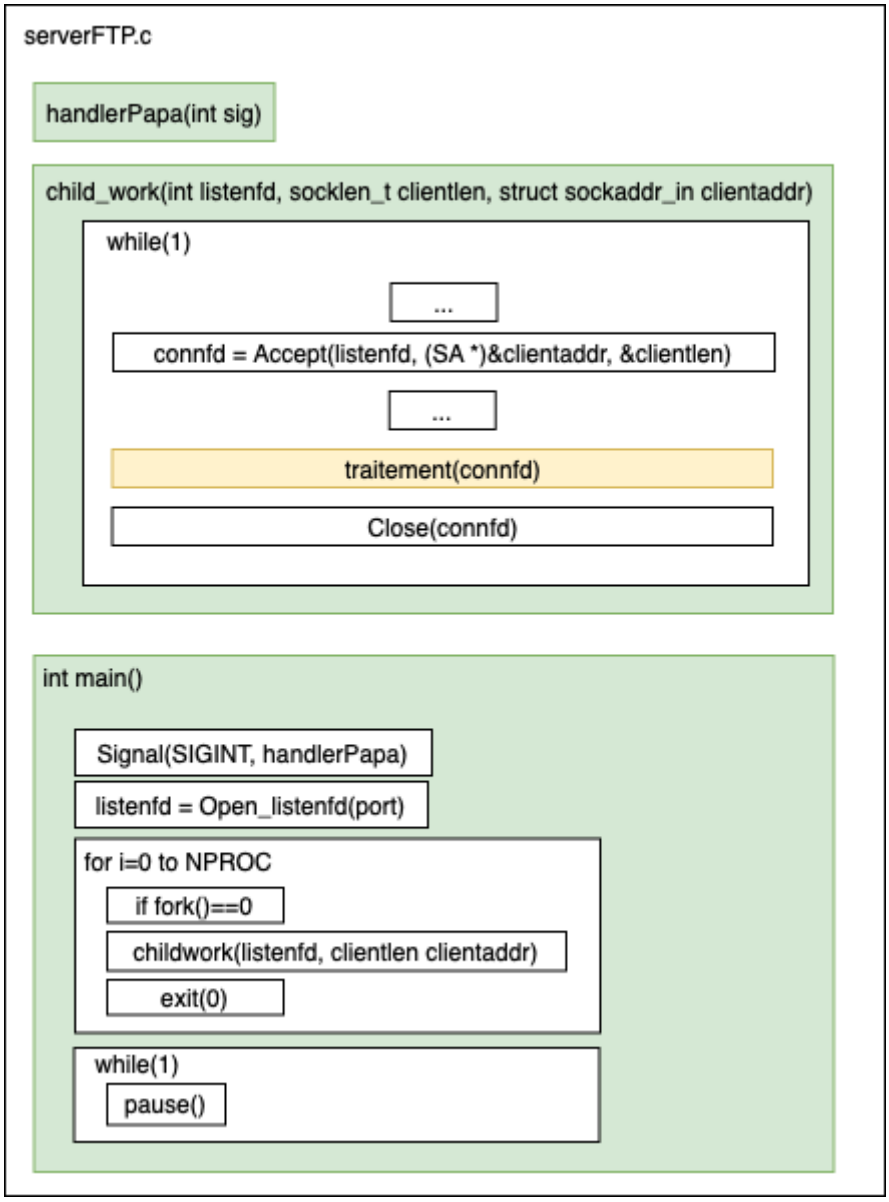
Les modules log et utils viennent avec les fonctions suivantes :

utils	
char * servpath(char * filename)	renvoie le chemin du fichier dans le serveur
char * clipath(char * filename)	renvoie le chemin du fichier dans le client
int catchsize(char *filepath)	renvoie la taille en octets d'un fichier

log	
bool existslog()	renvoie un booléen si une log existe dans le répertoire
log_t readlog()	rempli et renvoi une structure de type log_t

3. Serveur

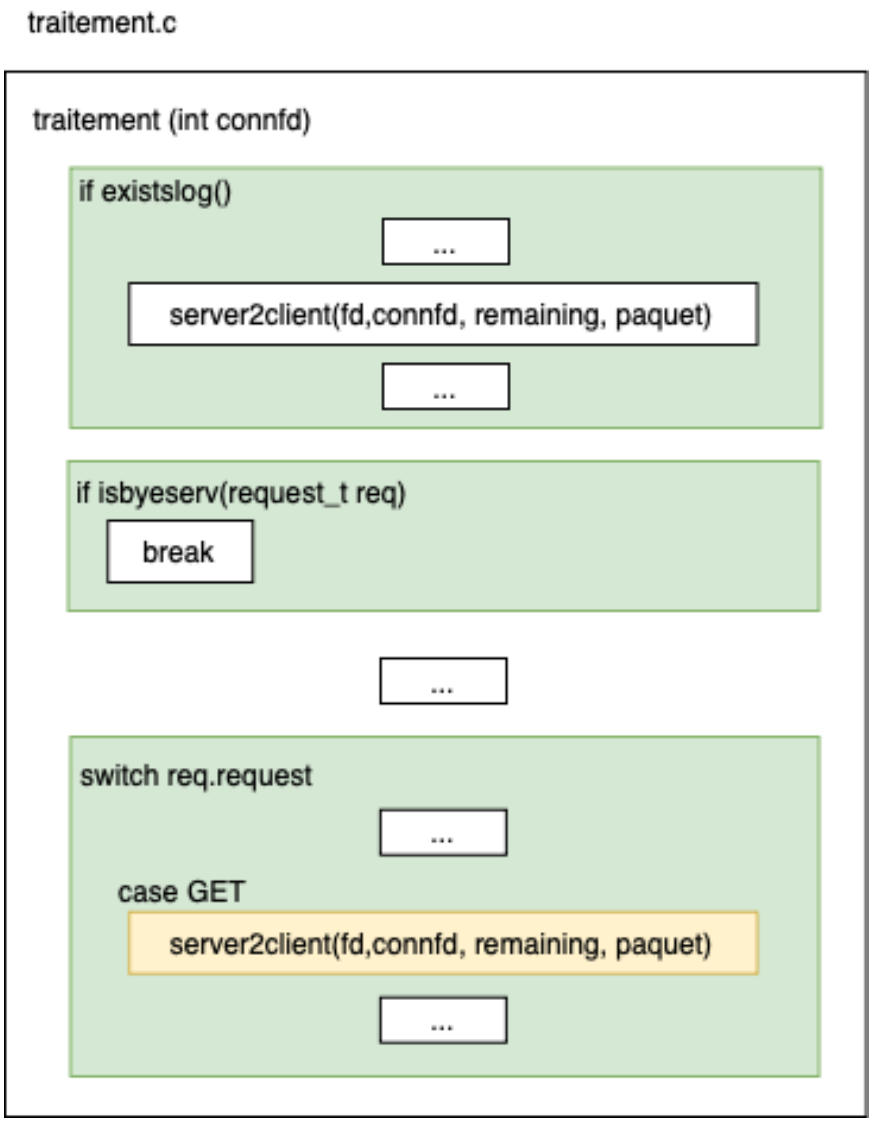
Notre serveur prend la structure suivante :



Un handler est défini pour traiter les interruptions au clavier afin de fermer les zombis. NPROC exécutants remplissent la pool. Dès qu'un client se connecte, le SE lui attribue un exécutant et l'échange peut commencer. L'échange des données est fait dans la procédure traitement() détaillé plus loin. Tant que le client ne coupe pas la communication en envoyant "bye" dans sa console, la socket de communication reste intacte et il peut y avoir une infinité d'échanges. Lorsque le client décide d'arrêter la communication, l'exécutant revient dans la pool, en attente d'une prochaine connexion.

4. Traitement

Les données sont échangées sur le socket de communication dans la procédure traitement(int connfd). Elle prend le descripteur de fichier de la socket en paramètre.



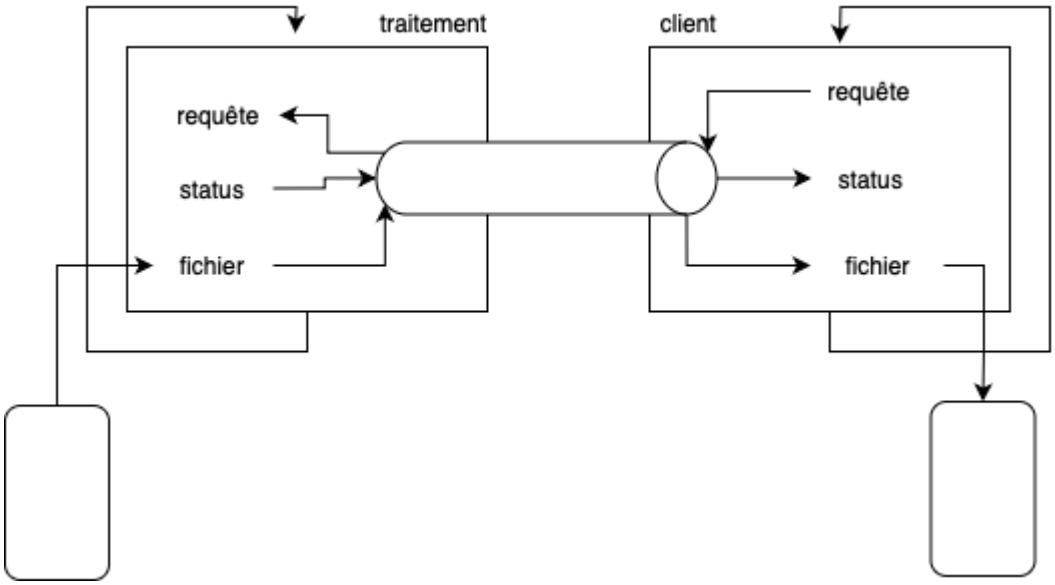
Le traitement vérifie d'abord si une log existe sur la machine, afin de reprendre le transfert d'un fichier si le client l'a interrompu via ctrl-c. Si le client tape "bye" au clavier, la socket se ferme. Sinon il procède à la communication entre le client et le serveur. La procédure qui réalise le transfert du fichier réclamé par le client est void server2client(int fd, int connfd, int remaining, int paquets). Elle prend en paramètre deux descripteurs, celui du fichier qu'elle transfère et celui du socket. Elle prend aussi le nombre d'octets restants à transférer et le nombre de paquets effectivement transférés.

5. Client

Le client est organisé et agit de la même manière que le traitement. En outre, le handler_iencli(int sig) gère les interruptions au clavier et déclenche l'écriture en local dans un fichier .log de données cruciales en vue de la reprise du transfert.

6. Application

Au final, la communication entre le client et le serveur peut se résumer à ce schéma.



7. Tests

La phase de test a été très importante lors du développement du projet. Une batterie de tests a été effectuée à chaque fin de grande partie.

Etape I : Serveur FTP de base

Spécification : Le serveur est sous forme de pool. Il y a un veillant et (dans notre cas) trois exécutants. Dès lors qu'un client se connecte, il lui est assigné automatiquement un exécutant. Le client ne peut gérer qu'une seule requête, après laquelle il ferme automatiquement la connexion. La requête est traitée par un « traitement » qui gère tous les flux du tube. Le serveur continue d'attendre de nouvelles connexions, jusqu'à recevoir une frappe ctrl-C au clavier.

Numéro	Nature du test	Test réalisé
1	fichier	get <fichier existant>
2	fichier	get <fichier inexistant>
3	requête	<requête inconnue> <fichier existant>
4	requête	<requête inconnue> <fichier existant>
5	requête	<requête> <null>
6	zombi	affichage des processus lorsque le veilleur est fermé

Chacun des tests réalisés ont eu le comportement attendu.

Etape II : Amélioration du serveur FTP

Spécification : Le programme (client, serveur, traitement) inclus les usages de l'étape I. En outre, le client se ferme après la commande 'bye', ce qui signifie que le client peut effectuer autant d'échange que souhaité par l'utilisateur. …

Numéro	Nature du test	Test réalisé
Tests de l'étape I OK		
7	fichier	Test 1 à plusieurs reprises
8	fichier	Test 2 à plusieurs reprises
9	fichier	alternance de requête get avec fichier existants et inexistants
10	requête	bye
11	requête	<requête inconnue>
12	requête	<requête> <chaine vide>
13	robustesse	serveur qui se ferme alors, la communication se ferme aussi

Petit compte rendu

Concernant les 2 premières questions on a tout simplement retranscrit en C ce qui était demandé à savoir définir un type de structure de données et un type énuméré.

Pour la question suivante nous avons recycler le code du tp précédent dans lequel on crée un pool de processus à l'aide d'une boucle for, de la fonction fork() et d'un entier NPROC (Nombre PROCessus) qui représente le nombre max de processus fils qui peut être utilisé simultanément (arbitrairement initialisé à 3 pour faciliter les tests).

Question 4: Pour cette question nous avons rencontré pas mal de problèmes notamment avec l'envoi des SIGINT vers les fils. On a créé un "handlerPapa" qui gère les SIGINT envoyés sur le serveur principal et qui à l'aide de la fonction Kill fait terminer tous les autres processus qui lui sont associés. Kill a besoin qu'on lui donne en paramètre le pid du processus qu'on veut stopper, on a donc initialisé un tableau de pid_t contenant tous les pid des processus créer dans la boucle for. Cependant on avait gardé le handler de SIGCHLD des tp précédents pour le traitement des zombies mais cela ne fonctionnait pas correctement car lorsqu'on fait un Ctrl C un SIGINT pouvait être envoyé à un processus fils en priorité et non pas au père et donc cela n'arrêtait pas le programme. On a donc fait un autre handler "handlerFils" qui lui gère les SIGINT directement envoyé au processus fils. Et donc maintenant le serveur s'arrête proprement lorsqu'il reçoit un signal d'arrêt.

Question 5: Création de deux répertoires pour séparer le serveur du client et nous avons également modifié le Makefile afin de pouvoir compiler depuis la racine une seule fois. Cette modification nous sert tout simplement à éviter de faire plusieurs Makefile qui feraient quasiment la même chose chacun. On a également créé un répertoire “src” réunissant tous les .c et .h du projet afin d’améliorer la clarté (voir schéma plus haut).

Question 6: Nous créons une nouvelle structure response_t qui va nous servire à communiquer une valeur de retour indiquant au client si sa requête est valide ou non. Pour le traitement des requêtes on utilise une fonction “traitement (int connfd)” par chaque processus qui grâce à un switch va déterminer de quel type il s’agit (ici GET nomFICH)
. Si le fichier demandé existe dans le répertoire Serveur alors celui-ci va changer son statut en FOUND et charger le fichier dans son intégralité dans le socket de communication et se remettre en attente. Le client lui de son côté va vérifier que sa requête a bien été acceptée puis va ouvrir(créer) un fichier à l’aide de la primitive **Open()** pour y copier le contenu du tube. Remarque: si le fichier existe déjà il sera écrasé pour y mettre le nouveau contenu.

Question 7: Ici contrairement à la question précédente nous voulons pouvoir faire la même chose mais en utilisant une sorte d’interface sur l’entrée standard. Pour cela nous avons simplement affiché “ftp>” à l’aide d’un printf et ensuite nous récupérons ce que le client écrit au clavier avec la méthode “**fgets**” puis stockons cela dans un String et par la suite avec un **sscanf** on étudie si le client remplit bien les conditions de requête à savoir <requête> <nomFICH>. Si c’est le cas, elle est envoyée sur le socket de connexion à l’aide d’un **Rio_writen** pour pouvoir être traitée comme vu précédemment à la question précédente.

Question 8: Pour cette étape nous avons tout simplement changé la taille du buffer qu'on utilisait de façon dynamique par une taille fixe, 4096 Octets (valeur choisie arbitrairement). Nous ne partageons pas le nombre de paquets que le client va recevoir mais directement la taille du fichier, on les récupère donc en vérifiant combien d’octets il reste à chaque paquet récupéré. La taille est communiquée via le socket avec la structure response_t.

Question 9: Pour pouvoir effectuer plusieurs demande de fichier nous avons ajouté une boucle infinie dans le client et retiré le **Close** et **exit** qui forçait l'arrêt. Le "bye" nous a donné un peu plus de fil à retordre …

Question 10: Pour garder les informations du fichier interrompu nous avons utilisé un une autre structure log_t avec fichier .log qui est fabriqué avec les données qui nous intéressent (taille, nom, paquet reçu). Grâce à cette structure dans un handler du client nous stockons l'état actuel du client puis le fermons. Une fois qu'un client se reconnecte, celui-ci télécharge la suite du fichier qui avait été interrompu.

Petits problèmes

Erreur intéressante relevée : Lorsque que l'on lance clientFTP sur plusieurs terminaux simultanément et qu'on interrompt brusquement le transfert sur un des clients, alors la prochaine demande de n'importe quel client récupère le reste du fichier qui avait été interrompu. Par contre si par malheur le client qui récupère le fichier n'est pas celui qui l'avait interrompu, alors la récupération est impossible pour ce client.

Ainsi, un client qui se connecte ou juste effectue une requête récupère d'entrée la suite du fichier interrompu, ce qui n'est pas très réaliste car un autre client n'ayant aucun lien avec l'interruption devrait pouvoir effectuer ses requêtes indépendamment du bon fonctionnement des autres. Comme solution possible on a pensé à ajouter dans le fichier .log l'adresse de la machine qui a eu une interruption. Sur la même machine on ne verra pas la différence mais sur 2 machines séparées oui.