Projet Système Réseau

BOU SERHAL Jean, CABROL Camille, FRANCES Tom, GOURDON Jérémie

December 3, 2021

Contents

Introduction

Le but de ce projet est de créer une application réseau client/serveur TCP/IP simplifiée, permettant l'échange d'images entre le client et le serveur. Le client pourra choisir de déposer ou de récupérer des images sur le serveur. Ce serveur acceptera les fichiers images déposés sous conditions d'admission, et devra assurer le service pour plusieurs clients simultanément.

Nous avons découpé ce projet en plusieurs étapes :

1. Création du noyau client/serveur
2. Conception de l'application (architecture et protocole d'échange)
3. Conception et implémentation des fonctionnalités de base (transfert de fichiers, critères d'admissibilité...)
4. Intégration des fonctionnalités au noyau

Chapter 1

Conception du noyau

Dans les 2 sections suivantes, nous avons décidé de gérer toutes les erreurs des fonctions système (*socket()*, *connect()*, etc) en affichant **errno** via la commande **perror()**, en profitant du fait que ces fonctions le positionnent.

Au départ, nos variables de type **sockaddr_in** étaient déclarées en tant que pointeurs, mais pour simplifier l'écriture du code et ne pas avoir à utiliser de **malloc()**, nous avons préféré ne pas utiliser de pointeurs, et de transmettre l'adresse de ces variables (**&nom_variable**) aux méthodes les utilisant en paramètre.

Nous avons choisi d'attribuer le port numéro 6067 au serveur, après avoir vérifié que ce port était libre grâce à la commande **netstat -t** listant tous les ports TCP ouverts.

1.1 Client

Nous avons tout d'abord créé une socket afin de permettre la communication entre le client et le serveur, puis nous avons récupéré les informations du serveur en utilisant le nom de notre machine (commande **gethostbyname()**).
Dans un premier temps, nous avons préféré travailler sur une même machine, en utilisant le nom **localhost**, afin de faciliter nos tests. Dans un second temps, une fois la création du noyau terminée et fonctionnelle, nous sommes passés sur l'utilisation de notre programme sur 2 machines distinctes, l'une jouant le rôle du client, l'autre du serveur. Pour cela, nous avons récupéré le nom de la machine serveur via la commande **hostname**, qui remplace le nom **localhost** dans le code de *Client.c*.
Ensuite, à l'aide de la commande **connect()**, nous avons établi une connection entre le client et le serveur. Une fois cette connection établie, nous avons choisi d'envoyer un entier au serveur via la méthode **write()**, pour des raisons de facilité (taille, lecture, écriture...).
Enfin, en utilisant la méthode **read()**, le client récupère une réponse de la part du serveur.

1.2 Serveur

Nous avons tout d'abord créé une socket permettant au serveur d'écouter les messages de clients, puis attaché cette socket au port choisi précédemment (commande **bind()**) : une socket étant un descripteur de fichier, il faut lui assigner un port afin de lui indiquer où elle doit écouter.
Afin d'éviter une accumulation trop importante de connections en attente, nous avons décidé de limiter leur nombre à 5 (car nous avons décidé que c'était suffisant dans notre cas).
Nous avons géré l'acceptation de la connection du client au serveur dans une boucle **while(1)** (1 car tout le temps vrai, ce qui nous permet de gérer manuellement la mort du serveur étant donné que les fonctionnalités s'occupant de cet aspect ne sont pas encore implémentées). Dans cette boucle, le serveur lit le message envoyé par le client via une socket de service renvoyée par la fonction **accept()**.
Pour le test de cette fonction, nous avons rencontré une erreur : nous souhaitions afficher un message lors de la réussite du test, hors rien ne s'affichait sur la console. Nous nous sommes rendu compte qu'il manquait un "**\n**" à la fin du **printf()**, car sans celui-ci, il n'y avait pas de flush du

buffer sur la sortie standard, donc pas d'affichage. Afin de permettre au serveur de traiter plusieurs connections simultanées, ce dernier délègue chaque nouvelle connection à un processus fils (1 fils par connection), créé au sein d'un **switch(fork())** :

cas -1 erreur de création du processus, on arrête le programme (**exit(-1)**)

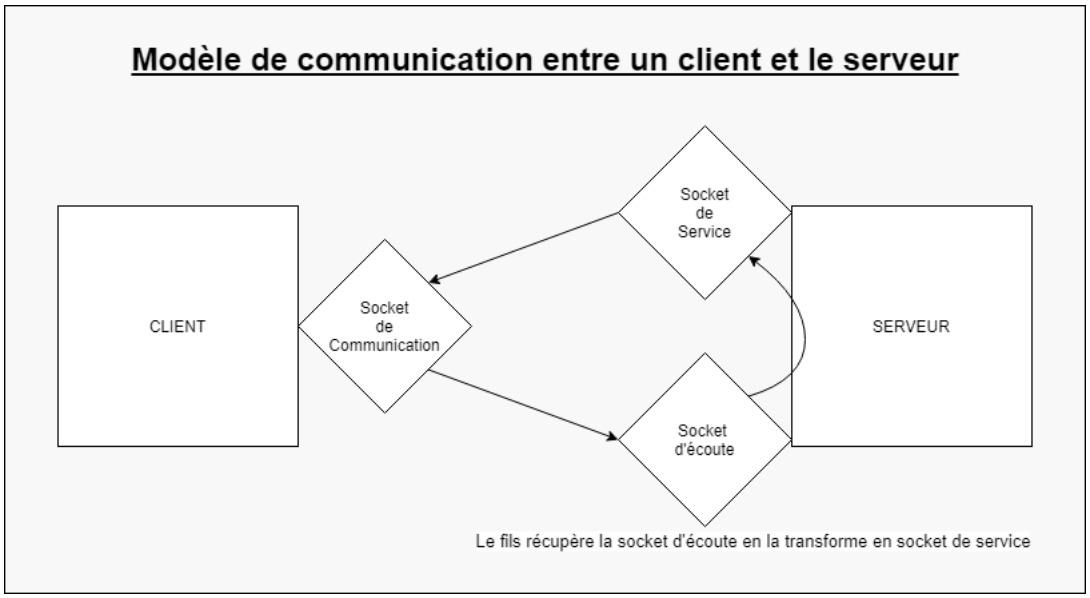
cas 0 comportement du fils : fermeture de la socket d'écoute (car le processus fils ne l'utilise jamais), réinitialisation du comportement du signal **SIGCHLD** au cas où ce processus doive lui-même créer des fils.

cas default comportement du père, qui ici ne fait rien. On pourrait vouloir le faire attendre la mort de son fils à l'aide d'un **wait(NULL)**, mais en faisant cela, on bloquerait le programme qui ne pourrait ainsi pas traiter plusieurs connections et requêtes simultanément.

Chapter 2

Conception de l'application

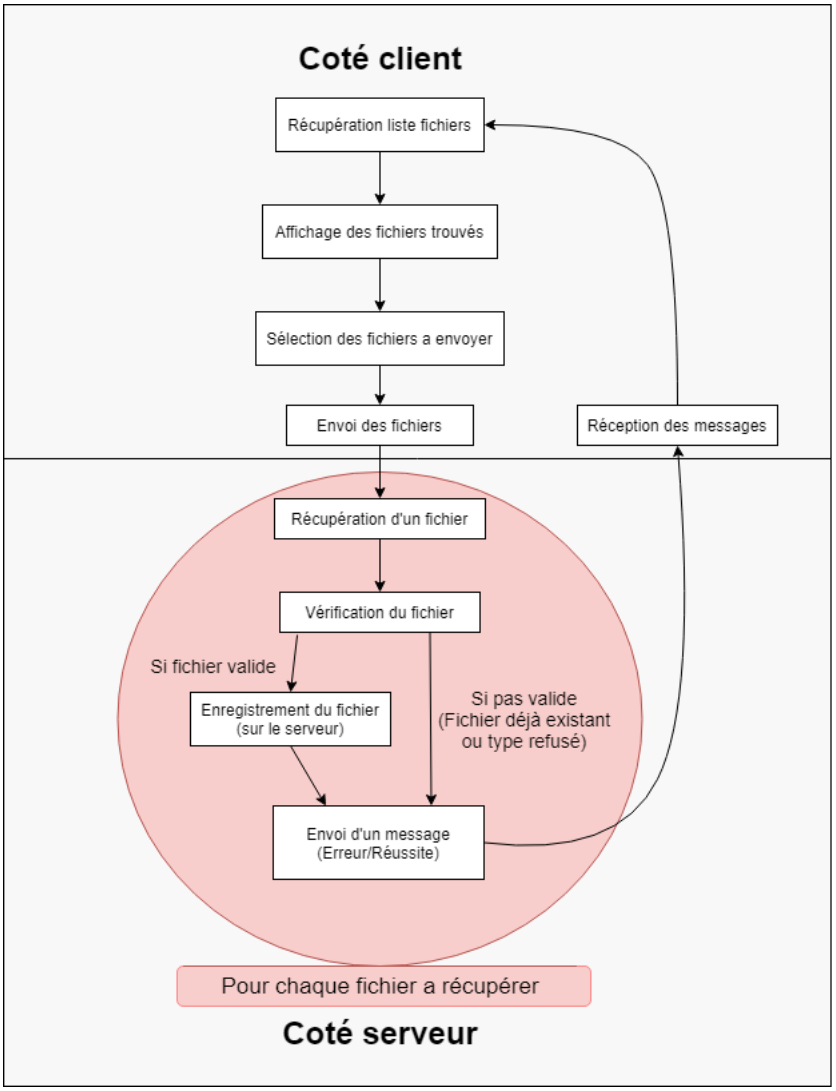
Pour cette partie, avant toute chose nous avons posé nos idées sur papier afin de se mettre d'accord sur l'architecture ainsi que le protocole de notre application. Découlant du noyau expliqué précédemment voici un premier schéma de notre architecture :



Pour nos échanges, nous avons utilisé une socket. Comme expliqué ci-dessus, le client a une seule socket (*socketCommClilent*), et le serveur une seule également (*socketService/socketEcoute*). Les deux sockets du serveur représentées sur notre schéma sont donc les mêmes : la socket d'écoute est la socket utilisée par le processus père du serveur. Lors de la connection d'un client au serveur, ce dernier transmet la socket à un fils qui gère l'échange. La socket d'écoute devient alors la socket de service reliant le fils au client.

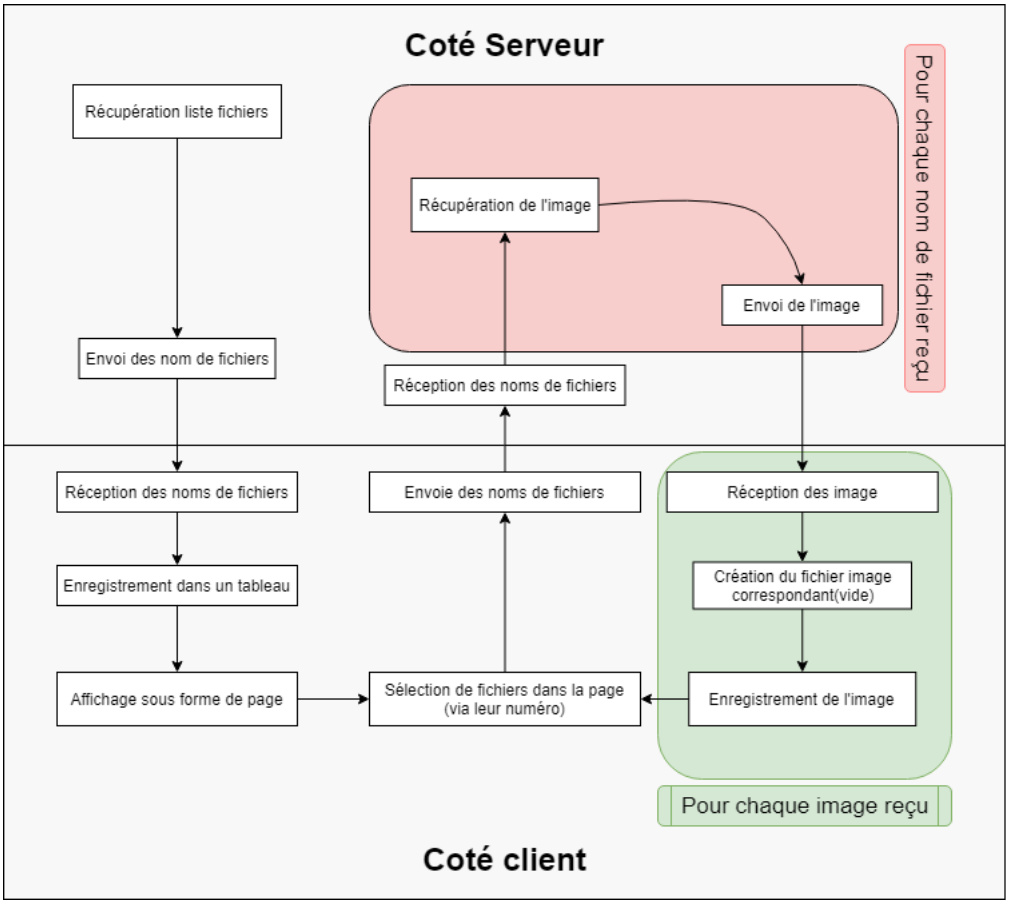
2.1 Dépôt d'image(s) sur le serveur

Pour cette partie, nous avons imaginé partir du client, et afficher à l'utilisateur la liste des images qu'il a à sa disposition pour pouvoir les déposer à sa guise sur le serveur. Ensuite, l'utilisateur aurait la possibilité de donner le(s) fichier(s) qu'il souhaite envoyer. Après la sélection, on envoie le(s) fichier(s). Puis, pour chaque fichier reçu, on vérifie son admissibilité. S'il est admissibile on le dépose sur le serveur et on envoi un message au client pour l'assurer de la fin du dépôt de son fichier sur le serveur, s'il n'est pas admissibile on prévient le client à l'aide d'un message que le dépôt n'est pas possible.



2.2 Téléchargement d'image(s) depuis le serveur

Pour cette partie, nous avons imaginé partir du serveur, récupérer la liste des images présentes sur celui-ci et transmettre au client la liste des noms des fichiers. Le client récupère cette liste et la stocke dans un tableau permettant d'afficher à l'utilisateur la liste des fichiers disponibles au téléchargement. Après le choix de l'utilisateur sur le(s) fichier(s) qu'il souhaite récupérer, on transmet au serveur le nom du ou des fichier(s) choisi(s), ce dernier les récupère et suite à ça envoie une par une les images correspondantes. Lors de la réception de celles-ci auprès du client, pour chaque image reçue, on crée un fichier vide, puis on y enregistre l'image.


Coté Serveur
Récupération liste fichiers
Envoi des nom de fichiers
Réception des noms de fichiers
Enregistrement dans un tableau
Affichage sous forme de page
Sélection de fichiers dans la page (via leur numéro)
Envoie des noms de fichiers
Réception des noms de fichiers
Récupération de l'image
Envoi de l'image
Pour chaque nom de fichier reçu
Réception des image
Création du fichier image correspondant(vide)
Enregistrement de l'image
Pour chaque image reçu
Coté client

Chapter 3

Réalisation des fonctionnalités de base

Pour cette partie,

3.1 Dépôt d'image(s) sur le serveur

Nous avons choisi pour l'envoi de fichier(s) vers le serveur, de réaliser une fonction globale qui appelle toute les autres méthodes. Cette partie doit pouvoir :

1. Récuperer la liste des fichiers se trouvant dans le dossier image du client
2. Afficher cette liste à l'utilisateur
3. Lui permettre de sélectionner un ou plusieurs fichier(s) dans la liste
4. Les envoyer au serveur

La fonction **envoiServeur()** est la fonction principale :

1. Elle récupère la liste des fichiers du dossier en appellant la fonction correspondante
2. Elle affiche la liste avec un système de pages (4 fichiers par page)
3. Elle demande à l'utilisateur ce qu'il souhaite faire (naviguer entre les différentes pages, sélectionner le(s) fichier(s) à envoyer)
4. Si l'utilisateur a choisi l'envoi de fichier, alors elle envoie les fichiers
5. Ramène l'utilisateur au menu principal

1- Récupère la liste

RecupereListeImagesClient() est la fonction qui se charge de récuperer la liste à l'aide d'une structure de type **dirent**. Ensuite, on parcourt l'arborescence du dossier, et pour chaque fichier, à condition que ce soit bien un fichier (i.e : ce n'est pas un sous dossier ou un fichier mal formé) mais pas un fichier caché (i.e : commence par un point), on ajoute son nom à la liste à l'aide d'un **realloc()** qui agrandi notre tableau de chaîne de caractères, puis renvoie la liste.

2- Choix possibles

1. Page précédente
2. Sélectionner les fichiers à envoyer
3. Page suivante
4. Revenir au menu principal

2.2- Sélectionner des fichiers à envoyer

recupereListeImagesAEnvoyer() est la fonction qui se charge de demander à l'utilisateur la liste des fichiers qu'il souhaite envoyer. Il s'agit d'une boucle demandant à l'utilisateur d'entrer le numéro de fichier à ajouter à la liste. Tant que ce n'est pas 0 (code de fin), on ajoute dans le tableau des fichiers à envoyer le nom du fichier correspondant au numéro. Une fois la saisie terminée, la fonction renvoi le tableau contenant la dite liste.

4- Envoi des fichiers

Appelle la méthode **envoiImages()** en lui fournissant la liste des fichiers à envoyer. Cette méthode appelle pour chaque fichier de la liste la fonction **envoiImage()** qui envoi un seul et unique fichier au serveur. **envoiImage()** genère le chemin d'accès à l'image ainsi que le chemin du fichier à l'arrivée. Puis, envoi au serveur le chemin/nom du fichier qu'il va réceptionner suivi de la taille du fichier. L'image est lue par bloc de 4096 octets et chaque et directement envoyer au server. Une fois l'envoi terminé, on attend le retour du serveur signifiant que la réception a bien été effectuée. L'admissibilité des fichiers envoyés est gérée par une fonction qui sera intégrée dans ce qui suit (voir partie 3.3 Admissibilité des fichiers).

0- Retour au menu

A l'exception de ce choix, le code boucle sur l'affichage de la liste et demande à l'utilisateur ce qu'il souhaite faire, que ce soit après un changement de page ou après l'envoi de fichier(s).

De son coté, le serveur une fois la connexion établie avec le client délègue l'échange à un fils (On le mentionnera comme le server dans cette partie) Le serveur (fils qui gère le dialogue) attend de recevoir un code lui indiquant ce que le client souhaite faire. Quand il reçoit le code lui indiquant qu'il va recevoir des fichiers, il attend tout d'abord de savoir combien de fichiers il va recevoir, puis pour chaque fichier il crée un fichier vide à l'aide de la commande *touch*, et écrit le contenu qu'il reçoit dedans. Une fois fait, il renvoi au client un code indiquant la fin du téléchargement.

3.2 Téléchargement d'image(s) depuis le serveur

3.3 Admissibilité des fichiers

La fonction listeMymes()

Ci-dessous, nous partons du principe qu'il n'y a qu'un nombre de types MIME précis passé en paramètre à la fonction **listeMymes()**; Nous avons choisi de récupérer les types MIME admissibles du fichier texte *MimeTypes.txt* dans un tableau (équivalent à un char** ou char *[]). Cette méthode de récupération a été implémenté en haut niveau et en bas niveau, mais nous avons choisi de maintenir les fonctions de haut niveau dans notre code pour raison d'une meilleure maîtrise de ces dernières. En partant du principe que la taille d'un type MIME n'excède pas 30 caractères, nous récupérons ligne par ligne chaque type dans une mémoire allouée. Chacune de ces allocations mémoire possède un pointeur dans une autre mémoire allouée qui est le tableau regroupant tous les types MIME. La fonction renvoie le tableau contenant les types MIME admissibles.

Phase de vérification de l'admissibilité d'un fichier : la fonction admissible()

Cette fonction prend en paramètre le nom du fichier à vérifier (qu'il est de type image valide). Ce dernier est passé dans une ligne de commande exécutée par un processus fils. La ligne de commande : **$ file –mime-type nomFichier**. Nous avons choisi de remplacer le **file -i** par un **file –mime-type** puisque c'est la seule information qui nous intéresse à récupérer, et puisque ça nous facilitera également la gestion et le découpage de la chaîne de caractères récupérée dans ce qui suit. On fait un **fork()** afin de déléguer la récupération du type MIME du fichier à un processus fils via un **execvp()**. Un **read()** dans un pipe nous permet de lire la chaîne de caractères récupérée, en redirection de la sortie standard à l'aide d'un **dup()**. Nous avons fait face à des difficultés lors du découpage de la chaine de caractères contenant le type MIME, pour récupérer uniquement l'information qui nous intéresse (le type MIME), via des fonctions comme **strrchr()**, pour prendre en compte la dernière occurrence du caractère

d'espace et tout ce qui suit. Nous manipulerons cela pour supprimer ce qui entoure le type (espaces avant, retours à la ligne et autres caractères après, ...). Nous comparons finalement ce résultat final, aux types MIME contenus dans le tableau des types admissibles pour vérifier son existence et par conséquent son admissibilité.

1. Dans le cas où le fichier est admissible (admettons qu'il ai été au préalable placé dans un répertoire temporaire *tmp*, il va être placé dans le répertoire *FilesServeur* chez le serveur. En réalité, il va être déplacé du répertoire temporaire *tmp* vers le répertoire où il est supposé initialement être admis, et ce en faisant un **fork()** : le processus fils créé se charge d'effectuer la commande *move (mv)* via un **execlp()**.
2. Dans le cas contraire le fichier est non admissible, et son type MIME ne figure pas dans le tableau admissible (admettons qu'il ai été AUSSI au préalable placé dans un répertoire temporaire *tmp*), il ne sera pas téléchargé chez le serveur (et plus précisément dans le répertoire *FilesServeur*. En réalité, il va être complètement supprimé du répertoire temporaire *tmp*, et ce en faisant un **fork()** : le processus fils créé se charge d'effectuer la commande *remove (rm)* vis un **execlp()**.

La fonction **admissible()** est intégrée dans la fonction **receptionImage()** côté serveur.

Chapter 4

Intégration du noyau avec toutes ses fonctionnalités

Chapter 5

Mini manuel pour l'exécution de l'application

Conclusion

(Penser à dire qu'on a ajouter un type vidéo dans les MimesTypes et qu'on a réussi à en envoyer).