

Projet Système Réseau

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

December 3, 2021

Contents

1	Conception du noyau	3
1.1	Client	3
1.2	Serveur	3
2	Conception de l'application	5
2.1	Dépôt d'image(s) sur le serveur	5
2.2	Téléchargement d'image(s) depuis le serveur	6
3	Réalisation des fonctionnalités de base	8
3.1	Dépôt d'image(s) sur le serveur	8
3.2	Téléchargement d'image(s) depuis le serveur	9
3.3	Admissibilité des fichiers	10
4	Intégration du noyau avec toutes ses fonctionnalités	12
5	Mini manuel pour l'exécution de l'application	13

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.

sigaction() contient 3 paramètres :

1. le signal sur lequel on l’applique
2. la structure sigaction définissant ce que l’on fait de ce signal
3. NULL

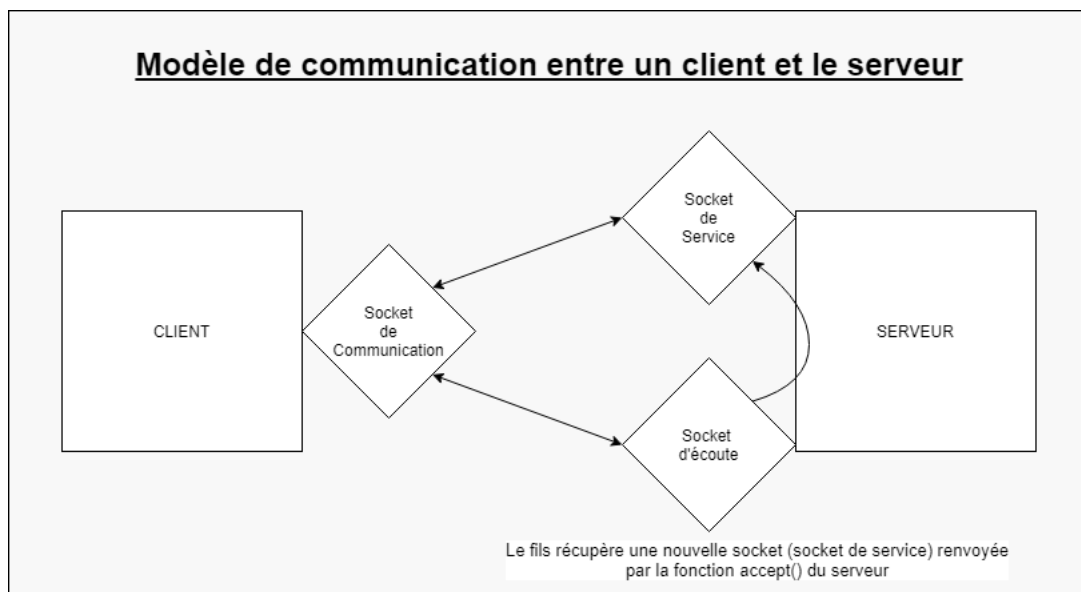
Quand un fils meurt, son signal SIGCHLD s’ajoute dans la pile des SIGCHLD. La fonction **wait(NULL)** vient dépiler le premier SIGCHLD de réception des signaux.

On utilise donc **wait()** afin de s’assurer que les fils soient morts et qu’il n’y ai pas de zombies. La commande sigaction ainsi que la structure sigaction sont nécessaire ici afin de pouvoir gérer les signaux SIGCHLD et s’assurer de la mort des fils tout en ne bloquant pas le code. Cela permet en effet de laisser le père continuer de faire ce qu’il a à faire sans s’occuper de ses fils car c’est désormais la structure qui s’en occupe.

Chapter 2

Conception de l'application

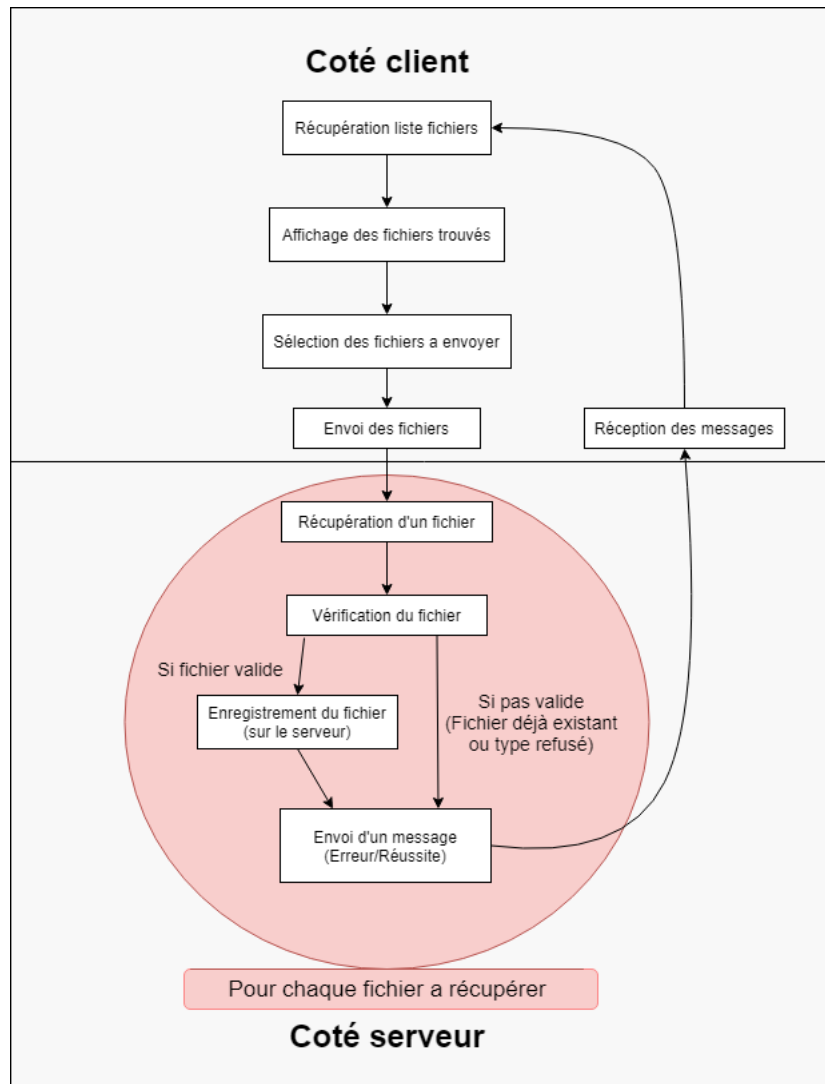
Pour cette partie, avant toute chose nous avons posé nos idées sur papier afin de se mettre d'accord sur l'architecture ainsi que le protocole de notre application. Découlant du noyau expliqué précédemment, voici un premier schéma de notre architecture :



Pour nos échanges, nous avons utilisé une socket. Comme expliqué ci-dessus, le client a une seule socket (*socketCommClient*), et le serveur une seule également (*socketEcoule*). A chaque connexion d'un client, il crée une nouvelle socket (*socketService*) qui sera utilisée par un processus fils pour effectuer tous les échanges avec le client.

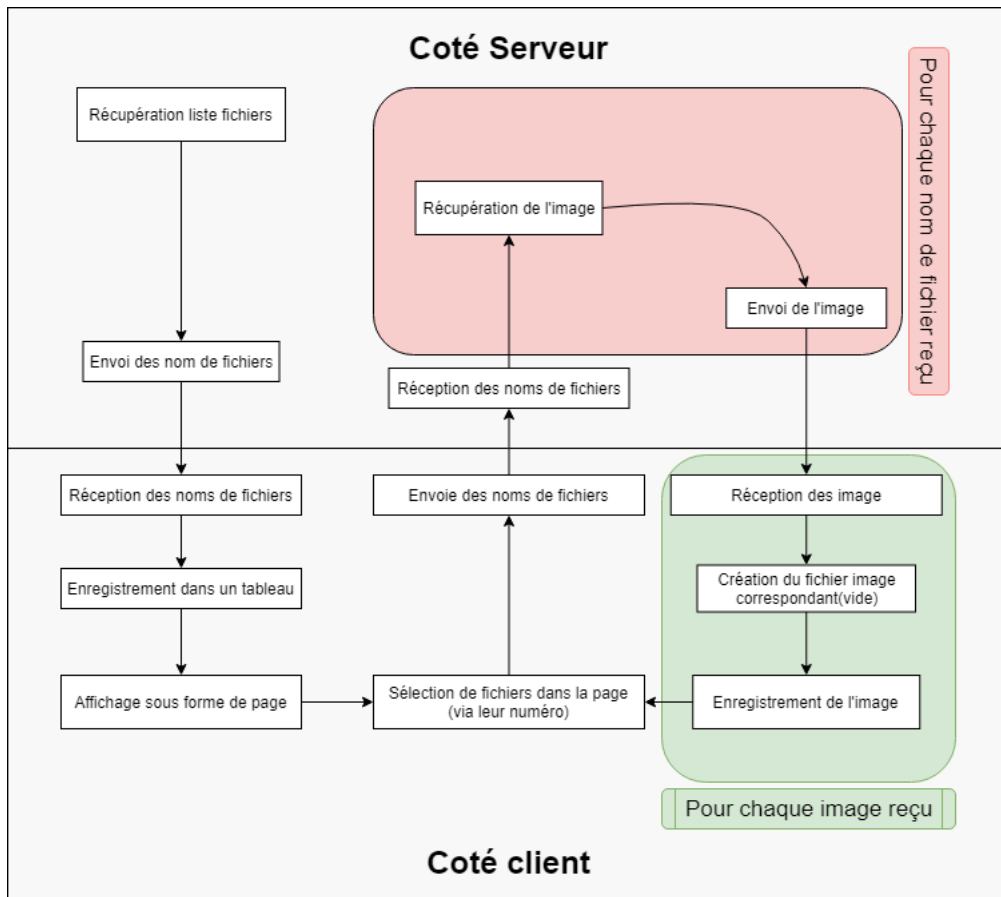
2.1 Dépôt d'image(s) sur le serveur

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 admissible on le dépose sur le serveur et on envoie un message au client pour l'assurer de la fin du dépôt de son fichier sur le serveur, s'il n'est pas admissible 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.



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. Récupérer 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 appelant 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 récupérer 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 envoie un seul et unique fichier au serveur. **envoiImage()** génère le chemin d'accès à l'image ainsi que le chemin du fichier à l'arrivée. Puis, envoie 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 envoyée au serveur. 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 côté, le serveur une fois la connexion établie avec le client délègue l'échange à un fils (On le mentionnera comme le serveur 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

Dans le cas où l'utilisateur choisit de télécharger (récupérer) des fichiers images depuis le serveur, et les placer dans le dossier d'images chez le client ;

La fonction principale ici est la fonction **telechargementServeur()** du côté client : elle appelle les fonctions prises en charge du côté client.

1. DU COTE SERVEUR La fonction **recupereListeImagesServeur()** : Cette fonction se charge de récupérer la liste à l'aide d'une structure de type **dirent**. Ensuite, on parcourt l'arborescence du dossier (le répertoire *FichiersServeur*) 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 agrandit notre tableau de chaîne de caractères. Cette fonction renvoie la liste. La manipulation de l'ouverture, de la lecture et de la fermeture du dossier se fait respectivement grâce à un **opendir()**, **readdir()** et **closedir()**. La liste renvoyée par la fonction ci-dessus est envoyée/écrite dans la socket qui communique avec le client.
2. DU COTE CLIENT La fonction **receptionListeImagesServeur()** : Elle récupère la liste d'images présentes sur le serveur en lisant dans la socket. Elle lit tout d'abord le nombre de fichiers présents, puis les noms des fichiers images, et les stocke dans un tableau de chaînes de caractères.
3. DU COTE CLIENT La fonction **choixImagesATelecharger()** : Elle permet au client (utilisateur) de choisir les images qu'il souhaite télécharger chez lui, en les sélectionnant grâce à leur numéro d'affichage. Les images choisies seront placées dans une liste. La liste précédente sera envoyée au serveur via son écriture dans la socket (socketCommClient) dans la fonction appelée **envoiListeImagesATelecharger()**.
4. DU COTE SERVEUR La fonction **recupereListeImagesATelecharger()** : Cette fonction lit dans la socket la liste des images à télécharger (en fonction de leur nombre) et la renvoie.

5. TOUJOURS DU COTE SERVEUR, on reprend les fonctions d'envoi des fichiers du client au serveur, mais cette fois-ci, dans le sens inverse : c-à-d du serveur vers le client = téléchargement. Appel à la fonction **envoiImages()** en lui fournissant la liste des fichiers à envoyer. Cette fonction appelle pour chaque fichier de la liste, la fonction **envoiImage()** qui envoie un seul et unique fichier au client. **envoiImage()** génère le chemin d'accès à l'image ainsi que le chemin du fichier à l'arrivée. Puis, envoie au client 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 bloc est directement envoyé au client. Une fois l'envoi terminé, on attend le retour du client signifiant que la réception a bien été effectuée.
6. DU COTE CLIENT : il réceptionne les images qui lui sont envoyées grâce à la fonction **receptionImage()**, via la lecture dans la socket de communication. L'image est tout d'abord créée par la commande *touch* au sein d'un **fork()**, puis remplie par des blocs de 4096 octets qui sont lus dans la socket ; jusqu'à atteindre la taille du fichier image récupérée.

3.3 Admissibilité des fichiers

La fonction **listeMypes()**

Ci-dessous, nous partons du principe qu'il n'y a qu'un nombre de types MIME précis passé en paramètre à la fonction **listeMypes()**; 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 implémentée 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 chaîne de caractères contenant le type MIME, pour récupérer uniquement l'information qui nous intéresse (le type MIME), via des fonctions comme **strchr()**, 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 ait 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 **rename()**. Une autre piste de réflexion que l'on a préféré abandonner puisqu'il existe une fonction C pour faire cela : en faisant un **fork()**, le processus fils créé se charge d'effectuer la commande *move (mv)* via un **execvp()**.
2. Dans le cas contraire le fichier est non admissible, et son type MIME ne figure pas dans le tableau admissible (admettons qu'il ait 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 **unlink()**. Une autre piste de réflexion qu'on a préféré abandonner aussi : en faisant un **fork()**, le processus fils créé se charge d'effectuer la commande *remove (rm)* vis un **execlp()**.

La fonction **checksumInterdite()**

Un second critère d'admissibilité que nous avons implémenté est la vérification d'une somme de contrôle, ou checksum, sur les fichiers de type PPM (x-portable-pixmap).

Le principe de la somme de contrôle est de vérifier la somme des valeurs d'un sous-ensemble de pixel de l'image. Il nous faut dans un premier récupérer l'ensemble des valeurs des pixels de l'image. Nous utilisons pour cela un tableau d'entiers, de dimension `nombreDePixels * 3` : dans chaque ligne du tableau seront stockées les valeurs des couleurs d'un pixel (bleu, vert et rouge).

Une fois tous les pixels récupérés, nous allons en extraire un nombre fixe de 100, choisis de manière pseudo-aléatoire à l'aide de la fonction **rand()** : puisque la checksum doit être la même à chaque vérification sur un même fichier, nous initialisons la graine du générateur de nombres pseudo-aléatoire à l'aide d'une constante, 100, choisie arbitrairement. Ensuite, nous faisons la somme de toutes les valeurs de chaque couleur séparément, et nous utilisons ces valeurs pour construire la checksum de l'image.

Nous avons décidé, par manque de temps, de construire et de stocker la checksum non pas sous un format binaire, mais hexadécimal.

XXXX XXXX XXXX XXXX

Les bloc de 4 valeurs hexadécimales (16 bits) représentent, de la droite vers la gauche, les valeurs hexadécimales de la somme des couleurs bleue, verte, et rouge du sous-ensemble de pixels extrait. Le dernier bloc le plus à gauche est mis à 0000 afin que la checksum soit représentée sur 64 bits.

Nous aurons au préalable enregistré, dans un fichier `checksum.txt` coté serveur, une checksum décidée interdite, dans notre cas celle de l'image `mammouth.ppm`. Il ne sera alors plus possible de transférer cette image sur le serveur, et un message d'erreur annoncera au client qu'elle a été refusée car la checksum est interdite.

Chapter 4

Intégration du noyau avec toutes ses fonctionnalités

Afin d'intégrer les fonctionnalités développées ci-dessus dans le noyau du côté client, nous avons décidé de faire appel aux fonctions concernées suivant le choix de l'utilisateur (ce choix est également passé dans la socket en écriture pour informer le serveur). Si l'utilisateur choisit d'envoyer des fichiers au serveur, la fonction **envoiServeur()** est appelée. Si l'utilisateur choisit de récupérer des images depuis le serveur, la fonction **telechargementServeur()** est appelée. L'appel de telles fonctions principales provoque un enchaînement d'appels des différentes fonctions des 2 côtés, Client ET Serveur.

Du côté serveur, le fonctionnement est pareil. La lecture du choix d'action fait par l'utilisateur se fait dans la socket et en fonction. Si l'utilisateur choisit d'envoyer des fichiers au serveur, les fonctions appelées dans l'ordre sont :

1. **recupereListeImagesServeur()** qui liste les images présentes chez le serveur dans un tableau.
2. **envoiListeImagesServeurClient()** qui envoie au client cette liste en écrivant dans la socket.
3. **recupereListeImagesATelecharger()** qui récupère un tableau des images choisies par l'utilisateur au niveau du client (lecture dans la socket).
4. **envoiImage()** qui envoie une par une les images demandées par l'utilisateur.

Si l'utilisateur choisit de récupérer des images depuis le serveur, les fonctions appelées dans l'ordre sont :

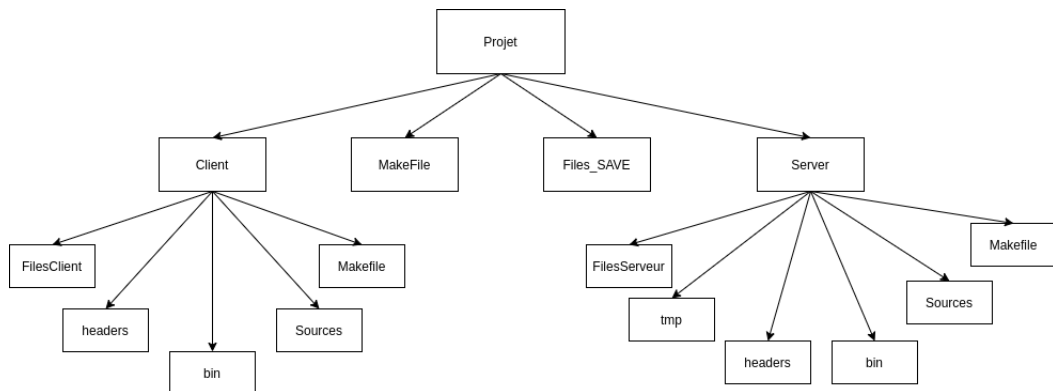
1. Après la lecture dans la socket du nombre de fichier que l'utilisateur souhaite envoyer au serveur, et pour chaque fichier, **receptionImage()** qui se chargera de "lire" les fichiers à garder, et qui fera appel à la vérification de l'admissibilité (enchaînement d'appels de fonctions...). Si l'utilisateur choisit de quitter ou ferme brusquement l'application, la socket est fermée.

Chapter 5

Mini manuel pour l'exécution de l'application

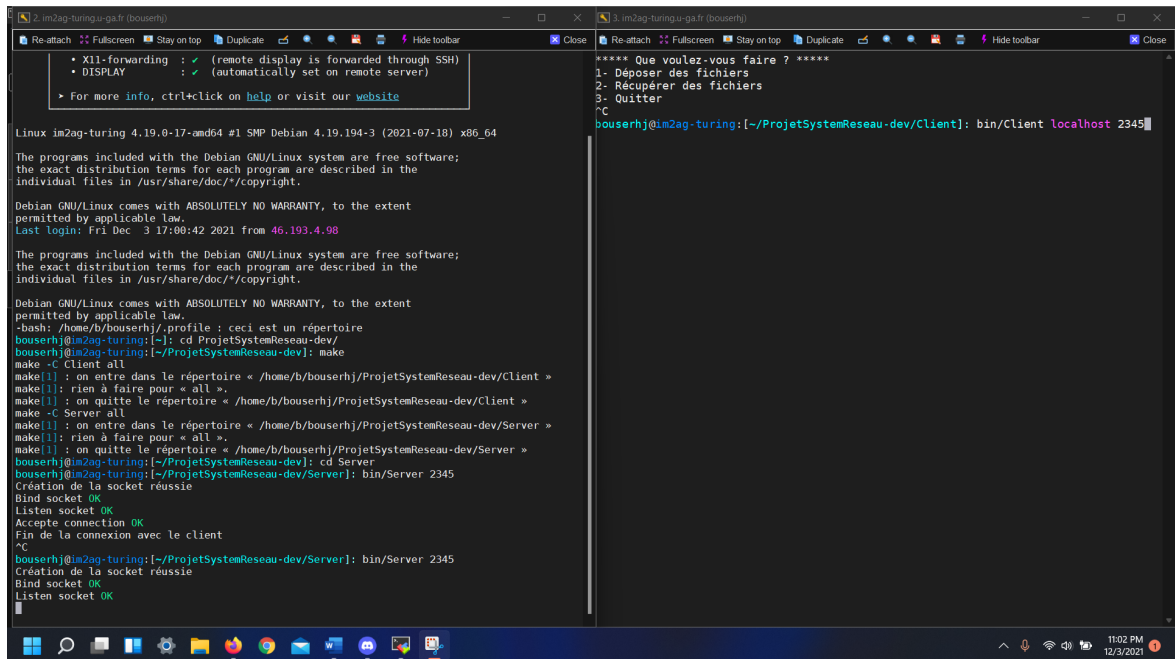
Organisation du répertoire et Makefile

Nous avons organisé notre répertoire de projet de la manière suivante :



Les répertoires Client et Server contiennent chacun un répertoire Files contenant les images qu'ils possèdent, un répertoire headers contenant les définitions des fonctions, un Source contenant tous les fichiers .c, un bin contenant leur exécutable, et un MakeFile. Le répertoire Server contient en plus un répertoire tmp, qui accueille les images temporairement lors de l'envoi d'images du client au serveur, afin de vérifier leur admissibilité. Chacun des 2 Makefile créent l'exécutable pour les programme Client et Serveur. Un dernier Makefile dans le répertoire Projet s'occupe d'exécuter les 2 Makefile des répertoires Client et Serveur afin de pouvoir tout compiler d'un coup. Pour lancer le programme Client, il faut lui passer 2 paramètres : le nom de la machine abritant le serveur, et le numéro du port utilisé. Pour lancer le programme Serveur, il faut lui passer en paramètre le numéro de port utilisé. Les 2 numéros de port cotés client et serveur doivent bien sûr être les mêmes.

Interface et fonctionnement de l'application



```
Linux im2ag-turing 4.19.0-17-amd64 #1 SMP Debian 4.19.194-3 (2021-07-18) x86_64

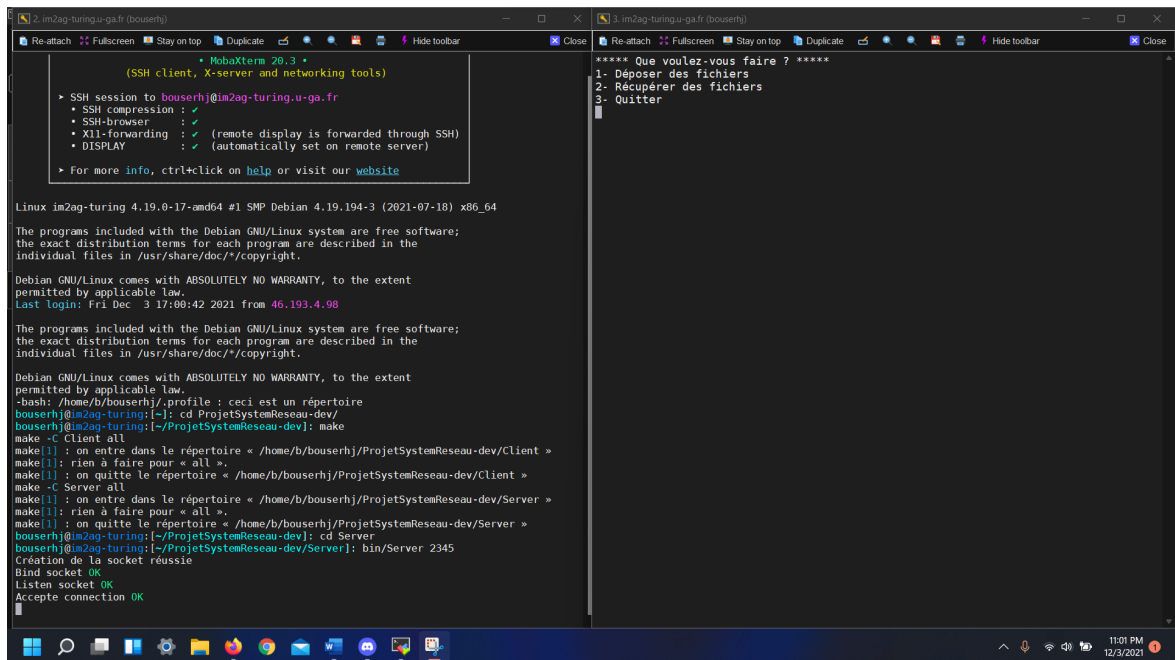
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Dec 3 17:00:42 2021 from 46.193.4.98

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
-bash: /home/b/bouserhj/.profile : ceci est un répertoire
bouserhj@im2ag-turing:~$ cd ProjetSystemReseau-dev/
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev$ make
make -C Client all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make -C Server all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev$ cd Server
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
listen socket OK
Accepte connection OK
Fin de la connexion avec le client
^C
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
listen socket OK
```

Juste après l'exécution du Client



```
Linux im2ag-turing 4.19.0-17-amd64 #1 SMP Debian 4.19.194-3 (2021-07-18) x86_64

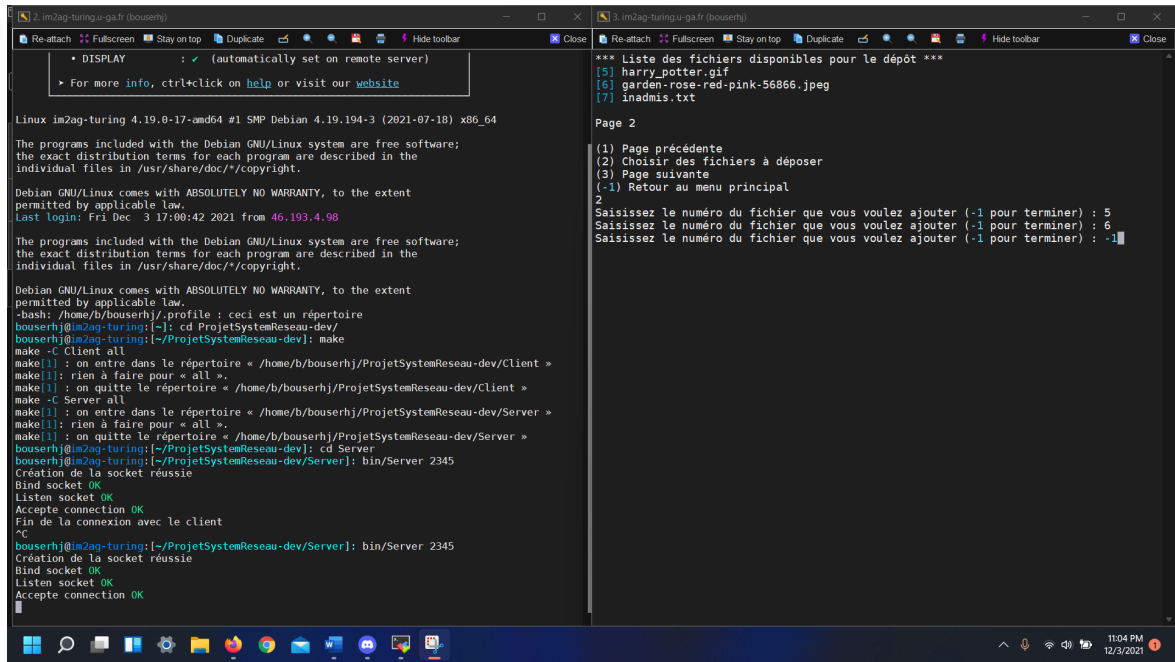
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Dec 3 17:00:42 2021 from 46.193.4.98

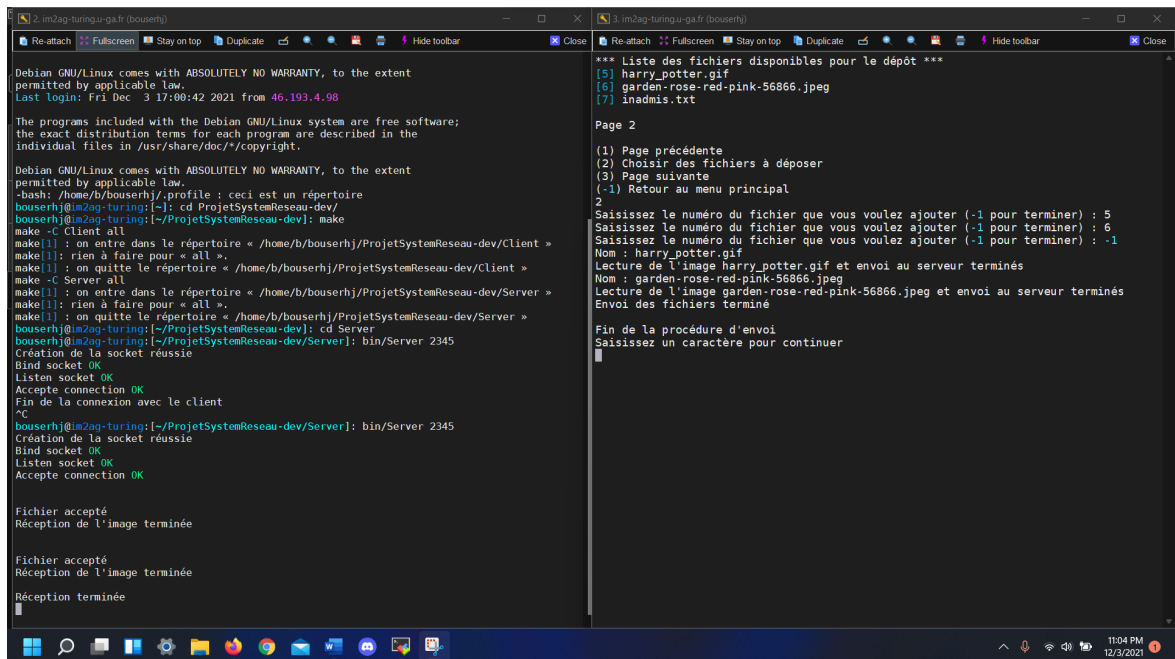
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
-bash: /home/b/bouserhj/.profile : ceci est un répertoire
bouserhj@im2ag-turing:~$ cd ProjetSystemReseau-dev/
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev$ make
make -C Client all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make -C Server all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev$ cd Server
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
listen socket OK
Accepte connection OK
```

Choix de l'option 1, Déposer des fichier (sur le serveur)



Dépôt réussi des fichiers



Dans le cas où un des fichiers n'est pas admissible

```
2. in2ag-turing-ga.fr (bouserhj)
make -C Client all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make -C Server all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev$ cd Server
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK
Fin de la connexion avec le client
^C
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier accepté
Réception de l'image terminée

Fichier accepté
Réception de l'image terminée

Réception terminée
Accepte connection OK
Erreur ouverture image
^C
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier refusé (type inode/x-empt non admis)
Réception de l'image terminée

Réception terminée
^C
```

```
3. in2ag-turing-ga.fr (bouserhj)
*** Liste des fichiers disponibles pour le dépôt ***
[5] harry_potter.gif
[6] garden-rose-red-pink-56866.jpeg
[7] inadmis.txt

Page 2
(1) Page précédente
(2) Choisir des fichiers à déposer
(3) Page suivante
(-1) Retour au menu principal
2
(1) Page précédente
(2) Choisir des fichiers à déposer
(3) Page suivante
(-1) Retour au menu principal
2
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 7
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : -1
Nom : inadmis.txt
Lecture de l'image inadmis.txt et envoi au serveur terminés
Envoi des fichiers terminé

Fin de la procédure d'envoi
Saisissez un caractère pour continuer
^
```

Saisie de l'option 2, Récupérer des fichiers (téléchargement depuis le serveur vers le client) et choix des images à télécharger : (puis -1 pour marquer la fin du choix)

```
7. in2ag-turing-ga.fr (bouserhj)
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Dec 3 16:58:42 2021 from 46.193.4.98

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
bosh: /home/b/bouserhj/.profile : ceci est un répertoire
bouserhj@in2ag-turing:~$ cd ProjetSystemReseau-dev/
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev$ make
make -C Client all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make -C Server all
make[1]: on entre dans le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
make[1]: rien à faire pour « all ».
make[1]: on quitte le répertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev$ cd Server/
bouserhj@in2ag-turing:~/ProjetSystemReseau-dev/Server$ bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier accepté
Réception de l'image terminée

Fichier accepté
Réception de l'image terminée

Fichier accepté
Réception de l'image terminée

Réception terminée
^C
```

```
8. in2ag-turing-ga.fr (bouserhj)
*** Liste des images du serveur ***
[1] perl5-sandbox-murhar-3852361.jpg
[2] mickey.ppm
[3] mumsouth.ppm
[4] image-soel-9.png

Page 1
(1) Page précédente
(2) Choisir des fichiers télécharger
(3) Page suivante
(-1) Retour au menu principal
2
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 2
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 3
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : -1
```

Téléchargement des images réussi

```
2. im2ag-turing-u-ga.fr (bouserhj)
Re-attach Fullscreen Stay on top Duplicate Hide toolbar
make[] : on quitte le repertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make -C Server all
make[] : on entre dans le repertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
make[] : rien à faire pour « all ».
make[] : on quitte le repertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev$ cd Server
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ ./bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK
Fin de la connexion avec le client
^C
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ ./bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier accepté
Réception de l'image terminée

Fichier accepté
Réception de l'image terminée

Réception terminée
Accepte connection OK
Erreur ouverture image
^C
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ ./bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier refusé (type inode/x-empty non admis)
Réception de l'image terminée

Réception terminée
Lecture de l'image et transmission terminées
Lecture de l'image et transmission terminées
Lecture de l'image et transmission terminées

3. im2ag-turing-u-ga.fr (bouserhj)
Re-attach Fullscreen Stay on top Duplicate Hide toolbar
*** Liste des images du serveur ***
[1] pexels-sanaan-mazhar-3052361.jpg
[2] mickey.ppm
[3] mammoth.ppm
[4] image-noel-9.png

Page 1
(1) Page précédente
(2) Choisir des fichiers télécharger
(3) Page suivante
(-1) Retour au menu principal
2
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 1
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 6
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 3
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : -1
Fin de l'envoi de la liste d'images à télécharger
Téléchargement de l'image terminé
Téléchargement de l'image terminé
Téléchargement de l'image terminé
Saisissez un caractère pour continuer
```

Saisie de l'option 3 (Quitter)

```
2. im2ag-turing-u-ga.fr (bouserhj)
Re-attach Fullscreen Stay on top Duplicate Hide toolbar
make[] : on quitte le repertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Client »
make -C Server all
make[] : on entre dans le repertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
make[] : rien à faire pour « all ».
make[] : on quitte le repertoire « /home/b/bouserhj/ProjetSystemReseau-dev/Server »
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev$ cd Server
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ ./bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK
Fin de la connexion avec le client
^C
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ ./bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier accepté
Réception de l'image terminée

Fichier accepté
Réception de l'image terminée

Réception terminée
Accepte connection OK
Erreur ouverture image
^C
bouserhj@im2ag-turing:~/ProjetSystemReseau-dev/Server$ ./bin/Server 2345
Création de la socket réussie
Bind socket OK
Listen socket OK
Accepte connection OK

Fichier refusé (type inode/x-empty non admis)
Réception de l'image terminée

Réception terminée
Lecture de l'image et transmission terminées
Lecture de l'image et transmission terminées
Lecture de l'image et transmission terminées

3. im2ag-turing-u-ga.fr (bouserhj)
Re-attach Fullscreen Stay on top Duplicate Hide toolbar
*** Liste des images du serveur ***
[1] pexels-sanaan-mazhar-3052361.jpg
[2] mickey.ppm
[3] mammoth.ppm
[4] image-noel-9.png

Page 1
(1) Page précédente
(2) Choisir des fichiers télécharger
(3) Page suivante
(-1) Retour au menu principal
2
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 1
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 6
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : 3
Saisissez le numéro du fichier que vous voulez ajouter (-1 pour terminer) : -1
Fin de l'envoi de la liste d'images à télécharger
Téléchargement de l'image terminé
Téléchargement de l'image terminé
Téléchargement de l'image terminé
Saisissez un caractère pour continuer
```