BRAY Amandine
TURC Julien
VOLAT Louis
ZAETTA Lucas

L3 MIAGE

Projet système et réseau

Sommaire :

I. Introduction

Le sujet que nous devons réaliser est la mise en place d'une application client/serveur qui permet de partager des images. Le client doit consulter, déposer et récupérer un fichier sur le serveur.

Ce projet va nous permettre de mettre en place ce que nous avons appris en cours. Tout d'abord en réseau, nous allons mettre en place un réseau client/serveur en utilisant le protocole TCP/IP qui va permettre de faire circuler les données grâce aux sockets. Enfin, dans la partie système, nous allons mettre en place des processus père et fils qui vont traiter les informations reçues .

II. Conception logiciel

Pour gérer notre projet, nous avons mis en place un git pour travailler en collaboration et en gestion de projet (sous forme de pull request et de cards).

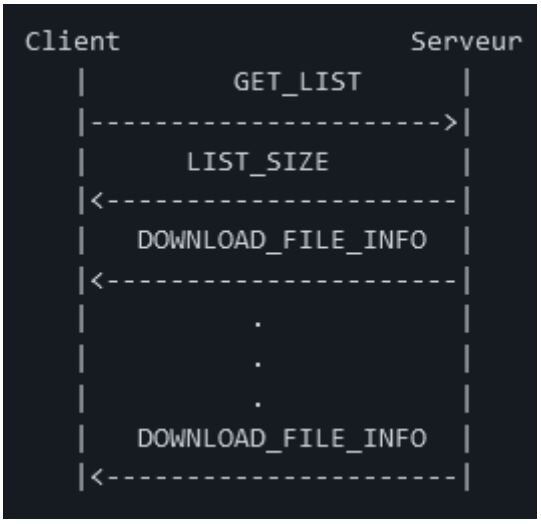
Pour conceptualiser le logiciel, nous avons décidé que le client, via une interface de commande pourra :

- consulter et récupérer des fichiers,
- déposer un ou plusieurs fichiers.

A. Consulter des fichiers

Pour consulter la liste des fichiers disponibles, le client commence par envoyer un message de type **GET_LIST** au serveur. Le serveur répond avec un message de type **LIST_SIZE** contenant le nombre de trame de fichier à récupérer. Il poursuit ensuite par l'envoi d'autant de messages de type **DOWNLOAD_FILE_INFO** qu'indiqué dans le message **LIST_SIZE**.

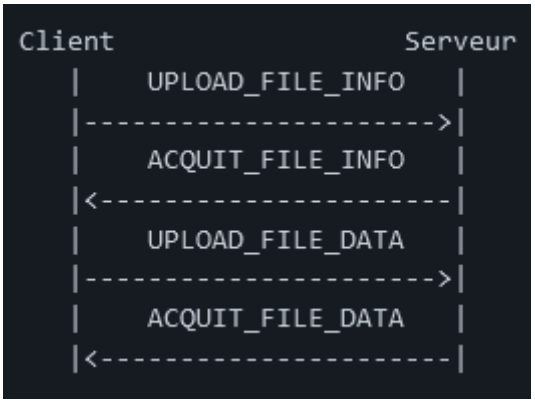
Echanges d'informations entre client et serveur



B. Déposer un fichier sur le serveur

Le client envoie un message de type **UPLOAD_FILE_INFO** au serveur. Le serveur répond par un message de type **ACQUIT_FILE_INFO** qui confirme ou annule le dépôt. Si le serveur accepte le dépôt, le client envoie le fichier dans un message de type **UPLOAD_FILE_DATA**. Le serveur répond alors par un message de type **ACQUIT_FILE_DATA** pour confirmer ou annuler le dépôt.

Echange d'information pour déposer entre un client et un serveur



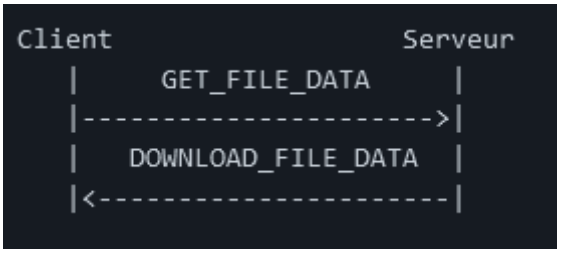
Avant d'envoyer le **ACQUIT_FILE_DATA**, nous allons vérifier que le type MIME du fichier correspond bien à une image. Pour cela nous avons créé une fonction getType qui extrait l'information du résultat de la commande shell "file -i <nom du fichier>". Elle prend donc en paramètre le nom du fichier (avec son chemin relatif depuis la racine du projet).
Elle va ensuite créer un processus fils qui va lancer l'exécution de la commande *file -i* sur notre paramètre. Nous allons ensuite lire le contenu de cette commande via un tube.
Puis nous allons isoler la partie qui nous intéresse. Pour cela nous allons découper le contenu de la commande en une série de tokens en utilisant comme séparateur le caractère espace (ce qui permet donc d'avoir accès aux 3 parties du résultat du file -i).
Nous sélectionnons ensuite le token contenant la partie après le premier espace, que nous allons ensuite découper à nouveau en délimitant avec le caractère ';' présent à la fin de ce deuxième token.
Cette solution pose un problème, si un fichier contient des espaces dans son nom, nous n'allons pas sélectionner le bon token. Toutefois, nous pouvons identifier et gérer cette erreur. S'il y'a un espace ou plus dans le nom, nous allons utiliser, comme dans le premier cas le délimiteur avec ';'. Ce qui va englober la fin du nom de fichier ainsi que les ':' présent à la fin de celui-ci. Nous pourrons ainsi rechercher la présence de ces ':' pour savoir si un fichier contenait des espaces dans son nom, et dans ce cas traiter cette erreur.

```
tests/types/github-git cheat sheet.pdf: application/pdf; charset=binary
```

C. Récupérer un fichier sur un serveur

Le client envoie un message de type **GET_FILE_DATA** au serveur. Le serveur répond alors par un message de type **DOWNLOAD_FILE_DATA** pour envoyer le fichier.

Echange d'information pour récupérer entre un client et un serveur



D. Gestion des erreurs

Pour la gestion des erreurs, nous nous sommes inspirés de la façon dont les erreurs sont gérées avec Erno. Cela veut dire que nous avons mis en place une variable globale **SPP_Erno** qui contient les exceptions rencontrées. SPP_Erno est par défaut à -1. Lorsque une erreur est rencontrée, la fonction va la placer sur le bon code d'erreur. Code d'erreur correspondant à l'emplacement du message d'erreur dans le tableau **DDP_errList[]**. Ce qui nous laisse ensuite deux options :

- Faire appeler la fonction SPP_perror() pour avoir le message d'erreur correspondant au code d'erreur
- Directement l'imprimer dans la sortie erreur avec DDP_errList[SPP_Erno].

Les erreurs SPP sont principalement renvoyées par les fonctions de vérification de l'encapsulation des données. Les fonctions de sérialisation et de désérialisation peuvent ensuite les propager en renvoyant un code d'erreur égal à 0xff sur l'emplacement de la commande (soit frame[0], soit struc->cmd). Si le code d'erreur est détecté, on peut alors faire appel à la fonction SPP_perror() pour avoir le message d'erreur correspondant.

III. Conception du protocole

A. Les messages

Pour le protocole, nous avons mis en place différents types de champs qui permettent de gérer plus facilement nos commandes et nos erreurs.

Voici les différentes erreurs que nous pouvons avoir après la saisie du client :

- 0x20 : Accepté,
- 0x41 : Refusé : Nom de fichier invalide,
- 0x42 : Refusé : Extension de fichier invalide,
- 0x43 : Refusé : Nom de fichier déjà pris,
- 0x44 : Refusé : Fichier introuvable,
- 0x45 : Refusé : Fichier ne respecte pas la checksum,
- 0x50 : Refusé : Erreur interne,
- 0x51 : Refusé : Plus de place disponible,
- 0x52 : Refusé : Fichier trop volumineux.

Ensuite, nous avons mis en place différentes commandes pour le client (commençant par la lettre C), pour le serveur (commençant par la lettre A) et l'envoie de la donnée du fichier (finissant par la lettre D).

Commandes serveur :

- 0xA1 : LIST_SIZE,
- 0xA2 : DOWNLOAD_FILE_INFO,
- 0xA3 : ACQUIT_FILE_INFO,
- 0xA4 : ACQUIT_FILE_DATA,
- 0xAD : DOWNLOAD_FILE_DATA.

Commandes client :

- 0xC1 : GET_LIST,
- 0xC2 : UPLOAD_FILE_INFO,
- 0xC3 : GET_FILE_DATA,
- 0xCD : UPLOAD_FILE_DATA,
- 0xCE : CLOSE_SOCKET.

Avec le langage C, nous insérons les commandes sous forme d'énumération. Cette façon est plus simple et permet d'insérer plus facilement nos éléments et aussi d'apprendre à gérer par nous même nos erreurs et les commandes.

Enumération des commandes

```
enum COMMANDES {
    LIST_SIZE=0xA1,
    DOWNLOAD_FILE_NAME=0xA2,
    ACQUIT_FILE_NAME=0xA3,
    ACQUIT_FILE_DATA=0xA4,
    DOWNLOAD_FILE_DATA=0xAD,
    GET_LIST=0xC1,
    UPLOAD_FILE_NAME=0xC2,
    GET_FILE_DATA=0xC3,
    UPLOAD_FILE_DATA=0xCD,
    CLOSE_SOCKET=0xCE
};
```

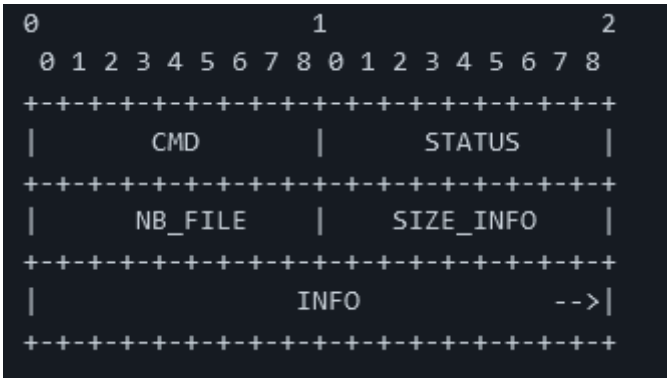
Enumération des statuts

```
enum STATUS{
    SUCCESS=0x20,
    INVALID_NAME_FILE= 0x41,
    INVALID_EXTEND_FILE= 0x42,
    NAME_ALREADY_TAKEN = 0x43,
    NO_FOUND_FILE= 0x44,
    CHECKSUM_ERROR= 0x45,
    INTERNAL_ERROR= 0x50,
    LACK_OF_SPACE= 0x51,
    FILE_TOO_LARGE= 0x52
};
```

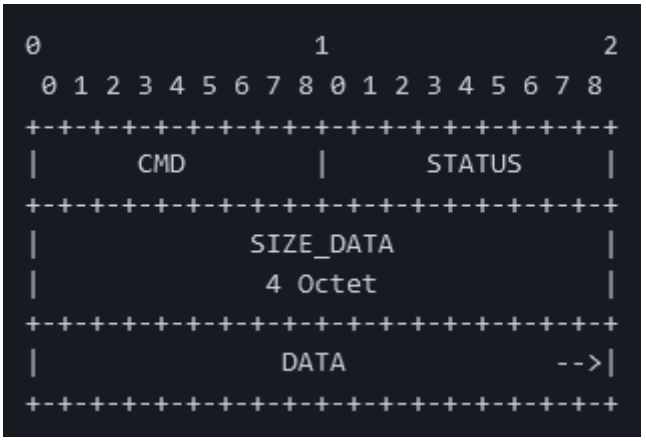
B. Les trames

Nous avons mis en place deux trames différentes. Une première trame nommée **INFOS_TRAME** qui possède les informations connexes à l'envoi de fichiers. La seconde trame **DATA_TRAME** contient l'entête du fichier. Lors des envois des trames, nous envoyons seulement l'entête de chaque trame au serveur puis nous envoyons les données.

Trame de INFO_TRAME



Trame de DATA_TRAME



C. Implémentation en C

Pour l'implémentation de notre protocole en C, nous avons effectué différents choix. En effet, nous avons décidé de mettre en place un système de sérialisation et désérialisation des trames. Cette méthode nous permet d'envoyer des trames sous forme de unsigned char* et de les récupérer et les traiter sous forme de structure plus simple à manipuler.

C1. Structure

Pour essayer de ressembler au protocole nous avons décidé de faire deux structures différentes, une par trame. Pour chaque structure, toujours par simplicité, nous avons décidé de mettre en place un type custom permettant d'accéder au pointeur de la trame.

Structure en _InfosTrame

```c
typedef struct _InfosTrame{
    unsigned char cmd;
    unsigned char status;
    unsigned char nbFiles;
    unsigned char sizeInfos;
    char* infos;
} InfosTrame;

typedef InfosTrame * PInfoTrame;
```

Structure de _DataTrame

```
typedef struct _DataTrame{
    unsigned char cmd;
    unsigned char status;
    unsigned int sizeData;
    int dataFd;
} DataTrame;

typedef DataTrame * PDataTrame;
```

Sur ces deux images, nous pouvons voir que les noms correspondent aux noms sur les schémas des trames.

D. Sérialisation

Les fonctions de sérialisation nous permettent de convertir une structure en unsigned char* dans le but de l'envoyer. Elles ont aussi pour but d'encapsuler les données qu'elles contiennent afin de rester cohérent avec le protocole.

Ici nous avons décidé de mettre en place deux fonctions de sérialisation (encodeInfosTrame() et encodeDataHead()). Une pour les trames de type INFO_TRAME et une pour les trames de type DATA_TRAME.

Chacune de ces fonctions fait appel à une fonction de vérification de l'encapsulation des données (**checkDataTrameError()** et **checkInfoTrameError()**). Ces fonctions d'encapsulation d'erreur ont pour but de générer des erreurs d'exécution si les données ne sont pas cohérentes, afin de nous informer de potentielles mauvaises utilisations.

La sérialisation des trames de type **DATA_TRAME** est un peu particulière, en effet on ne sérialise pas les données directement, seulement l'entête de la trame. Les données, quant à elles, sont envoyées et traitées par le serveur ou le client directement dans le socket, après l'envoi de l'entête grâce au descripteur de fichier présent dans la structure DataTrame.

E. Désérialisation

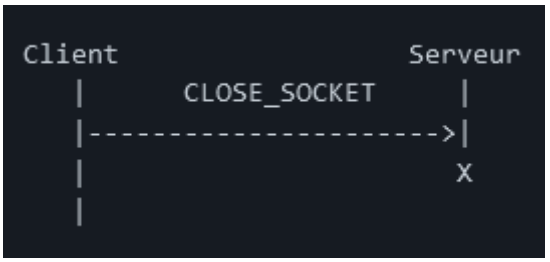
Les fonctions de désérialisation nous permettent de convertir une trame unsigned char* en structure dans le but de la traiter plus simplement dans le code. Elles sont l'exact inverse des fonctions de sérialisation. La particularité ici est que nous testons l'encapsulation des données une fois qu'elles sont désérialisées avec les mêmes fonctions que pour la sérialisation.

Ici nous avons décidé de mettre en place trois fonctions de désérialisation (decodeInfosTrame(), decodeInfosTrame_Infos() et decodeDataHeadTrame()). La particularité ici est que decodeDataHeadTrame() demande en paramètre le descripteur de fichier du socket pour le placer dans la structure. La fonction decodeInfosTrame() permet de décoder l'entête de la trame info, tandis que que la fonction decodeInfosTrame_Infos() permet de décoder le champ info de la trame infos (le nom du fichier).

F. Fermeture de la connexion avec le serveur

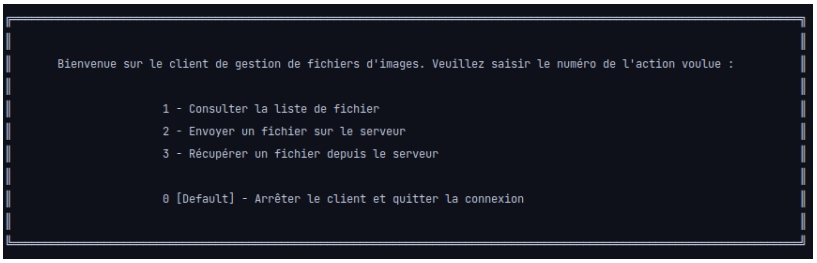
Pour informer le serveur d'une déconnexion. Le client envoie un message de type CLOSE_SOCKET au serveur. Le serveur ferme alors le processus de communication.

Schéma de la fermeture du serveur



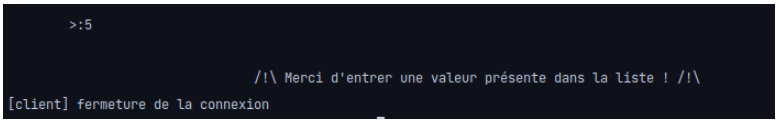
IV. Exemples d'expérimentations

Dans cette partie, nous pouvons voir que l'interface graphique est composée des actions qui sont associées à un numéro.



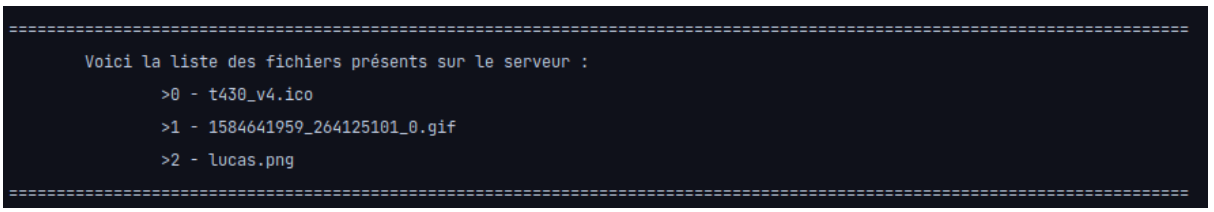
Interface du menu du client

Pour sortir du programme, il suffit de cliquer sur 0 ou sur n'importe quel caractère.



Cas ou nous saisissons une erreur

Pour consulter la liste des fichiers, il faut saisir le chiffre 1 et nous pouvons voir les différents fichiers disponibles sur le serveur qui s'affiche.

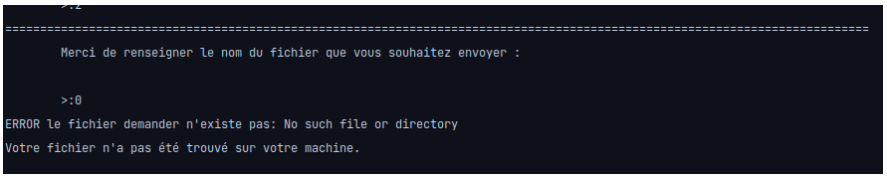


Liste des fichiers

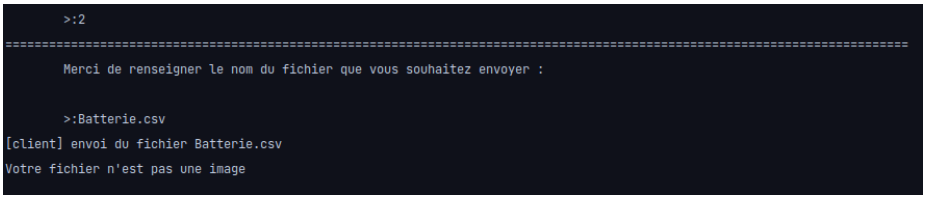
Pour envoyer un fichier sur un serveur, il faut cliquer sur 2 et renseigner le fichier souhaité (se trouvant dans le dossier data/client).

Interface pour la partie envoie

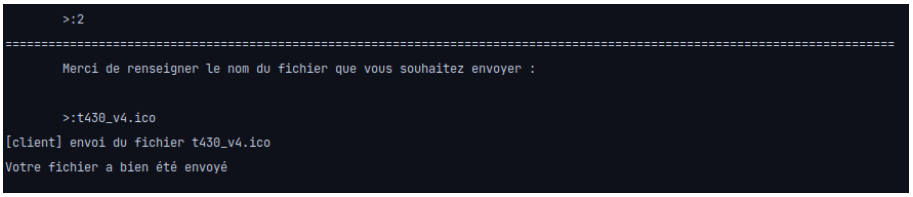
Si il envoie un nom inexistant ou à une erreur dans l'écriture de l'extension alors il renvoie une erreur sinon il indique que l'envoie à bien fonctionner.



Nom inexistant



Problème d'extension



Envoie réussi

Enfin, si il souhaite récupérer un fichier, il clique sur le 3 et sélectionne celui que l'utilisateur souhaite.

Interface pour le cas de la récupération

L'utilisateur saisit le chiffre et le reçoit.

```
Voici la liste des fichiers présents sur le serveur :
        >0 - t430_v4.ico
        >1 - 1584641959_264125101_0.gif
        >2 - lucas.png
Quel fichier voulez vous récupérer ? Merci d'entrer le numéro correspondant:
>:2
lucas.png
Voulez-vous récupérer un autre fichier ?
        Entrer 0 pour revenir au menu principal.
        Entrer 1 pour récupérer un autre fichier.
>:3
        /!\Merci de rentrer une valeur valide /!\
```

Saisie du chiffre

V. Tests driven et développement

Afin de faciliter la vérification de la fonctionnalité, nous avons décidé de faire du TDD (test driving development). Cette méthode signifie que nous avons commencé par rédiger nos headers, puis les tests unitaires des fonctions que nous avons prototypées, et enfin nous les avons implémentées en c en veillant à respecter les spécifications des tests.

Pour se faire, nous avons mis en place un dossier test qui contient toutes les fonctions de test. Comme nous ne connaissons pas les frameworks de tests unitaires, nous avons mis en place notre propre core de test dans le fichier test-core.c.

Ce core de test contient les fonctions de test de la fonctionnalité. Ainsi que quelques fonctions d'affichage utile. Le principe ici est le suivant :

- Un fichier de test par fichier c à tester
- Chaque fonction à tester doit avoir une ou plusieurs fonctions de test qui lui sont associées.
- Chaque fonction de test renvoie un boolean qui doit être vrai si le test est passé.
- Chaque fonction de test doit avoir un commentaire qui explique ce qu'elle fait.
- Toutes les fonctions de test sont appelées par une fonction de fichier C.
- Cette fonction de fichier C est déclarée dans all_tests.h et appelée dans all_tests.c.

Pour lancer les tests, nous nous sommes arrangés pour que notre makefile soit en mesure de les build séparément du code utilisable.

Nous avons donc créé la phony rule test qui permet de build les fichiers de test. Ensuite on peut lancer les tests en appelant le fichier run_tests dans le dossier bin.

VI. Makefile et build

Dans ce projet, nous avons décidé de respecter l'arborescence de fichiers suivante :

- **bin** pour les fichiers exécutables,
- **debug** pour les fichiers exécutables de debug ainsi que les .obj de debug,
- **doc** pour les fichiers de documentation,
- **headers** pour les fichiers headers,
- **obj** pour les fichiers objets,
- **sources** pour les fichiers sources,
- **tests** pour les fichiers de tests.

Pour commencer, nous avons d'abord indiqué dans notre makefile les fichiers à build un à un. Nous nous rendons bien compte que cela ne serait plus vraiment efficace pour la suite du projet. Par conséquent, nous avons essayé de l'implémenter nous même à l'aide des variables de configuration.

Or n'étant pas expert en makefile, nous avons donc décidé d'utiliser un template de makefile que nous avons modifié afin d'inclure nos headers et de permettre la compilation de nos tests unitaires.

Celui-ci dispose des commandes suivantes :

- **all** qui permet de build tous les fichiers du projet,
- **test** qui permet de build les fichiers de test,
- **debug** qui permet de build les fichiers du projet en mode debug,
- **clean** qui permet de clean tous les fichiers obj et bin du projet.

Nous avons aussi d'autres commandes annexes qui sont utilisées par les commandes précédente :

- **makedir** qui permet de créer les dossiers de build si ils n'existent pas
- **build-test** qui permet de build les fichiers de test sans créer les dossiers de build
- **distclean** qui permet de supprimer les dossiers de build et bin du projet

Par défaut nous appelons la commande all lorsque l'on exécute la commande make.

VII. Compiler et exécuter

Pour lancer l'application, il faut se placer à la racine du projet et avoir :

- un environnement Linux,

- un compilateur GCC installé,
- un Makefile installé.

Pour exécuter le projet, nous utilisons les commandes :

- $ make
- $ ./bin/mimage

Pour exécuter les tests unitaires, nous utilisons les commandes :

- $ make test
- $ ./bin/run_tests

VIII. Conclusion

Pour le moment, nous avons fait la fermeture du serveur, la consultation et la récupération des fichiers et l'upload des fichiers. Pour gérer la gestion des erreurs, nous avons mis en place des tests unitaires qui permettent de vérifier si nos programmes sont fonctionnels.

Mais nous n'avons pas fait pas la checksun et mis en place l'ensemble des gestion des erreurs sur le serveur car nous n'avons pas eu assez de temps.

Mais il est possible d'améliorer notre projet en restructurant les méthodes en sous méthode. Cela permet d'éviter d'avoir les mêmes morceaux de code identiques. Mais aussi de voir l'image reçue par le client directement sur le terminal.