

# Systemes-Réseaux



Beginmay KONUSHBAEVA  
Ilyass BOUISSA  
Rayan GUERBAA  
Houssemeddine FEZAI

03/11/2021  
Licence 3 MIAE

## Introduction

Nous avons eu pour projet de programmer une application de communication TCP/IP entre un serveur et des clients.

Nous devons écrire en langage C deux programmes, serveur et client, permettant de partager des fichiers images entre deux machines à distance.

Le programme serveur sera lancé sur une première machine et devra employer des protocoles pour recevoir des connexions externes et dialoguer avec les clients .

Le programme client quant à lui devra implémenter le comportement du client lors de la connexion avec le serveur. C'est-à-dire permettre au client d'effectuer certaines tâches qui vont initialiser le dialogue avec le client.

Le serveur devra pouvoir gérer:

- Les connexions simultanées des clients.
- Connexion entre le serveur et le client à distance.
- La cohérence des protocoles d'échange entre le serveur et un client.
- Pouvoir partager des fichiers images avec des clients connectés en simultané.

## Objectifs de ce projet

1. Permettre de relier les notions de réseaux et système.
2. Relier des équipements sur un serveur commun.
3. Élaboration d'une application d'échange serveur - client.
4. Envoyer et recevoir des informations (fichiers image, commande...) à un serveur

## Organisation du projet

Pour ce projet, nous avons procédé par étapes.

Phase I. Élaboration du noyau de client/serveur TCP/IP

Phase II. Conception de l'application et répartition des tâches

Phase III. Implémentation des fonction de transfert de fichier, et de vérification

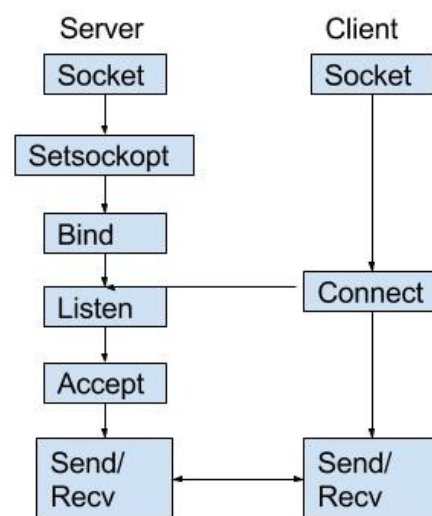
Phase IV. Assembler l'ensemble de nos réalisation dans le noyau client/serveur

## Phase I

Cette étape correspond à la conception d'un noyau client/serveur permettant de recevoir la connexion des différents clients au serveur et ainsi établir un dialogue entre machines.

Dans cette partie, nous expliquerons d'abord notre raisonnement pour créer correctement le serveur/client, puis les différentes notions abordées dans ce projet tel que les sockets. Le schéma ci-dessous représente la forme globale du noyau et des différents protocoles à employer.

Notre conception du noyau est très similaire au schéma, et correspond à ce que nous avons étudié en cours. Il manque seulement à préciser la création de processus fils lors d'une connexion client, la gestion des déconnexions des clients ne figurant pas sur le schéma et le fonctionnement et l'utilisation des sockets de communications entre différentes machines.



## Création de processus fils

Dans le noyau du serveur, nous devons veiller à ce que le serveur reste toujours disponible pour accepter de nouvelles connexions clients. Si un client se connecte, le serveur doit pouvoir traiter ce client et revenir à l'état disponible. Pour cela, nous effectuons une boucle afin de détecter la connexion d'un client. Puis nous utilisons la fonction fork pour générer des processus fils qui vont dialoguer avec celui-ci, et donc ne pas affecter la fonction du processus principal.

## Gestion des déconnexions client

Pour détecter les déconnexions des clients dans le serveur nous utilisons la fonction sigaction pour intercepter les signaux de mort des processus fils dans une fonction appelée handler.

cette fonction est placée au début du code et va automatiquement réceptionner les signaux de mort des fils sans déranger le comportement du processus principal qui doit rester disponible pour accepter de nouveaux clients

## Notions abordées

### Sockets

La communication entre les machines dans ce projet est réalisée par l'utilisation des "sockets". Pour connecter les machines, les sockets sont attachés à leurs adresses.

#### Du côté serveur :

Initialement, le serveur crée et attache une socket d'écoute pour recevoir les demandes de connexion, et utilise la primitive "écoute" pour ouvrir le service de connexion

Ce socket attend une demande de connexion.

On initialise toutes les caractéristiques de la structure de ce socket et on lui associe un port précis que l'on utilisera pour connecter le client. La primitive bind nous permet de rattacher toutes ces caractéristiques aux descripteur de fichier de ce socket créé.

#### Du côté client :

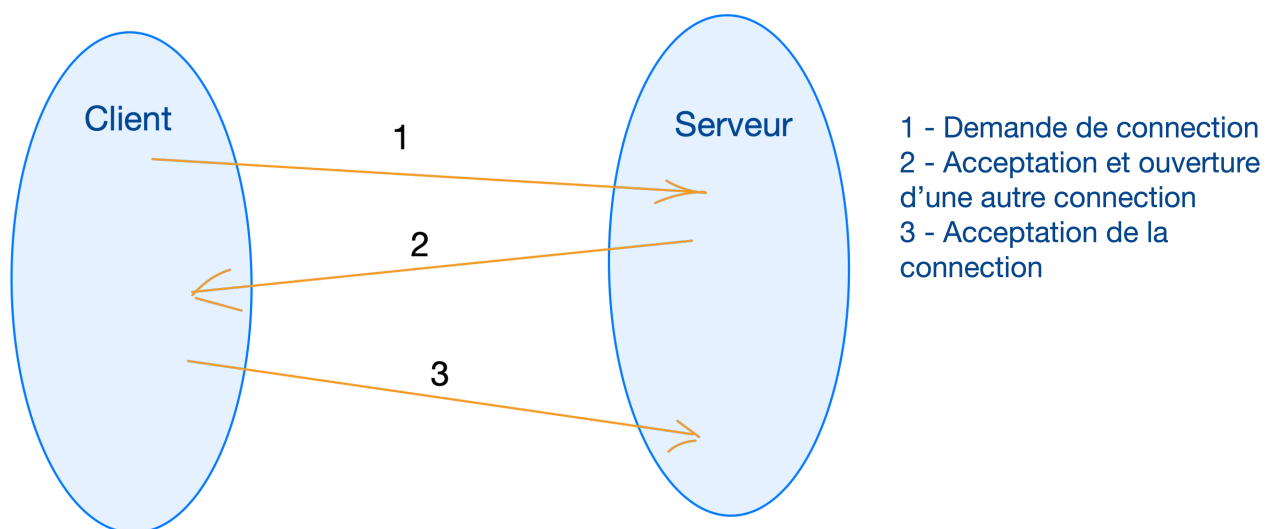
Le client quant à lui, envoie une demande de connexion en indiquant bien l'adresse IP du serveur et le port.

Avant toute chose, le client crée son socket d'envoi qui va lui servir pour établir une connexion avec le serveur.

## Protocole d'échange TCP/IP

Le protocole TCP est un protocole dit connecté. Il contrôle si le paquet est arrivé à destination si ce n'est pas le cas il le renvoie. Ainsi on évite une perte de données des paquets, le protocole est donc plus fiable c'est pourquoi on a choisi de s'échanger les données en utilisant ce protocole pour garder toutes nos informations des images lors de l'envoi.

## Utilisation de la Socket



Chaque processus client et serveur possèdent une socket, toutes les deux sont connectées par un canal externe comprenant un port et une adresse internet . Ici on choisit de se contenter d'utiliser seulement 2 sockets, une socket d'envoi et une socket de réception.

### **Implémentation du comportement serveur :**

On crée le fichier serveur qui va implémenter notre serveur pour, cela on crée notre socket d'écoute et on initialise la socket avec le port de notre machine serveur, et un port de connexion identique au client. On attache ensuite la socket aux informations de la structure grâce à bind. Puis on ouvre le service et le serveur est sous écoute.

### **Implémentation du comportement client:**

Dans un premier temps on crée le fichier client qui va implémenter notre serveur, pour cela on crée notre socket d'envoi. cette socket sera initialisée avec l'adresse internet de la machine serveur et un port de connexion.

Lors du dialogue le client envoie un message qui correspond à une chaîne de caractère qu'il entre en entrée standard et l'envoie au serveur à travers notre socket d'envoi. Puis il reçoit la réponse du serveur et l'affiche.

Puis dans une boucle infinie il accepte une connexion avec le client si elle est acceptée.

La gestion du dialogue avec un client est déléguée au fils afin que le serveur puisse établir d'autres connexions avec d'autres clients en parallèle. Lors du dialogue le serveur reçoit un message du client et le lit puis le serveur envoie une réponse qui correspond à une chaîne de caractère qu'on entre en entrée standard sur le serveur.

Une fois le noyau correctement implémenté, nous pouvons passer à la suite du projet, qui correspond à la conception des protocoles d'échanges entre le serveur et le client.

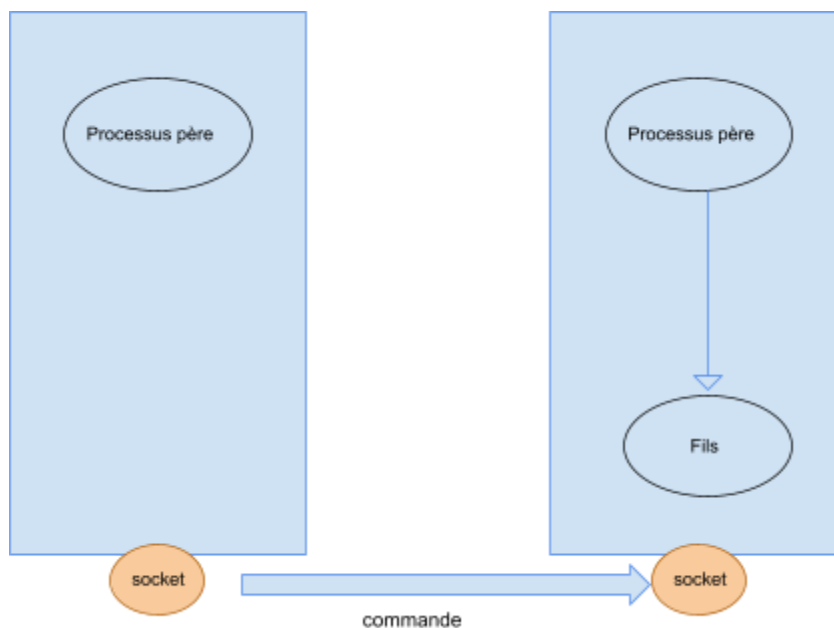
## Phase II

Dans cette partie du projet, nous devons concevoir les différents protocoles d'échanges entre le serveur et les clients. Dans un premier temps, nous présenterons les schémas de conception sur lesquels nous nous sommes mis d'accord, puis nous expliquerons notre raisonnement, les notions abordées et ce que nous avons pu et voulu réaliser .

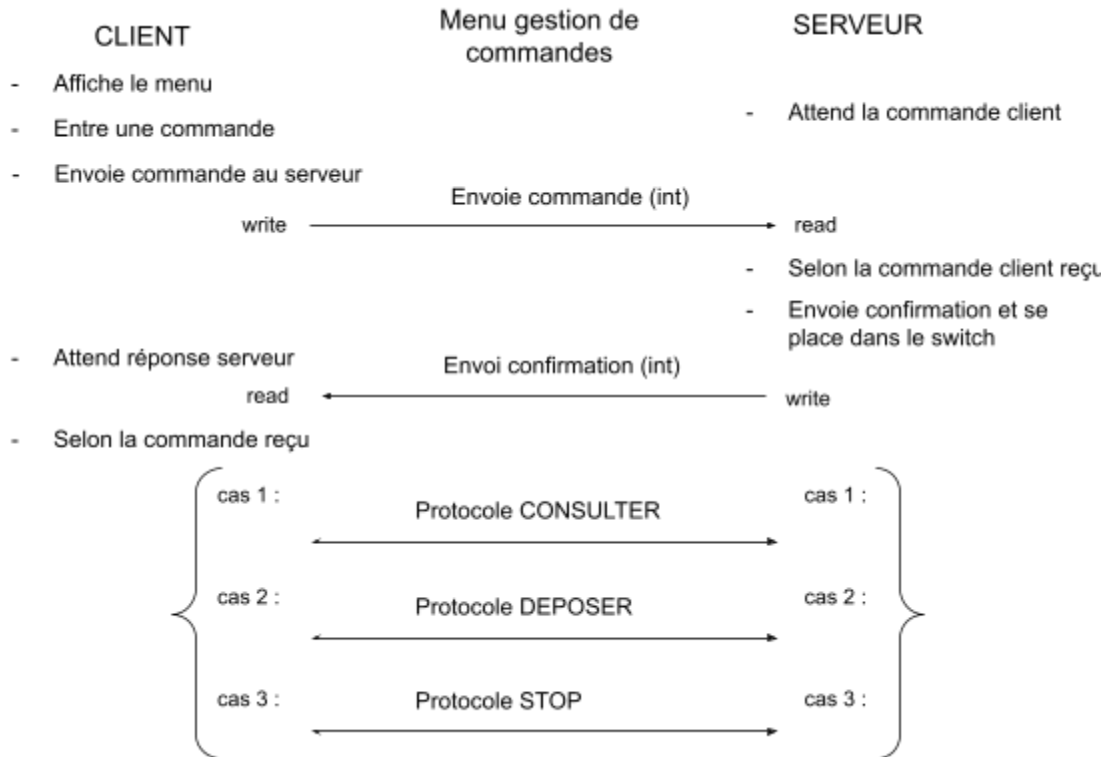
### Architecture générale de l'application Serveur - Client

Pour cela, nous avons conçu une première modélisation générale correspondant au comportement du serveur et du client après la connexion.

Comme il a été expliqué précédemment, le processus serveur crée un fils pour communiquer avec le processus client. La communication entre les 2 processus étant à distance sur des machines différentes, les échanges s'effectuent à travers une socket.



C'est dans ce corps du code que l'on affiche le menu, les différentes commandes possibles (consulter (1), déposer (2) ou se déconnecter (3)) puis synchroniser le serveur et le client dans le bon protocole d'échange.



A chaque envoi de commande par le client le serveur envoie un accusé de réception à celui-ci avant de se placer dans le bon protocole et ainsi pouvoir exécuter la commande demandée.

### Les différentes étapes effectuées par le Serveur

1. Démarre le programme
2. Attend une connexion
3. Créer un processus fils pour chaque nouvelle connexion
4. Reçoit une commande depuis le client
5. Confirme la réception de cette commande en l'envoyant au client
6. Exécute la commande reçue

### Le comportement du serveur en fonction de la commande :

- La commande 1: Consulter la liste des fichiers disponibles

Les fichiers disponibles sont situés dans le dossier Files depuis le répertoire courant du serveur. On récupère côté serveur, la liste des fichiers en définissant un pointeur pour le répertoire d'entrée afin d'accéder à la structure puis on compare si opendir échoue ou non lorsqu'on veut accéder au répertoire et on récupère dans notre buffer la liste qu'on concatène par la suite dans notre buffer un par un.

Puis on write au client afin de préparer le message qu'on va afficher pour le client

Puis on demande à l'utilisateur s' il souhaite déposer une image dans son répertoire en entrant un nom de fichier de la liste affichée si le type MIME du fichier est présent dans le fichier MimeTypes.txt.

- La commande 2 : Déposer un fichier

Le serveur crée un fichier et y place les données préalablement envoyé par le client dans une socket seulement si le type MIME du fichier correspond aux types présent dans le fichier MimeTypes.txt qui énumère tous les fichiers acceptables

- La commande 3 : Quitter serveur

Déconnecte le client du serveur.

### **Les différentes étapes effectuées par le Client**

1. Démarre le programme
2. Établit une connection avec le serveur
3. Procède à l'affichage de l'interface client avec le choix de commandes
4. La commande saisie sera envoyé au serveur
5. Vérifie la réception de la même commande de serveur
6. Change de comportement en fonction de la commande

### **Le comportement du client en fonction de la commande :**

- La commande 1 : CONSULTER, consulte la liste des noms de fichiers image disponibles.

- La commande 2 : DÉPOSER, dépose un fichier du client au serveur.

L'interface demande au client le nom du fichier souhaité (se trouvant sur la machine de client)

Le contenu de ce fichier est ensuite recopié dans la socket et envoyé à travers celui-ci dans un fichier du même nom vide côté serveur qu'on va remplir avec les données que la socket transporte.

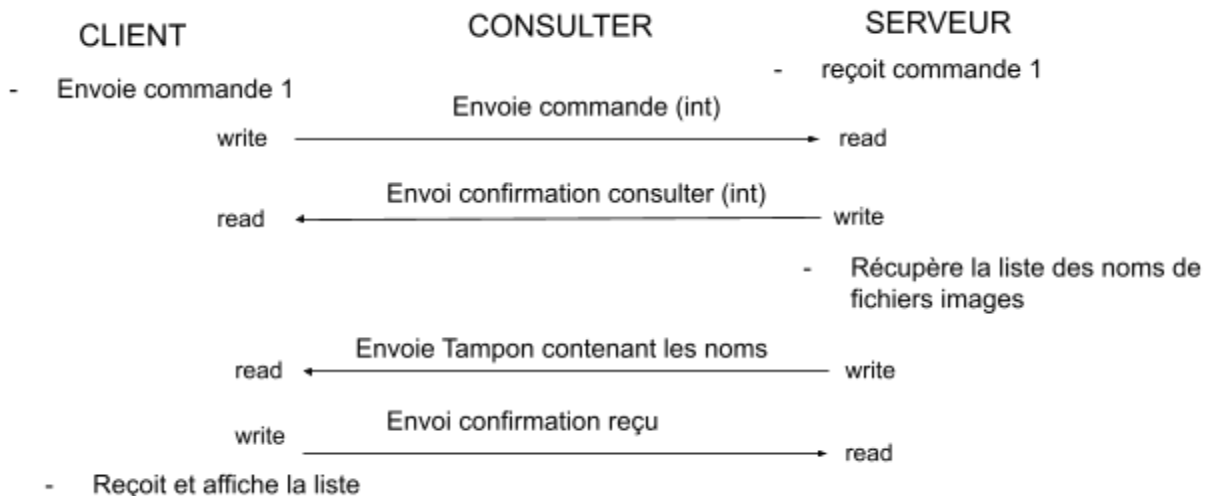
- La commande 3: STOP, permet de quitter le serveur

Pour ce faire, le client envoie la commande stop puis une fois la commande reçue par le serveur. Celui-ci renvoie un qu'il a bien reçu la commande du client et effectue la déconnection . Confirme la déconnection.



## Conception de la commande Consulter

Pour cette partie on fait la conception du protocole d'échange permettant d'afficher une liste de noms de fichier du côté client. Le schéma ci-dessous représente le schéma général de ce protocole.



## Réflexion

Pour entrer dans le cas de la consultation d'image, il suffit du côté client, d'entrer la commande associée pour afficher la liste de noms de fichiers.

Dans notre cas, nous avons décidé de dissocier la partie consultation de la liste d'images et la demande au client dans le cas où il souhaite effectuer un téléchargement de un ou plusieurs fichiers images.

Pour cela, nous avons préféré ajouter une option de commande sur le menu client permettant d'uploader une image sur le serveur. Cependant, en raison de certains bugs apparus tardivement dans la semaine, nous avons décidé de retirer la fonction d'upload (téléchargement d'une image du serveur au client) pour garder un code fonctionnel. Le client ne peut donc pas obtenir d'images du serveur.

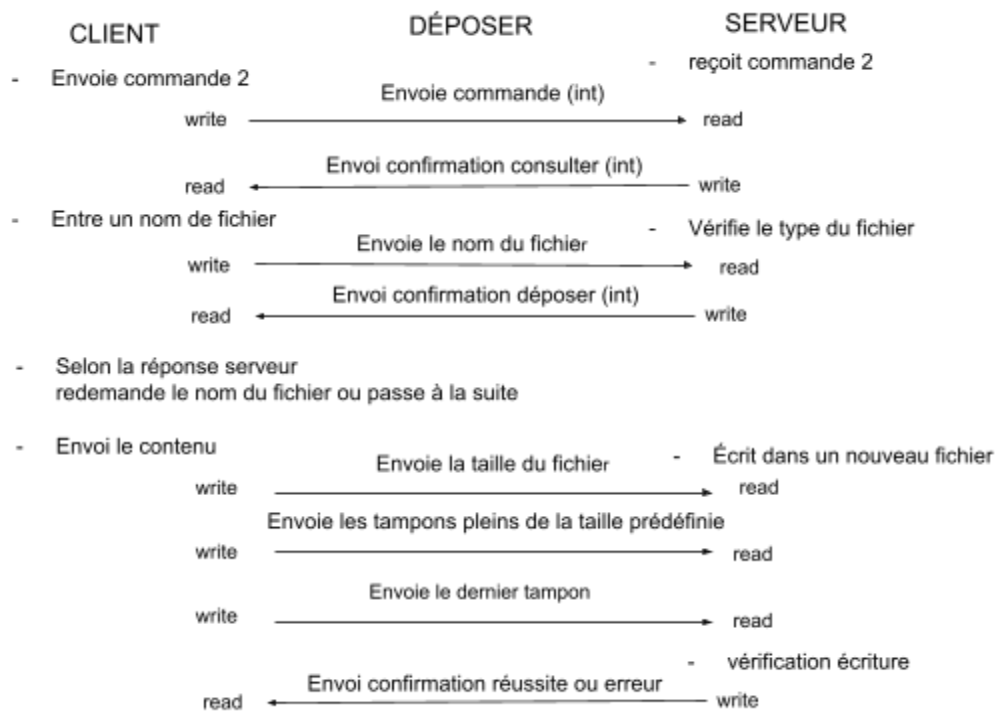
Nous aurions souhaité ajouter cette fonctionnalité directement après le consulter, en ajoutant dans le code un affichage demandant au client quel fichier il souhaite télécharger, puis un scanf permettant d'obtenir le fichier souhaité.

Puis nous aurions voulu ajouter la possibilité de télécharger 1 ou plusieurs fichiers en lui demandant par exemple de les écrire à la suite dans une même commande séparé d'un caractère spécial.

Et enfin, recevoir une confirmation de téléchargement de la part du client au serveur pour vérifier si le fichier a bien été envoyé sans perte.

## Conception de la commande Deposier

Pour cette partie on fait la conception du protocole de dépôt de fichiers images du client au serveur. Le schéma ci-dessous représente le schéma général de ce protocole.



## Réflexion

Pour déposer un fichier se trouvant sur la machine du client l'interface client demande au client d'entrer le nom du fichier qu'il souhaite déposer.

Puis, ce nom de fichier est envoyé au serveur pour effectuer une vérification de son type pour vérifier s'il est acceptable par le serveur ou incompatible.

Ensuite, dans le cas où le nom de fichier image est compatible, on envoie une confirmation au client pour effectuer la suite du transfert du contenu de cette image.

Le serveur va donc lire ce contenu et l'écrire dans un nouveau fichier qui portera le nom reçu précédemment.

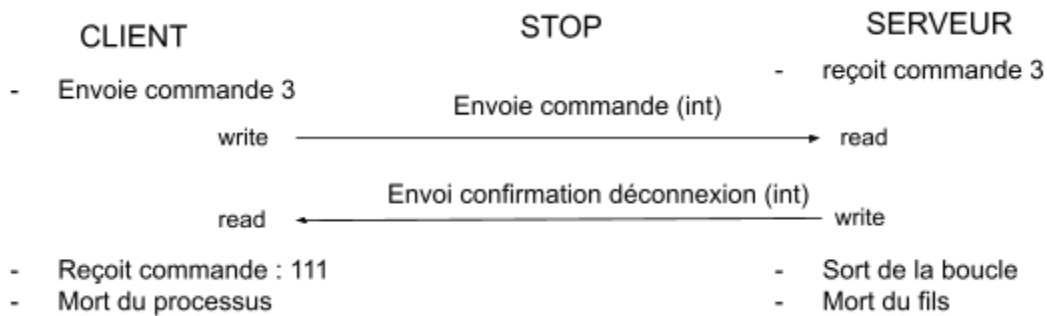
Lors de l'envoi du contenu du fichier, l'émetteur va d'abord envoyer la taille du fichier où l'on calcule le nombre de tampons pleins qu'il faudra lire et la taille précise du dernier tampon restant.

On a rencontré beaucoup de difficultés dans cette partie vu que la mémoire était saturée et donc le serveur ne lisait pas correctement le nom du fichier. Dans l'implémentation finale nous avons mis le nom du fichier en dur. Par conséquent nous n'avons pas de possibilité d'envoyer plusieurs fichiers ou d'autres fichiers que celui installé.

Nous aurions voulu ajouter la vérification du type de fichier entrée par le client. Nous aurions utilisé les commandes bash pour vérifier le type du fichier (exec, grep, file -i) puis comparé avec les types de fichiers compatibles présents dans le fichier MineTypes du côté serveur.

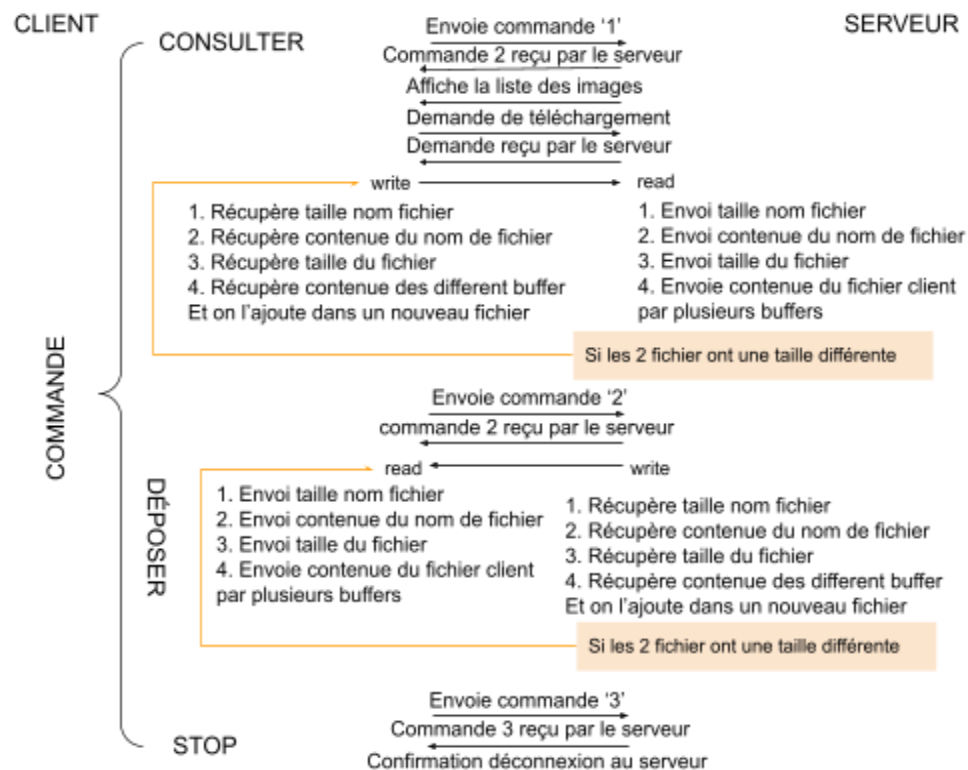
## Conception de la déconnexion

Pour cette partie on fait la conception du protocole de déconnexion qui s'effectue dans le schéma général.



Il s'agit ici d'un simple envoi de message de déconnexion au serveur qui à son tour envoie une confirmation au client pour lui permettre de bien se déconnecter en tuant son processus au bon moment.

## Résumé du schéma global des protocoles d'échanges



## PHASE 3

### **Implémentation de la consultation de la liste des fichiers disponibles :**

Pour consulter la liste des fichiers disponibles, on suppose que ces derniers se trouvent dans le répertoire Files. On initialise la primitive de répertoire et on ouvre le répertoire souhaité. Ensuite on transmet le contenu de ce répertoire dans le socket.

### **Implémentation du transfert de fichier :**

On commence par récupérer le nom d'un fichier ainsi que la taille du nom qu'on envoie au client avec la fonction "write" et le client reçoit avec la commande "read".

Suite à cela, on récupère tout d'abord la taille du fichier.

Avec cette taille, on effectue la division entière de la taille de nos buffers (n), et le reste (x) de cette division entière. Le résultat de cette division est le nombre de paquets à envoyer. Ensuite on cherche à trouver la taille précise du dernier paquet.

Et on envoie "n" buffers + "x" buffer au client qui reçoit ces buffers et les ajoute dans un nouveau fichier du même nom.

### **Implémentation de vérification de type :**

On crée le fichier serveur qui va implémenter notre serveur pour ça on crée notre socket d'écoute et on assigne les attributs de la structure socket\_adresse.

On attache la socket aux informations de la structure grâce à bind.

On ouvre le service et le serveur est sous écoute.

### **Puis dans une boucle infinie il accepte une connexion avec le client si elle est acceptée :**

La gestion du dialogue avec un client est déléguée au fils afin que le serveur puisse établir d'autres connexions avec d'autres clients en parallèle.

Lors du dialogue le serveur reçoit un message du client et le lit puis le serveur envoie une réponse qui correspond à une chaîne de caractère qu'on entre en entrée standard sur le serveur.

## PHASE 4 et 5

Dans cette partie, on ajoute les vérifications des arguments entrées dans les commandes bash pour exécuter les exec serveur et client.

Dans le serveur, il suffit d'ajouter en argument le port utilisé.

Dans le client, il suffit d'entrer à la suite le nom de la machine serveur, ainsi que le port utilisé par le serveur.

Pour obtenir le nom de la machine, nous effectuons un gethostbyname, permettant d'obtenir automatiquement l'ip de notre host qui est le serveur, et ainsi pouvoir connecter correctement la socket envoie.

Pour finir, nous avons ajouté un makefile, permettant de compiler correctement les fichiers.

## Difficultés

- Lors du dépôt du client au serveur et du téléchargement de fichier, nous avons eu un souci sur le transfert de contenu de ce fichier entre le client et le serveur.  
Les buffers correspondent bien à ce qu'on lit du fichier image cependant lors de l'écriture du fichier, pour transmettre aux serveurs les buffers. Il y a des caractères spéciaux qui s'ajoutent et l'image se déforme à cause de l'ajout de données malgré qu'on ait réinitialisé nos buffers avec "bzero".
- Il reste le souci du transfert des fichiers : nous n'avons pas géré le dépôt de plusieurs fichiers à la fois. Pour chaque fichier reçu notre application crée un fichier "reçu.ppm". Donc il est impossible de réécrire le contenu de ce fichier.
- Dans l'implémentation de l'application nous utilisons des pointeurs pour déclarer des chaînes de caractères. Ce qui peut provoquer une saturation de la mémoire. En effet, en testant l'application plusieurs fois, au bout d'un moment on ne recevait plus le résultat attendu.

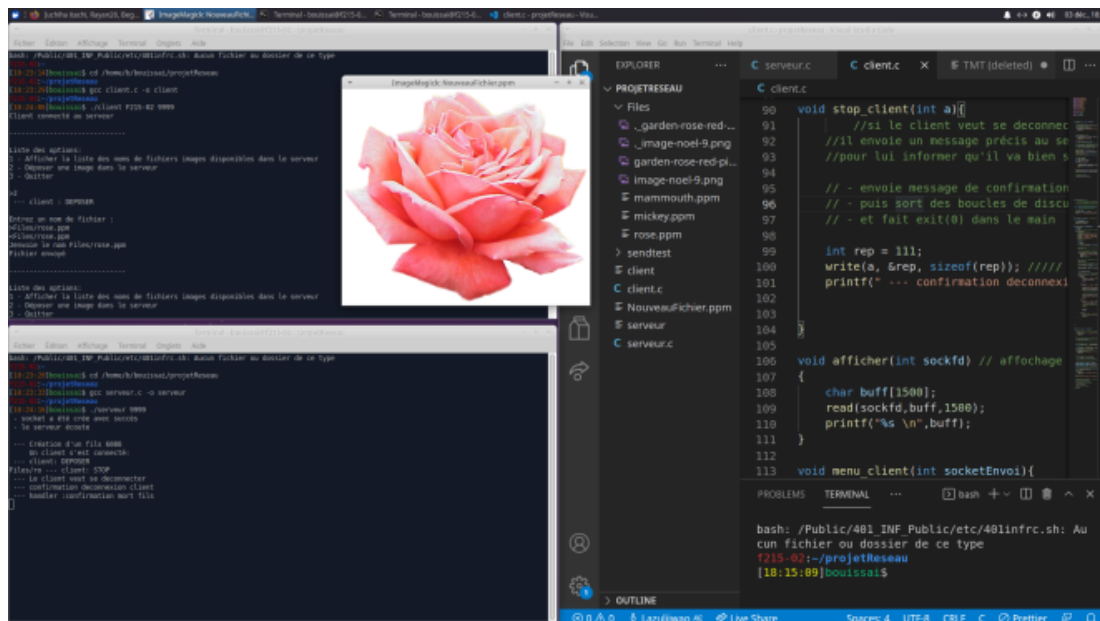
## Conclusion

L'objectif principal de ce projet était de fusionner et appliquer nos connaissances en systèmes et en réseaux. Nous avons eu pour but de mettre en place la communication entre deux machines en utilisant les primitive "sockets" qui utilisent le protocole TCP/IP.

Le produit final de notre travail exigeait d'agir en méthode agile en suivant soigneusement les consignes de chaque étape. Pour travailler en équipe nous avons suivi le prototype de notre application conçue au préalable dans la phase II à l'aide de schémas et dessins.

Nous avons ainsi pu utiliser les notions acquises en système telles que la gestion d'exécution de plusieurs processus en simultané.

## Exemple d'exécution



```
bash: /Public/401 INF_Public/etc/401infrc.sh: Au
cun fichier ou dossier de ce type
1219-02:~/projetReseau
[18:15:09]bouissai$
```