INF3405

Réseaux Informatiques

TP2 : Projet en réseaux informatiques
Traitement d'images par réseau

Remis par:

Matricule	Prénom & Nom
1728325	Soukaina Moussaoui
1828776	Tamer Arar

Soumis à :

Kadi, Mehdi
Itani, Bilal

École Polytechnique De Montréal

22 février 2018

Introduction

Dans le cadre de ce laboratoire, notre mandat était de créer une solution pouvant faire communiquer plusieurs clients avec un seul serveur. Autrement dit, nous devions être capable de créer un programme dans un environnement *multithread*. Les clients devaient pouvoir d'abord se connecter au serveur en entrant l'adresse IP de ce dernier et le port d'écoute et envoyer une image à ce dernier à partir d'un répertoire en utilisant un *socket*. Ce dernier pour sa part, devait être capable de traiter l'image pour y appliquer un filtre en noir et blanc (filtre *sobel*) et la renvoyer au client. Finalement, le client devait lire l'image convertie et l'enregistrer sur le disque dur. En l'occurrence, l'objectif principal de ce laboratoire était d'approfondir nos connaissances en matière de communication instantanée entre plusieurs clients avec un seul serveur en utilisant des *sockets* et des *threads*.

Présentation

Nous avons créé un espace de travail nommé *client* et un autre nommé *serveur.*

Côté serveur :

Du côté serveur, nous avons créé quatre fichiers sources : *Server.java, FiltreThread.java, Authentification.java* et *Utilisateur.java.* Le fichier *Server.java* contient les méthodes de vérification d'entrée d'adresse IP et port et la classe *main()* qui permet la connexion du serveur à chaque client connecté à travers le *socket.* Par ailleurs, le fichier *FiltreThread.java* contient l'implémentation d'une classe de type *Thread*. Cette classe est appelée à partir du *main()* du fichier *Server.java* à chaque fois qu'un client se connecte. Elle permet l'application du filtre sur l'image reçu par le client par un mécanisme d'authentification d'un utilisateur. Elle permet alors la création d'un environnement *multithread.* En outre, les fichiers *Authentification.java* et *Utilisateur.java* contiennent des méthodes de gestion d'utilisateurs et mots de passe.

Concernant les fonctionnalités du point de vue serveur, nous devions dans un premier temps pouvoir saisir les paramètres du serveur ; c'est-à-dire : l'adresse IP et le port. Ainsi, la fonction *Server.obtenirAdresseIPDuServeurACreer* permet de demander à l'usager d'entrer l'adresse IP et de vérifier si celle-ci est sur 4 octets en utilisant la méthode *Server.adresseIPEntreParUtilisateurEstValide*.

Pour le port, la méthode *Serveur.obtenirPortDuServeurACreer* permet à l'usager d'entrer le numéro de port et de vérifier si celui-ci est entre 5000 et 5050, tel qu'il nous a été requis. Deuxièmement, le serveur devait tenir une base de données qui contiendrait les noms d'usagers et les mots de passe. Nous avons ainsi créé un fichier .CSV nommé *data.csv* qui contient ces informations. Ainsi, quand le client entre un identifiant et un mot de passe, dans

Authentification.java nous effectuons la vérification de ces informations avec la méthode *Authentification.rechercheUtilisateur* et en utilisant *data.csv.* Ce fichier est mis à jour dans le cas d'un nouvel usager. Cette mise à jour est gérée par la méthode *Authentification.creerUnNouveldentifiantEtMotDePasseDansBaseDeDonnee.*

Par rapport au traitement d'image, le serveur devait être capable de recevoir une image du client, y appliquer le filtre *Sobel* et la renvoyer convertie au client. Autrement dit, quand le client envoie une image à travers le *stream*, dans *FiltreThread.java* on lit l'image et sa taille. Quand nous validons l'identifiant du client et son mot de passe, nous appliquons la fonction de conversion d'image en noir et blanc grâce à l'appel de méthode de *Sobel.process* et nous écrivons la taille de l'image et l'image transformée, en forme de tableau d'octets, dans le *stream* de sortie. Ainsi, le client sera capable de lire la taille et le tableau d'octets à travers ce *stream* et enregistrer l'image désirée dans son répertoire.

Côté client :

Le fichier source utilisé du côté client est le *Client.java.*

En premier lieu, avec la méthode *Client.connexionAuServeur,* le client est capable d'entrer les paramètres du serveur tels que l'adresse IP et le port utilisé et vérifier si la connexion est valide. Le client doit tenter autant de fois possibles la connexion au serveur jusqu'à que cette dernière soit valide.

Par ailleurs, quand la connexion est validée, le client est capable d'entrer son identifiant et son mot de passe qui sont envoyés et vérifiés dans le serveur, ainsi que le nom de l'image source en format *.JPG* (ex. *lassonde.jpg*). Nous vérifions si l'image est disponible sur le répertoire à l'aide de la méthode *Client.imageTrouve*.

Ensuite, lors de la validation de tous les paramètres entrés par l'utilisateur, nous procédons à l'envoi de l'image source au serveur pour sa conversion. L'image et sa taille sont envoyées en forme de tableau d'octets à travers le *stream* de sortie. Quand l'envoi s'effectue avec succès, un message dans la console est affiché.

Finalement, quand le traitement de l'image est effectué du côté du serveur et que celui-ci envoie l'image convertie à travers le *stream*, du côté client nous lisons en premier lieu la taille de l'image convertie, ensuite le tableau d'octets qui constitue l'image et finalement nous convertissons ce dernier en *BufferedImage*. Nous procédons en dernier lieu à la vérification de l'image : si celle-ci est vide, nous affichons un message d'erreur. Sinon, nous l'enregistrons dans le répertoire et nous affichons sur la console son chemin.

Exécution du programme :

Afin de lancer les deux exécutables, il est nécessaire de le faire dans des terminaux de commandes avec les commandes *java –jar serveur.jar* et java -jar client.jar.

Difficultés rencontrées

Lors de la réalisation du travail pratique, nous avons rencontré plusieurs types de problèmes. Premièrement, sur un niveau d'organisation, il y a eu une répartition de tâches. Une personne s'occupait du client, et une autre du serveur. Des commentaires devaient aussi être ajoutés pour faciliter la compréhension du code qui grossissait de jour en jour.

Deuxièmement, du point de vue technique, la difficulté majeure était d'envoyer l'image. On ne pouvait pas simplement envoyer le fichier. Il fallait aussi indiquer sa taille pour permettre à son destinateur de savoir quand arrêter de lire le fichier envoyé, car, sinon, il le lisait à l'infini. Ainsi, l'expéditeur doit spécifier dans le *outputStream*, en plus de l'image, un tableau de octets contenant la taille de l'image. Aussi, parfois, l'image ne se recevait pas complètement par le client après l'application du filtre *Sobel*. Un *readFully()* à la place d'un *read()* sur le *inputStream,* lors de la réception, a résolue la situation.

Une autre difficulté était celle de l'authentification de l'usager sur le serveur. En effet, on envoyait d'abord l'identifiant et le mot de passe au serveur pour s'inscrire ou s'authentifier, mais le serveur n'arrivait pas à retourner plusieurs requêtes (des *flush*). On a donc opté pour une solution qui invoque qu'un seul envoi de la part du client et du serveur. Ainsi, on envoie l'identifiant, mot de passe et image à filtrer en même temps, plutôt qu'à tour de rôle.

Finalement, une troisième difficulté était la gestion des entrées des utilisateurs. Par exemple, une adresse IP doit être de format *x.x.x.x* où les x représentent des nombres sur un octet (0-255) et un port doit être un nombre entre 5000 et 5050.

Critiques et Améliorations

Concernant l'organisation de ce laboratoire, nous considérons que les instructions sont bien définies et décrites en détails. Aussi, les chargés de laboratoire nous ont bien guidés durant les séances. Nous espérons que les prochains laboratoires se dérouleront de la même manière.

Conclusion

Ce laboratoire nous a permis d'utiliser le concept de *sockets* et de *threads* vus dans les cours précédents tout en intégrant de nouveaux concepts liés à la réseautique tels que l'adressage IP et l'utilisation de ports. Nous avons pu remarquer l'importance de l'utilisation des *sockets* afin de permettre la communication instantanée entre les entités ainsi que l'importance de l'usage de *threads* afin de permettre la communication de plusieurs clients avec un seul serveur. De plus, nous avons pu revoir l'importance des principes de la programmation orientée-objet pour la réduction de couplage dans le code. En résumé, ce laboratoire nous a permis d'appliquer les concepts de bases du développement, de communication *multithreading* en utilisant des *sockets* tout en ayant un avant-goût de nouveaux concepts en réseautique.