

Jérémy Barrette – 1736976

Alexis Vailles – 1742139

**Rapport TP #1 :**

**Projet en réseaux informatiques – Gestionnaire de fichiers**

Soumis à : Kadi, Mehdi

INF3405 (01 – B1) – Réseaux informatiques

Session Automne 2018

École Polytechnique de Montréal

Vendredi le 26 octobre 2018

# **Introduction**

De nos jours, le besoin d’espace de stockage numérique est de moins en moins un problème. Désormais, de nombreuses plateformes de stockage en ligne, comme Dropbox ou Google Drive, permettent de sauvegarder nos fichiers informatiques sur le web et d’y avoir accès de partout. Physiquement, ces fichiers sont toutefois stockés sur un serveur externe, auquel nos ordinateurs ont accès via une connexion.

C’est ce type de technologie que nous fait explorer le premier travail pratique (TP) du cours *INF3405 : Réseaux informatiques*. L’objectif de ce laboratoire est de programmer nous-mêmes un stockage sur un serveur externe, en programmant à la fois les fonctions nécessaires pour le compte client et le compte serveur, en ne conservant pour l’instant que l’interface console.

# **Présentation des travaux**

La première chose que nous avons faite a été d’implémenter correctement la communication entre le serveur et le client, conformément aux exigences dans l’énoncé du laboratoire. Il s’agissait, essentiellement, de s’assurer que l’adresse IP entrée était dans le bon format (respectant le nombre de bits, uniquement des nombres positifs et aucun caractère autre que des chiffres), et que le port soit compris entre 5000 et 5050. Nous avons donc imposé une série de conditions pour que le code respecte ces conditions, et qu’il rejette toute adresse entrée en dehors des plages permises.

Il était ensuite demandé d’implémenter 6 commandes : les fonctions **cd**, **ls**, **mkdir**, **upload**, **download**, et **exit**.

Pour coder la commande cd, qui permet de se déplacer vers un répertoire parent ou enfant, nous avons utilisé une variable de type Path pour identifier l’emplacement absolu où se trouve le serveur et donc les fichiers et dossiers qui se trouvent autour de lui. On utilise ensuite l’argument passer à la commande afin d’essayer de continuer dans un sous-dossier s’il existe ou de retirer le dernier dossier du Path actuel pour remonter à un dossier mère. On retire le dernier dossier en identifiant la position du dernier « \ » dans le format *String* du *Path* et en le retirant avec tout ce qui se trouve après.

La commande ls, qui sert à afficher les dossiers et les fichiers dans le répertoire courant, a été implémentée avec un tableau dans lequel les positions des répertoires étaient regroupées grâce à la fonction *listFiles* de la variable *File*. Nous sommes ensuite passé au travers de la liste en vérifiant la nature du *File*, soit un fichier ou un dossier, pour les afficher regroupés entre eux. Nous envoyons chaque ligne individuelle au client avec une dernière ligne contenant *#End* afin de l’informer que l’envoie est terminé. Cela nous permet d’envoyer un nombre varaible d’information facilement.

La commande mkdir a pour but de créer un dossier dans le serveur à partir du client. Pour l’implémenter, nous n’avons eu qu’à utiliser la fonction *createDirectory*, en testant en premier lieu que le dossier n’existe pas déjà.

La commande upload vise à télécharger un fichier du client dans le serveur. Pour réaliser son implémentation, on commence par vérifier que le fichier à envoyer existait déjà dans le répertoire, puis ensuite on créé un Buffered Input stream du fichier sur l'envoyeur et un Buffered output stream pour le receveur afin de lire et écrire le fichier. On envoie ensuite le contenu du fichier dans un array (tableau) d’octets par le socket entre deux streams de données.

La commande download a pour but de télécharger un fichier du serveur dans l’ordinateur du client. Son implémentation est sensiblement la même que pour la commande upload, mais avec le transfert fait de manière inversée. Une modification supplémentaire est que le client attend une première réponse du serveur afin de confirmer que le fichier existe avant de préparer les Streams du transfert. Dans le Upload, le client vérifiait si le fichier était existant avant de contacter le serveur, ce qui évitait de la communication inutile.

Finalement, la commande exit permet au client de se déconnecter du serveur. Elle a été implémentée simplement en lisant la commande « Exit » entrée par l’utilisateur. Le message est envoyé au serveur pour que la boucle infinie d’écoute soit brisée, et un booléen local du client est mis à vrai afin d’informer à l’utilisateur qu’il est déconnecté s’il essaie de faire d’autres commandes, pour éviter d’éventuels problèmes dans l’appel de commandes sans connections serveur.

# **Difficultés rencontrées**

Plusieurs difficultés ont été rencontrées au cours de la complétion de ce TP. D’abord, nous avons remarqué que l’utilisation de la fonction *in.readLine()* s’arrêtait aux retours à la ligne et séparait donc un *String* sur plusieurs lignes en plusieurs lectures indépendantes, nous empêchant d’envoyer un unique large message pour la commande *ls*. Pour régler cela, nous avons utilisé le message de fin *#End* mentionné plus haut.

Ensuite, nous avons eu de la difficulté à bien lire les adresses et leur port, puisque deux caractères coupaient la lecture : le « **.** » et le « **:** ». Nous avons remédié à ce problème en créant un tableau string nommé *parsedAddress[]* pour bien séparer les valeurs de l’adresse IP et du port afin de confirmer les limites imposées à chaque valeur et leur nombre et position dans l’adresse.

Finalement, nous avons eu de la difficulté les transfert de fichier des fonctions upload et download. En effet, en faisant nos recherches nous avons trouvés plusieurs implémentation du transfert de fichier et avons du passer par quelques itérations avant de trouver un mode fonctionnel. De plus, lors de leurs tests, il était difficile de déterminé la cause d’un blocage de la commande, étant donné que le client tombait en attente infinie si le nombre de message n’était pas équilibré entre le serveur et le client. Il a fallut utiliser de nombreux messages console afin de confirmer à quels niveau de leur déroulement étaient les deux programmes.

# **Critiques et améliorations**

À la lumière de ce qui a été accompli au cours de ce laboratoire, nous croyons que quelques améliorations pourraient être apportées.

D’abord, pour faciliter la compréhension des étudiants, ce serait une bonne idée de séparer ce laboratoire en deux sous-laboratoires : un visant à comprendre et se familiariser avec les *sockets*, et le second visant à se concentrer plus efficacement à l’implémentation des tâches du serveur et du client.

Pour contourner ce problème du manque de connaissances par rapport aux sockets, il aurait aussi été possible de fournir un exemple de transfert de données par socket (autre qu’avec les fonctions *println* et *read*), pour aborder la tâche complexe du fichier avec une base plutôt que les sources internet comme principal support.

# **Conclusion**

En conclusion, nous pouvons affirmer que ce laboratoire nous a été fort utile. Il nous a permis, entre autres, de nous initier à la communication entre serveur-client par socket, un élément de base dans le cadre du cours sur les réseaux informatiques. Ces connaissances risquent d’être fort utiles dans nos futures carrières d’ingénieurs, et constituent une excellente base de connaissances pour la suite du cours et du programme.