INF3405 Réseaux informatiques

H2021

Travail pratique 1 : Gestionnaire de fichier

Groupe 04

Soumis à :

Itani, Bilal

Par :

Gargouri, Karim (1946439)

Safaa, Samia (1981631)

Tjiu-Gildart, Madeleine (1690615)

Département de génie informatique et génie logiciel

Polytechnique Montréal

26-02-2021

INTRODUCTION

Une application client-serveur a été construite dans le but de créer un lieu de stockage de données indépendant des comptes infonuagiques qui dominent présentement le marché. Cette application a été écrite en Java et contient deux projets : *Client.java* et *Serveur.java* qui communiquent de manière bidirectionnelle par moyen de sockets. Des sockets sont des points de communication virtuelle situées entre les couches application et transport du réseau. Les sockets sont non persistants et, dans le présent travail, sont orientés-connexion TCP. De plus, le serveur permet la connexion simultanée de plusieurs clients grâce à des threads. Les deux applications sont basées sur les codes exemplaires fournis dans les notes de cours du module 1 [1].

PRÉSENTATION DES TRAVAUX

Deux projets Java ont été créés afin d'exécuter l'application client-serveur, soient les projets *clients* et *serveur*. Du côté du client, le projet demande d'abord à l'utilisateur d'entrer au clavier l'adresse IP et puis le port du serveur auquel il voudrait se connecter. Un objet de classe *Scanner* garde ces entrées respectivement dans les variables *serverAddress* et *port*. Dans la fonction *main*, la fonction *ipValide* vérifie que l'adresse est composée de quatre octets avec des valeurs allant de 0 à 255, chacun séparé par un point de ponctuation, mais ne se terminant pas par un point. La fonction *readPort* vérifie que le numéro de port entré est un entier faisant partie de l'intervalle [5000, 5050] et affiche un message d'erreur si la valeur entrée par le client ne répond pas à ces critères. À la suite de la validation de l'adresse IP et du port, l'application crée un socket avec ces deux valeurs comme paramètres. Ce socket établie une connexion avec un socket du côté serveur ayant les même adresse IP et numéro de port. Un canal d'entrée de type *DataInputStream* et un canal de sortie de type *PrintStream* sont ensuite initialisés. Ces canaux permettent respectivement d'écrire et de lire au serveur. L'application affiche un message de bienvenue provenant du serveur et attend que l'utilisateur entre une commande au clavier. Cette commande est enregistrée dans la variable *strKb* et envoyée au serveur pour traitement.

Une fonction switch est employé pour décerner deux commandes clients qui mettent en jeux l'envoie de paquets : *upload* et *download*. Le fonctionnement des fonctions *sendFile* et *saveFile* a été inspiré par les fonctions de même nom trouvés dans les applications Java de l'utilisateur Carl Ekerot sur GitHub [2]. Lorsque l'utilisateur désire télécharger un fichier, la fonction *sendFile* du *ClientFileManager* est employé. *ClientFileManager* est une extension de la classe *File* et a été rajoutée au *classpath* du projet *Client.java*. La fonction *sendFile* prend comme arguments le nom du fichier à envoyer en variable *String* et un objet de type *DataOutputStream*. Un objet *FileInputStream* sert de fichier temporaire se comportant comme tube d'envoi de fichier. Pendant que cet objet est en train d'être lu, son contenu est enregistré dans un tableau *buffer* de type *byte* et ses contenus sont écrits à l'objet *DataOutputStream* lorsque la taille maximale de *buffer* est atteinte. On continue à écrire du *FileInputStream* au *buffer* et ensuite du *buffer* au *DataOutputStream* jusqu'à ce que le fichier temporaire soit lu en entier. Lorsque le client désire télécharger un fichier à partir du serveur, l'application client se sert de la fonction *saveFile*, contenu elle aussi dans le *ClientFileManager*. Cette fonction prend comme arguments objet *DataInputStream* et un *String* contenant le nom du fichier. Le nom d'accès du

fichier est obtenu par moyen d'un objet *FileOutputStream* et cet objet sert de tube d'envoi de fichier temporaire. Un entier *int read* est initialisé à zéro et prendra la longueur de la table *buffer*, qui reçoit des parties des données à télécharger. Tant que la lecture du *DataInputStream* n'est pas finie, le *FileOutputStream* écrit le contenu du fichier téléversé. La fonction *readMessagesFromServeur* prend comme paramètre l'objet *DataInputStream* et affiche la réponse du serveur à la demande du client. Finalement, lorsque l'utilisateur soumet la commande *exit*, la connexion entre les sockets est rompue. Les ressources *Socket*, *PrintStream*, et les *Scanner* sont fermées par la suite.

L'application serveur crée un socket nommé *listener* de type *ServerSocket* qui pourra se lier avec tout socket client qui connait son adresse IP et son numéro de port. La méthode *bind()* lie le serveur à l'adresse et le port désignés pour cet exemple, soient 127.0.0.1 et 5000 respectivement. Ensuite, le serveur est à l'écoute de connexions provenant d'applications clients. Lorsqu'une requête de connexion est perçue, le serveur fait appel à la fonction *ClientHandler* afin de gérer plusieurs interactions de clients simultanées. *ClientHandler* initie une connexion avec la méthode *accept* et incrémente le numéro du client *clientNumber* avec chaque nouvelle connexion client. Un message de bienvenue est envoyé au client et la fonction est en mode d'attente jusqu'à ce que le client écrive au serveur. Un objet *Scanner* est utilisé pour lire les commandes écrites par le client et un objet *DataOutputStream* écrit des messages au client. Une boucle *while* vérifie l'existence d'un message non-nul du client et enregistre cette entrée dans un variable *String strClient*. Un objet *FileManager* est initialisé pour que le répertoire d'origine au moment de la connexion soit le répertoire nommé « Stockage ».

Le serveur est en mesure de reconnaître sept commandes et de leur associer l'action appropriée. Pour certaines commandes, seul le nom de la commande est requis. Pour d'autres, le nom d'un fichier ou d'un répertoire doit suivre le nom de la commande. Dans tous les cas, l'entrée du client est traitée pour séparer les mots selon les espaces blancs. Les segments individuels sont enregistrés dans un tableau de type *String* nommé « command ». Une opération *switch* traite la première entrée de la table command, qui devrait représenter la commande à effectuer. La commande *ls* fait référence à l'objet *FileManager dir* et demande que, pour chaque élément faisant partie du répertoire du serveur, que le nom de ce dossier ou fichier soit écrit au client. Pour que le client puisse se déplacer entre répertoires parent et enfant, le cas de la commande *cd* nécessite que le client ait aussi fourni le nom d'un répertoire vers lequel se déplacer. L'objet *FileManager* prend la valeur du répertoire de destination à fur qu'on se déplace entre répertoires enfant et parent. Le serveur envoie un message au client pour lui confirmer dans quel répertoire il se trouve. La fonction *mkdir* nécessite aussi que le client ait fourni le nom d'un dossier *path* comme information supplémentaire. Cette commande permet de créer un nouveau dossier dans le répertoire courant. Une méthode *mkdir* du *FileManager* crée un nom d'accès abstrait de type objet *File*. Si le dossier peut être créé, le serveur écrit un message au client comme quoi il a été créé. Sinon, le serveur envoie un message demandant à l'utilisateur de réexécuter la commande avec un nouveau nom de dossier. Lorsque l'utilisateur désire téléverser un fichier au serveur avec la commande *upload*, ce dernier emploi la fonction *saveFile* du *FileManager* pour y arriver. Dans le cas où l'utilisateur demande de télécharger un fichier avec la commande *download*, le serveur utilise la fonction *sendFile* du *FileManager.* Le

fonctionnement de ces deux méthodes est tel que décrit dans la section précédente concernant l'application client. Analogue à *ClientFileManager*, *FileManager* est une extension de la classe *File* qui a été rajoutée au classpath du projet *Server.java*. Une fonction additionnelle a été rajoutée pour supprimer un dossier ou un fichier lorsque l'utilisateur envoi *remove* comme commande. Afin de gérer les erreurs occasionnées lorsque l'utilisateur entre un nom de dossier ou de fichier invalide, la fonction *contains* du *FileManager* est employée pour vérifier si l'élément existe. Lorsque la commande *exit* est reçue, le serveur confirme au client qu'il a été déconnecté avec succès et le programme sort de la boucle *while* qui attend une directive du client. Le socket est fermé et le console serveur affiche un message comme quoi la connexion avec le client en question a été fermée. Enfin, si le client entre une commande qui ne répond à aucun de ces cas, le serveur lui envoie un message disant que la commande n'a pas été reconnue et il attend une nouvelle entrée.

DIFFICULTÉS RENCONTRÉES

Une première difficulté a été le traitement des entrées par le client. Il a fallu faire des recherches sur les objets *Scanner*, *DataInputStream*, et *BufferedReader* pour mieux saisir comment lire une ligne entière comportant des espaces blancs. La solution d'un *scanner* a été retenue pour lire les lignes de commandes provenant de l'utilisateur avec sa méthode *readLine*. Une deuxième difficulté a été le traitement de la commande *cd ..* pour passer d'un répertoire enfant vers le parent. Il a fallu faire des recherches sur différentes expressions régulières qui permettrait de traiter le type de séparateur utilisé par le système d'exploitation que l'application roulait dessus. Il a fallu aussi apprendre plus sur le *CanonicalPath* par rapport au *AbsolutePath* des chemins d'accès des répertoires pour résoudre le traitement de la commande *cd ..* .

CRITIQUES ET AMÉLIORATIONS

Tandis qu'un cours de programmation de base fait partie du tronc des programmes d'ingénierie à Polytechnique, une séance d'introduction à Java pourrait grandement bénéficier les étudiants qui n'étudient pas en génie informatique ou logiciel. Cette séance pourrait être fait en synchrone, par exemple le premier cours de laboratoire est consacré à des exercices avec le chargé de laboratoire, ou pourrait se faire de manière asynchrone avec des exercices guidés déposés disponibles sur Moodle.

CONCLUSION

En conclusion, l'objectif de créer une application serveur-client qui permettait une communication bidirectionnelle a été atteinte. D'abord, ce laboratoire a concrétisé la compréhension du fonctionnement des sockets orienté-connexion TCP. En créant un thread pour permettre la connexion de plusieurs clients, il a été possible de simuler avec plus de justesse le comportement d'un serveur face à de nombreux requêtes. De plus, il a été possible de comprendre le transfert de paquets de données avec les commandes *upload* et *download* du l'utilisateur. Enfin, les étudiants du cours pourraient bénéficier d'une session d'introduction à Java dans le cadre du cours INF3405 afin de mettre à niveau les étudiants à l'extérieur des programmes de génies informatiques et logiciels.

RÉFERENCES

[1] A. Quitero, *INF3405 Réseaux informatiques Module 1 : Introduction,* Moodle@Polytechnique Montréal.

[2] C. Ekerot, "FileClient.java," GitHub, [Online]. Available: https://gist.github.com/CarlEkerot/2693246. [Accessed 24 Feb 2021].