

Département de génie informatique et génie logiciel

INF3405

Réseaux informatiques

**TP1 : Projet en réseaux informatiques**

**Gestionnaire de fichier**

Soumis par

Antoine Chaffin ()

Yann Trottier (1745796)

Groupe : 01

Soumis à Mme. Émilie D-P.

Le 28 février 2019

# Introduction

Les services de stockage, tels que Dropbox ou Google Drive, en dépit de leur utilisation populaire, sont construits sur une fondation simple. En effet, il est possible de concevoir un service de stockage élémentaire chez soi sans trop de difficultés.

Dans le cadre de ce laboratoire, l’objet était de concevoir une application client-serveur en Java, similaire à Dropbox. Cette application permet à un utilisateur de stocker tout fichier sur un serveur de stockage à l’aide d’une interface graphique sur la console. De plus, suivant le contexte, le serveur doit pouvoir authentifier, servir plusieurs utilisateurs simultanément et attribuer à chacun leur propre espace de stockage.

Sur le plan théorique, l’application emploie le protocole de transport TCP. En effet, l’application requiert une conversation entre le client et le serveur, ce qui est difficile à gérer avec UDP. Le serveur doit pouvoir ouvrir, à l’aide de la méthode *accept()*, une session unique pour chaque client, effectuant un « handshake » initial TCP. Il doit authentifier chaque client afin de leur attribuer l’espace de stockage existant, et une fois authentifié, maintenir cette session en vie, ce qui n’est possible qu’avec TCP. Lors des transferts de messages et de fichiers, on assume aussi que les messages sont transférés en totalité et séquentiellement, ce qui est offert comme fonctionnalité de base de TCP.

Comme objectif de laboratoire, l’application doit aussi faire usage des sockets. Les sockets sont une abstraction d’un point de communication, identifiés par un couple Adresse/Port, et sont utilisés au niveau de la couche d’application comme interface pour communiquer avec la couche de transport. Dans l’application, le serveur crée un socket connu comme point d’écoute, dont le port est entre 5000 et 5050, et le client se connecte à ce socket à l’aide d’un socket éphémère. La session TCP est alors établie et identifiée par le couple de sockets client-serveur.

Un autre objectif est l’emploie des threads. Puisque le serveur doit ouvrir plusieurs connexions simultanément, l’application lance un thread pour chaque connexion unique.

Puis, ce laboratoire vise aussi à évaluer deux des douze qualités de l’ingénieur définies par le BCAPG, soit la conception et la communication.

# Présentation des travaux

Note application se sépare en deux projets Java, le client et le serveur.

## Projet serveur

L’application serveur est divisée en 5 classes. Les deux classes principales sont *Main* et *Service*. Trois autres classes, *ServerDB*, *ServerUtils*, et *SharedUtils* supplémentent les deux classes principales à l’aide de méthodes statiques. *SharedUtils* est une classe partagée avec le projet client (voir section 3).

### Classe Main

Dans la classe *Main* on retrouve la méthode m*ain* qui roule sur le thread principal. Cette méthode demande au démarrage le port et l’adresse du serveur à utiliser à l’aide des méthodes *DemanderPortAUtiliser* et *DemanderAdresseIPAUtiliser* (voir section 3)*.* Ensuite, elle initialise la base de données avec *InitDB* (voir section 1.3). Elle crée ensuite un socket à l’aide de l’adresse et du port demandés et bloque en attente d’une connexion client. Lorsqu’une connexion client est acceptée, à l’aide de la méthode Java *ServerSocket.accept()* effectuant un handshake, on démarre un thread de type *Service* pour la nouvelle session avec son numéro unique.

### Classe Service

Ainsi, le fonctionnement principal du serveur se trouve dans la classe *Service*, qui est étend la classe *Thread*. La classe *Service* a un constructeur, qui enregistre le numéro de session et le socket client (adresse IP, port), et n’a qu’une seule méthode, *run()*.

1.2.1 Fonctionnement de la méthode run()

On veut d’abord authentifier le client. On bloque jusqu’à ce qu’on reçoive l’usager du côté client. Une fois reçue, on vérifie si l’usager possède déjà un mot de passe dans la base de données l’aide de la méthode statique *ObtenirPasswordUsager*. Si un mot de passe existe, le serveur répond au client à l’aide d’un code de confirmation « EXIST ». On attend alors une réponse du client contenant le mot de passe. Si celui-ci est valide, on envoie le code « VALIDE » et l’authentification termine, sinon on envoie « INVALIDE » et on boucle la procédure d’authentification jusqu’à ce que les 3 essais soient échoués.

Une fois l’authentification terminée, on s’attend perpétuellement, à l’aide d’une boucle, de la part de l’usager, l’une des 5 commandes possibles, « dc », « ls », « upload », « download », « delete ». Pour le cas « dc », on envoie le code « Deconnexion » confirmant la déconnexion, puis on ferme le socket du côté serveur. Pour le cas « ls », on boucle tous les fichiers retournés par la méthode Java *File.listFiles()*, les préfixe de l’identifiant « [Directory] » ou « [File] », et les envoie au client. Après tous les envois, on envoie un caractère vide au client pour signaler la fin de la liste. Pour le cas « upload », on s’attend ensuite à un nom de fichier. Une fois le nom reçu, on crée un objet Java *File*. On lit le contenu du fichier envoyé du client à l’aide de la méthode Java *ObjectInputStream.readObject(byte[])* dans un tableau, qu’on écrit dans l’objet *File* à l’aide de *Files.write()*. Pour le cas « download », on s’attend ensuite à un nom de fichier. Si le fichier n’existe pas, on envoie le code « DOESNTEXIST ». Si oui, on envoie « STARTINGTODOWNLOAD », on enregistre le contenu du fichier dans un tableau de bytes et on envoie ce tableau au client à l’aide de la méthode Java *ObjectOutputStream.writeObject(byte[]).* Pour le cas « delete », on s’attend ensuite à un nom de fichier. Si le fichier existe, on le supprime de l’espace de stockage de l’usager, et on envoie un message au client confirmant la suppression. Sinon, on envoie un message que le fichier n’existe pas. Pour tout autre cas, on écrit un message d’erreur. Si jamais lors de la lecture ou d’une écriture la connexion au client échoue, par exemple dans le cas où *readLine()* retourne *null* ou une exception *IOException* est lancée, on ferme la connexion.

### Classe ServerDB

La classe ServerDB contient toutes les méthodes statiques utilitaires liées à la base de données. On y retrouve d’abord la méthode *ObtenirPasswordUsager(String user),* qui vérifie si un mot de passe existe dans la base de données, soit un fichier texte « Database.txt ». Dans ce fichier texte, chaque usager et son mot de passe sont mis sur une même ligne, séparé d’un caractère « : ». À l’aide de la méthode Java *String.Split(":"), c*ette méthode retourne le mot de passe si celui-ci existe, et sinon retourne une chaîne vide. Ensuite, on a la méthode *enregisterNouvelUtilisateur(String user, String password)*, qui écrit l’usager et son mot de passe dans la base de données avec la méthode Java *Files.write()*. On a aussi la méthode *createUser(String user)*, qui crée un dossier avec le nom de l’usager dans l’espace de stockage avec la méthode Java *File.mkdir()*. Puis, on a la méthode I*nitDB()* qui crée la base de données « Database.txt » si elle n’existe pas déjà.

### Classe ServerUtils

Cette classe ne contient qu’une méthode statique utilitaire *AfficherCommandeRecue(String socket, String commande, String user)* permettant d’afficher les commandes reçues au niveau serveur avec le bon formattage.

## Projet client

Le projet client est divisé en 3 classes, *Client*, *ClientUtils* et *SharedUtils*. *SharedUtils* est une classe partagée (voir section 3). La classe *ClientUtils* contient une fonction utilitaire *CheckWhitespace* vérifiant la conformité d’un mot de passe. La seule méthode principale dans le projet client est la méthode *main* de la classe *Client*.

2.1 Fonctionnement de la méthode main()

On demande d’abord le port et l’adresse à utiliser à l’aide des méthodes *DemanderPortAUtiliser* et *DemanderAdresseIPAUtiliser* (voir section 3). À l’aide de la classe Java *Socket*, le client initie une connexion au serveur avec l’adresse et le port demandés. Le constructeur de *Socket* fait le handshake initial. Une fois connecté, on veut authentifier le client en lui demandant un nom d’usager, en s’assurant que le format du nom est admissible avec la méthode *ClientUtils.CheckWhitespace*. On envoie le nom au serveur. Si on reçoit « NEW », on permet à l’usager d’entrer un nouveau mot de passe, et on l’envoie au serveur. Sinon, on demande d’entrer un mot de passe, et on vérifie sa validité à l’aide du code « VALIDE » envoyé du serveur jusqu’à ce que les 3 essais soient échoués.

Une fois authentifié, on s’attend à ce que le client entre l’une des cinq commandes « dc », « ls », « upload », « download » ou « delete ». Dans le cas de « dc », on envoie le code « dc » au serveur et attend que le serveur confirme la déconnexion. Dans le cas de « ls », on envoie le code « ls » au serveur, puis on lit et imprime chaque ligne reçue jusqu’à ce qu’on lise le caractère vide. Pour le cas « upload », on utilise une expression régulière pour assurer la validité de la commande et obtenir le nom de fichier. On envoie le code « upload » au serveur, signifiant le cas « upload » au serveur, suivi du nom du fichier. On lit le fichier dans un tableau de bytes et l’envoie au serveur à l’aide de *ObjectOutputStream*.*writeObject.* Dans le cas de « download », on utilise aussi une expression régulière et obtient le nom de fichier. On envoie le code « download » au serveur, signifiant le cas « download » au serveur, suivi du nom du fichier. Si le code reçu est « STARTINGTODOWNLOAD », on lit le contenu du fichier à l’aide de *ObjectInputStream.readObject* et l’écrit sur disque à l’aide de *Files.write()*. Sinon, on envoie un message d’erreur. Pour le cas « delete », on envoie le nom du fichier au serveur obtenu à partir d’une expression régulière. Pour tout autre cas, on écrit un message d’erreur.

## Classe partagée *SharedUtils*

La classe *SharedUtils* est une classe partagée entre les deux projets. En effet, les requis du côté client et serveur sont exactement les mêmes pour l’obtention d’un port et d’une adresse IP. Cette classe contient les méthodes statiques utilitaires *DemanderPortAUtiliser* retournant un port valide et *DemanderAdresseIPAUtiliser* retournant une chaîne contenant l’adresse IP. *DemanderPortAUtiliser* emploie une boucle qui demande un port jusqu’à l’obtention d’un port valide. Elle utilise la méthode *Scanner.hasNextInt()* pour vérifier si le port entré est un entier et vérifie si celui-ci est entre 5000 et 5050. Sinon elle écrit un message d’erreur et redémarre la procédure. *DemanderAdresseIPAUtiliser* emploie aussi une boucle qui demande une adresse IP jusqu’à l’obtention d’une adresse valide. Elle vérifie d’abord si la chaîne entrée ne contient que des nombres et le caractère « . » à l’aide d’une expression régulière. Si oui, elle vérifie s’il y a bien quatre parties à l’adresse à l’aide de la méthode Java *String.Split("\\.").* Si oui, elle vérifie que chaque partie est un entier entre 0 et 255 inclusivement. Dans tout cas d’erreur, elle écrit un message d’erreur à la console et redémarre la procédure.

# Difficultés rencontrées, critiques et améliorations

La difficulté principale de ce laboratoire était de synchroniser le comportement du client et du serveur. Une autre difficulté était d’allouer un espace de stockage pour chaque utilisateur.

# Conclusion

En conclusion, l’objectif de ce laboratoire était de concevoir une application client-serveur, soit un service de stockage pouvant servir plusieurs utilisateurs, utilisant des sockets et des threads. L’application emploie le protocole TCP. Les objectifs de ce laboratoire ont été atteints.