Université Lille 1 *IEEA*

Rapport de Travaux Pratiques

Algorithmes et Applications (AeA)
TP2 : Minimum Spanning Trees (MST)

Auteurs :

Leo PERARD
Salla DIAGNE

Professeur :

Bilel DERBEL

XX-XX-XXXX

Introduction

Ce TP est une implémentation en JAVA de deux algorithmes de recherche d'un arbre couvrant minimal d'un graphe, Prim et Kruskal. Ce TP permet aussi de tester la performance des deux algorithmes sur des graphes générés aléatoirement selon le modèle d'Erdos-Renyi.

Listing des dossiers et fichiers du projet

doc/ : javadoc du projet

lib/ : contient les librairies (`.jar`) dont le projet est dépendant

src/ : contient les fichiers sources (`.java`) du projet

test/ : contient les fichiers de tests

uml/ : contient les diagrammes de classes du projet

build.xml : fichier de gestion du projet (Ant)

mstFinder.jar : archive exécutable du projet

Utilisation

Pour lancer le serveur, il suffit d'ouvrir un terminal et taper les commandes :

- `ant package`
- `java -jar ftpServer.jar [port] [baseDirectory]` avec :

  port : le numéro du port TCP sur lequel écoutera le serveur

  baseDirectory : le répertoire auquel auront accès les clients potentiels

Architecture

Ce projet est composé de 33 classes répartis en 3 paquetages :

- *package ftp*
  - *package client*
    `FTPRequestHandler` : classe représentant un handler de requête FTP. Une

instance de cette classe sera lancée dans un nouveau Thread à chaque fois qu'un client est connecté.

- *package server*

`FTPServer` : classe représentant le serveur du projet. Il se chargera de lancer un processus en parallèle quand un client sera connecté.

- *package command*

`FTPCdupCommand` : classe représentant la commande CDUP.

`FTPCommand` : interface représentant une commande (polymorphisme).

`FTPCommandManager` : classe représentant le manager des commandes. C'est cette classe qui se chargera d'exécuter la commande adéquate quand le serveur en reçoit une.

`FTPBasicCommand` : classe abstraite représentant une commande valide.

`FTPConnectionNeededCommand` : classe abstraite représentant une commande nécessitant que l'utilisateur soit connecté.

`FTPCwdCommand` : classe représentant la commande CWD.

`FTPDeleCommand` : classe représentant la commande DELE.

`FTPEprtCommand` : classe représentant la commande EPRT.

`FTPEpsvCommand` : classe représentant la commande EPSV.

`FTPListCommand` : classe représentant la commande LIST.

`FTPNlstCommand` : classe représentant la commande NLST.

`FTPNotImplementedCommand` : classe représentant une commande inconnue.

`FTPPassCommand` : classe représentant la commande PASS.

`FTPPasvCommand` : classe représentant la commande PASV.

`FTPPortCommand` : classe représentant la commande PORT.

`FTPPwdCommand` : classe représentant la commande PWD

`FTPQuitCommand` : classe représentant la commande QUIT.

`FTPRetrCommand` : classe représentant la commande RETR.

`FTPStorCommand` : classe représentant la commande STOR.

`FTPSystCommand` : classe représentant la commande SYST.

`FTPTypeCommand` : classe représentant la commande TYPE.

`FTPUserCommand` : classe représentant la commande USER.

- *package util*

`FailedCwdCommandException`

- *package shared*

`FTPClientConfiguration` : classe représentant la configuration d'un client.

`FTPDatabase` : classe représentant la base de données du projet. Elle contient les

informations relatives aux comptes authentifiés, l'adresse IP du serveur ou encore la liste des codes de retour et des messages.

`FTPLoggerFactory` : factory de loggers

`FTPLoggerSimpleFormatter` : formatter qui nous abstient du sucre syntaxique des loggers habituels

`FTPMessageSender` : classe abstraite représentant un "envoyeur de messages". Toute classe héritant de cette classe est capable d'envoyer des commandes ou des données à travers les sockets.

`FTPRequest` : classe représentant une requête FTP. Une requête FTP est définie par une commande (obligatoire) et un argument (optionnel).

`FTPServerConfiguration` : classe représentant la configuration du serveur.

`FTPMain` : classe principale.

Les diagrammes de classes sont répertoriés dans le dossier `uml/`.

Parcours de code

En premier lieu, une base de données est mise en place en créant une instance de la classe `FTPDatabase`, instance que l'on injectera en constructeur de la classe `FTPServer` avec un numéro de port (> 1023) et un répertoire de base. La configuration du client (`FTPClientConfiguration`) sera paramétrée dans `FTPServer`.

```
public FTPServer(int port, String baseDirectory, FTPDatabase database) {
    super(database);
    _configuration = new FTPServerConfiguration(port, baseDirectory);
}
```

Comme tout bon serveur qui se respecte, le serveur sera en écoute continue sur le port. Une fois connecté à un client, le serveur "traite" le client en lançant un processus en parallèle, et ceci après lui avoir envoyé un message de bienvenuee. Ce Runnable se verra injecter la base de données, la configuration du serveur et le manager de commandes qui contiendra les commandes acceptées par le serveur.

```
final FTPCommandManager commandManager = new FTPCommandManager();
commandManager.addCommand(...);
...
while (true) {
```

```
    ftpServer.connectToClient();
    System.out.println("--> New client connected on this server.");
    ...
    final FTPRequestHandler requestHandler = new FTPRequestHandler(ftpDatabase,
    serverConfiguration, commandManager);
    new Thread(requestHandler).start();
}
```

La configuration client sera paramétrée dans `FTPRequestHandler`. À chaque commande reçue, le command manager se chargera d'exécuter le bon traitement.

```
FTPRequestHandler {
    public synchronized void processRequest(FTPRequest request) {
        \_commandManager.execute(request.getCommand(),
        request.getArgument(), \_clientConfiguration);
}
```

Le design pattern Command est ici dans le command manager. Ainsi, chaque commande sera visitée (Visitor) pour savoir si elle doit être exécutée ou pas. La requête est donc encapsulée et son mode de traitement est inconnu du `FTPRequestHandler`.

```
public void execute(String commandString, String argument,
        FTPClientConfiguration clientConfiguration) {
    for (final FTPCommand command : \_commands) {
        if (command.accept(commandString)) {
            command.execute(argument, clientConfiguration);
            break;
        }
    }
}
```

Chaque commande a un type commun `FTPCommand`.

```
public interface FTPCommand {

    boolean accept(String command);
```

```
    void execute(String argument, FTPClientConfiguration clientConfiguration);

}
```

La commande `FTPNotImplementedCommand` acceptera toutes les commandes.

Exemple d'utilisation (mode actif)

Lancement du serveur et connexion du client

Serveur
$ java -jar ftpServer.jar 1500 repo
–> FTP server opened on 0.0.0.0, port 1500
–> Base directory is : /home/salla/workspace/m1s2/CAR/tp1/repo

Client
$ ftp 127.0.0.1 1500
Connected to localhost.
220 FTP server ready.
Name (localhost:salla): test
331 Username okay, need password.
Password:
230 User logged in, proceed.
Remote system type is Remote.

Serveur
–> New client connected on this server.
Connection initiated at 23 fevrier 2015, 19:13:04
#1 —> USER test
#1 —> PASS test
#1 —> SYST

Commande LIST

Client
$ ftp> ls
200 PORT command successful.

150 File status okay, about to open data connection.
drwxr-xr-x salla salla 4096 f?vr. 23 08:56 test -rw-r–r– salla salla 47886 f?vr. 23 08:56 graphene.odt
-rw-r–r– salla salla 29810 f?vr. 23 08:56 Ramos2.jpg
WARNING! 3 bare linefeeds received in ASCII mode
File may not have transferred correctly.
226 Closing data connection, requested file action successful.

Serveur
#1 —> PORT 127,0,0,1,233,104
#1 —> LIST

Commande CWD

Client
$ ftp> cd test
250 Requested file action okay, completed.

Serveur
#1 —> CWD test

Commande PWD

Client
ftp> pwd
257 /home/salla/workspace/m1s2/CAR/tp1/repo/test
ftp> ls
200 PORT command successful.
150 File status okay, about to open data connection.
-rw-r–r– salla salla 45 f?vr. 19 15:12 file.c
WARNING! 1 bare linefeeds received in ASCII mode
File may not have transferred correctly.
226 Closing data connection, requested file action successful.

Serveur
#1 —> PWD

#1 —> PORT 127,0,0,1,144,9
#1 —> LIST

Commande CDUP

Client
ftp> cdup
250 Requested file action okay, completed.
ftp> pwd
257 /home/salla/workspace/m1s2/CAR/tp1/repo/

Serveur
#1 —> CDUP
#1 —> PWD

Commande RETR

Client
ftp> get graphene.odt
local: graphene.odt remote: graphene.odt
200 PORT command successful.
150 File status okay, about to open data connection.
226 Closing data connection, requested file action successful.
47886 bytes received in 0.18 secs (261.1 kB/s)

Serveur
#1 —> PORT 127,0,0,1,165,160
#1 —> RETR graphene.odt

Commande STOR

Client
ftp> put signature.html
local: signature.html remote: signature.html
200 PORT command successful.
150 File status okay, about to open data connection.
226 Closing data connection, requested file action successful.

754 bytes sent in 0.00 secs (17959.2 kB/s)

Serveur
#1 —> PORT 127,0,0,1,184,59
#1 —> STOR signature.html

Commande QUIT

Client
ftp> quit
221 Service closing control connection.

Serveur
#1 —> QUIT