TP DE PROTOCOLE RÉSEAUX

TABLE DES MATIÈRES

Présentation de l'application

L'application que nous avons implémentée est une application de type client/serveur et pair à pair. Un ensemble de clients (qui possèdent initialement une donnée) veut participer à cette application.

Pour ce faire le client doit impérativement envoyer une requête contenant le mot clé « MAGIC » . Dés que le serveur reçoit ce mot clé, il envoie à son tour une requête contenant la chaîne de caractère «OK » et la communication peut commencer à ce moment là. Le client envoie maintenant une adresse et un port sur lesquels il attendra des communications avec d'autres clients. Quand un client se connecte il envoie directement sa donnée aux autres clients connectés au même serveur, il attendra à son tour de recevoir les données qui lui manquent, et dès qu'il atteint cinq données reçues, il aura le choix entre continuer à recevoir, ou quitter l'application en alertant le serveur.

LANCEMENT DE L'APPLICATION

Pour exécuter notre application il suffit de lancer le serveur, qui va écouter sur le port TCP numéro :7777

```
java Serveur
```

Le client quand à lui s'exécute en prenant l'adresse en paramètre, comme nous avons travailler qu'en local, il faut passer «localhost » en argument

(cette méthode de passer l'adresse du client en paramètre nous permet si on veut améliorer l'application et travailler sur des applications lancées dans des machines différentes)

```
java LauncherClient localhost
```

CLASSES UTILISÉES

Notre projet contient 7 classes : 2 du coté Serveur et 5 du coté Client.

1. Côté Serveur

Le serveur contient 2 classes : ThreadServeur et Serveur.

1.1 Classe ThreadServeur

La classe *ThreadServeur* hérite de la classe *Thread*. Elle récupère l'adresse et le port de chaque client puis les enregistre dans une *Map* via la classe Data que nous décririons plus tard dans le rapport. Enfin elle envoie le tout au clients connectés. Le constructeur de la classe contient une socket et une *Data*.

```
public void run(){
    try{
        BufferedReader bf = new BufferedReader(new InputStreamReader(s.getInputStream()));
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
        String request = bf.readLine ();
        if (request.equals("MAGIC")){
            System.out.println("[A new client just connected]\n");
            String reply = new String("OK".getBytes(),ENCODING);
            pw.println(reply);
            pw.flush();
            String adresse = bf.readLine();
            String port = bf.readLine();
            String word = port+"-"+adresse;
            data.idClient++;
            data.saveIdClient = data.idClient;
            data.recordClient(data.idClient,word);
            System.out.println(data.display());
            pw.println(data.display2());
            pw.flush();
            bf.close();
        }//end if
        else System.out.println("Try again");

    }catch(Exception e){
        e.printStackTrace();
    }

}
```

1.2 Classe Serveur

La classe *Serveur* est une classe qui appelle le constructeur de la *ThreadServeur* et elle instancie une *ServerSocket* sur le port 7777.

Elle attend la connexion des clients sur cette *ServerSocket*.

2. Côté Client

Le client est composé de 5 classes :

- ClientTCP
- ListenServerTCP
- ReceiveUDP
- LauncherClient
- Data

2.1 Classe Data

La classe *Data* a pour but d'enregistrer les données d'un client.

Elle contient une méthode *recordClient()* qui enregistre l'identification d'un client, le port et l'adresse dans une *Map <Integer, String>*

Elle possède une autre méthode *display2()* qui met dans une chaîne de caractère le port et

l'adresse des client connectés .

La méthode *display()* affiche dans le *Serveur* les clients qui se sont connectés.

```
public String display2(){
    String str = "";
    for(Map.Entry<Integer, String> entry : clientConnected.entrySet()){
        if(saveIdClient != entry.getKey())   /*Pour empêcher le client courant de se mettre dans la map */
            str = str+ entry.getValue()+",";

    }

    return str;
}
```

2.2 Class ClientTCP

La classe *ClientTCP* commence par demander a chaque client d'entrée sa musique ensuite elle communique avec le serveur sur le port *7777* via une *Socket* .

Le client envoie son port et son adresse au serveur, le port à été choisie avec la méthode *getLocalPort()* de le classe socket puis elle ouvre un *DatagramSocket* avec le port du Client afin de préparer l'envoie et la réception des données en UDP. Pour finir elle appelle les classes *ListenServerTCP* et *ReceiveUDP* qui héritent tous les deux de la classe *Thread*.

2.3 Class ListenServerTCP

La classe *ListenServerTCP* a un constructeur qui prend en paramètre une *Socket*, une *DatagramSocket* et un chanson, lors de l'appel du constructeur il ajoute cette chanson à une *liste static.*

La méthode *run()* de la classe enregistre le port et l'adresse des clients connectés dans une liste appelés *listPortAddress*.

Ensuite on une methode *displayClient()* qui se contente d'afficher les clients connectés.

Pour finir la classe appelle la méthode *sendMsg()* qui a pour fonction d'envoyer la chanson du client via un *DatagramPacket* avec son port et son adresse.

2.4 Classe ReceiveUDP

Cette classe a pour fonction de recevoir les paquets. Tant que le client n'a pas reçus 5 données il reste en attente.

2.5 Classe LauncherClient

La classe *LauncherClient* lance tout simplement la classe Client.

Format et Protocoles utilisés

Cette application communique avec des données sous forme de String, sauf pour le cas de la première requête que client envoie et la réponse reçue, qui elles sont sous forme de bits codé avec le codage spécial `US-ASCII`.

L'application est basée sur deux principaux protocoles : TCP/IP et UDP.

Le serveur et le client communiquent en TCP/IP , quand aux clients, ils communiquent avec le protocole UDP.

Que se passe-il lors d'une déconnexion brutale *(pas géré dans notre application)* :

Si un client se déconnecte le serveur leve une exception,la traite en enlevant ce client de la table des clients.

Si le serveur se déconnecte, les clients continueront quand même à se parler.