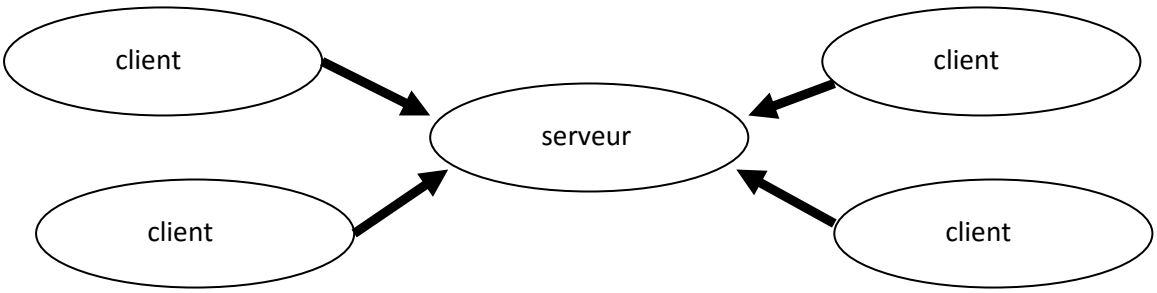
TP n°4 : client-serveur TCP en JAVA

Beaucoup d'applications utilisent des communications réseaux : pour récupérer des informations, pour communiquer des informations entre applications… Ces communications réseaux suivent différents protocoles, chacun étant adapté aux besoins des applications. Ainsi, une application de streaming a un besoin de vitesse et de débit sur le réseau (elle utilise un protocole UDP), une application de communication de données a besoin d'être fiable… Le protocole TCP permet une communication point-à-point entre deux applications. Il garantit l'intégrité des données reçues (contrairement au protocole UDP), c'est donc un protocole fiable au détriment de la vitesse de communication. Le langage JAVA permet de mettre en place des applications utilisant les protocoles TCP et UDP. Ce TP va vous permettre de découvrir le fonctionnement d'une application client-serveur TCP en JAVA.

Le but de ce travail est de disposer d'un serveur permettant d'afficher des données reçues de plusieurs clients.



1. Découvrir un programme client-serveur TCP

Voici deux applications JAVA pour tester une communication TCP client-serveur. Copiez/collez le code suivant.

Le client :

```
import java.io.*;
import java.net.*;

/**
 * Classe ClientSimple
 * Cette classe permet de créer un client pour communiquer avec un serveur TCP
 * @see ServeurSimple
 * @version 1.0
 */
public class ClientSimple {
   private InetAddress hote;
   private int port;
```

```
   private Socket socket;

/**
 * Constructeur par défaut
 * Les paramètres sont initialisés "en dur"
 */
   public ClientSimple() {
      //initialisations
      this.hote = null;
      this.port = 8888;
      this.socket = null;
      //recuperation de l'adresse IP du serveur (votre machine)
      try {
         hote = InetAddress.getLocalHost();
      }
      catch (UnknownHostException e) {}

      try {
         socket = new Socket(hote, port);
         System.out.println("Connecté au serveur: " + socket.getInetAddress() + ":" +
socket.getPort());
         //l'objet input contient le texte tapé sur la console
         BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
         //l'objet output est ce qui transmis sur la socket
         PrintWriter output = new PrintWriter(socket.getOutputStream(),true);
        String message;
         //acquisition via le clavier d'un message et envoi au serveur
         while(true) {
            message = input.readLine();
            output.println(message);
         }
      }
      catch(IOException e) {}
      try {
       //fermeture de la socket
         socket.close();
      }
      catch(IOException e) {}
   }

   public static void main( String [] args ) {
      new ClientSimple();
   }
}
```

Le serveur :

```
import java.io.*;
import java.net.*;
/**
 * Classe ServeurSimple
 * Cette classe permet de créer un serveur pour communiquer avec un client TCP
 * @see ClientSimple
 * @version 1.0
 */
public class ServeurSimple {

   private int port = 8888;
   private ServerSocket socketServeur = null;
   private Socket socket = null;

/**
 * Constructeur par défaut
 * Les paramètres sont initialisés "en dur"
 */
```

```
   public ServeurSimple() {
      try {
         socketServeur = new ServerSocket( port );
        System.out.println("Bonjour, je suis le serveur\nJ'attends des clients sur le
port " + socketServeur.getLocalPort());
      }
      catch( IOException e ) {
         System.err.println( "Impossible de créer un ServerSocket" );
      }
      while (true) {
         try {
            socket = socketServeur.accept();
            System.out.println("Connexion acceptée : " + socket.getInetAddress());
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            while(true) {
               String message = in.readLine();
               if (message == null) break;
               System.out.println(message);
            }
            socket.close();
            System.out.println("Le serveur ferme la socket avec le client");
         }
         catch( IOException e ) {}
      }
   }

   public static void main( String [] args ) {
      new ServeurSimple();
   }

}
```

Testez ce code sur vos machines. Attention, si vous utilisez JGrasp sous Windows, on ne peut pas ouvrir deux instances de JGrasp en même temps. Vous devrez lancer au moins l'une des deux applications en ligne de commande (le client par exemple) et l'autre avec JGrasp (le serveur par exemple) – vous trouverez en annexe comment procéder pour utiliser JAVA en ligne de commande sous windows. Envoyez des messages avec le client et vérifiez la bonne réception sur le serveur.

Modifiez le code, côté client, pour que l'envoi d'un mot clé spécifique (par exemple « STOP ») stoppe la communication et ferme automatiquement le client.

2. Un client qui se connecte où il veut !

Quels sont les paramètres nécessaires pour connecter un client à un serveur ?

Essayez de vous connecter sur le serveur d'un de vos camarades et de lui envoyer des messages via votre client. Pour cela, vous devez modifier des paramètres dans votre code source client.

Plutôt que de modifier « en dur » les paramètres de connexion au serveur, ajoutez, côté client, la possibilité d'entrer des arguments (cf TP 2) lors de l'exécution de votre application pour se connecter sur le serveur de son choix, avec le numéro de port de son choix. S'il n'y a pas d'arguments entrés par

l'utilisateur, faites en sorte que les connexions se déroulent par défaut en local comme ce qui est proposé dans la questions précédente.

3. Un serveur « threadé »

Essayez de lancer deux clients et d'envoyer des messages alternativement sur le serveur. Que remarquez-vous ? Pourquoi ce fonctionnement ?

Modifier le code source du serveur pour pallier au problème précédent. Cette modification consiste à ajouter un thread (cf TP 3) au serveur pour être capable de gérer plusieurs clients à la fois. Implémentez l'interface Runnable et faites en sorte que le serveur reçoive les données dans un thread. Vérifiez alors le comportement avec deux clients connectés au serveur.

4. Une IHM pour votre client-serveur

Créez deux IHM : l'une pour le client, l'autre pour le serveur. Vous commencerez par l'IHM serveur qui consiste simplement à afficher les informations texte reçues. Ensuite, vous pourrez mettre en place le client graphique avec une zone de texte et un bouton pour envoyer les données sur le serveur. Vous découperez votre code client en plusieurs méthodes :

- une méthode pour l'initialisation de l'IHM ;
- une méthode pour l'initialisation de la connexion au serveur ;
- une méthode pour l'envoi du message ;
- une méthode pour la fermeture de la connexion au serveur.

Pensez à gérer les erreurs (serveur non disponible par exemple). S'il vous reste du temps, vous pouvez améliorer les applications en :

- indiquant sur le serveur quel est l'émetteur du message ;
- indiquant sur le serveur la date et l'heure de réception du message du client.

Vous pouvez enfin mettre en place une application de « tchat » où plusieurs clients se connectent à un serveur et peuvent discuter entre eux. Pour cela, il suffit que l'IHM client affiche tous les textes reçus sur le serveur. Le serveur ne se contente donc plus uniquement de recevoir des messages, mais d'en émettre à tous les clients qui lui sont connectés. Cerise sur le gâteau pour parfaire votre IHM, chaque client dispose d'un affichage du texte selon une couleur différente.

Annexe – utilisation de java en ligne de commande sous Windows 10

Pour lancer un programme JAVA en ligne de commande, il faut tout d'abord commencer par ouvrir une fenêtre « cmd ». Tapez « cmd » dans la zone de recherche d'applications de votre Windows. Une invite de commande s'ouvre. Pour lancer la commande de compilation, il faut **écrire *javac MonProgramme.java*** en étant placé dans le répertoire contenant votre programme. Pour lancer l'exécution de la JVM, il faut taper ***java MonProgramme*** (attention, ne pas mettre l'extension .class)

Pour se déplacer dans votre arborescence :

- « d : » puis ENTER permet de changer de partition
- « cd NomRepertoire » puis ENTER permet d'aller dans le répertoire souhaité
- « cd .. » puis ENTER permet de se déplacer dans le répertoire parent
- « dir » permet de lister le contenu d'un répertoire »
- N'hésitez pas à abuser de la touche TAB permettant, lorsque vous écrivez les premières lettres d'un répertoire, de faire de l'autocomplétion

Si la commande javac ne fonctionne pas, c'est que votre environnement ne sait pas où se trouve votre compilateur. Il faut donc lui indiquer soit en écrivant le chemin complet de l'emplacement de votre programme javac soit en modifiant les paramètres de votre environnement. Ce changement n'est à faire qu'une seule fois. En voici la procédure :

1. Ouvrir un explorateur de fichiers (touches « Windows » + E) et placez-vous dans le répertoire contenant javac (souvent, c'est « C:\Program Files (x86)\Java\jdk1.8.0_60\bin » ou C:\Programmes\Java\jdk1.8.0_60\bin »). Une fois placé dans ce répertoire, vous pourrez copier/coller (étape 4) le chemin de l'emplacement du répertoire de javac en vous positionnant dans la barre d'adresse de l'explorateur.
2. Ouvrir votre panneau de configuration
3. Allez dans « Système et sécurité » puis « Système » puis « paramètres systèmes avancés » (fenêtre de gauche). Cliqez sur « Variables d'environnement… » puis dans « variables systèmes », cliquez sur la variable « Path » puis sur le bouton « Modifier » (cf Figure 1)
4. A la fin de ligne « Valeur de la variable » **(attention à ne pas effacer ce qu'elle contient déjà !!!)**, copier/coller le chemin du répertoire contenant l'exécutable « javac » de l'étape 1
5. Fermez votre invite de commande et relancez en une nouvelle. Tapez javac. Cette fois, vous pouvez désormais utiliser javac en ligne de commande

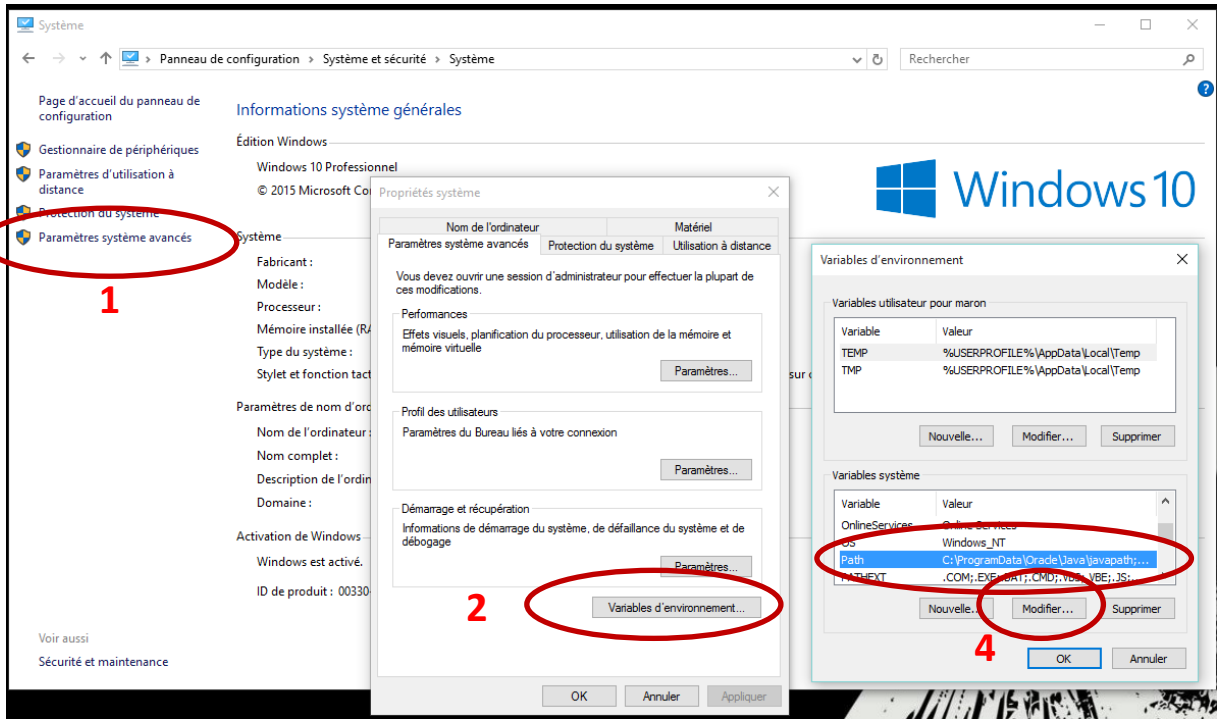


Figure 1 : configuration de la variable Path dans Windows 10