

# TD6<sup>1</sup>

## Applications client/serveur

### 1 Préalables

Ce TD aborde l'étude des bibliothèques (*libraries*) Java pour la programmation réseau. Ces classes sont rassemblées essentiellement dans les paquetages `java.net` et `java.rmi`. Les API des paquetages `java.io` et `java.nio` interviennent également.

Les pages qui suivent ne font qu'effleurer le sujet. On s'y intéresse à peu près uniquement aux fonctionnalités de bas niveaux qu'offrent les *sockets*. Ces derniers sont des composants logiciels qui permettent d'obtenir des flux de données d'une machine vers une autre. Nous les utiliserons directement. Les fonctionnalités de haut niveaux, telles celles du paquetage `java.rmi`, ne sont pas étudiées ici.

### 2 Les sockets

Cette section s'inspire très largement de *Introduction à Java™*, 2<sup>e</sup> édition, Pat Niemeyer & Jonathan Knudsen, Éditions O'Reilly, Paris, 2002.

Habituellement, Java propose une solution objet aux problèmes qu'il aborde. Le cas de la communication entre machines sur un réseau de télécommunication n'y déroge pas. Le paquetage `java.net` fournit une interface objet de manipulation des sockets<sup>2</sup>. Leur utilisation en Java est extrêmement simple!

Java utilise des sockets pour supporter trois classes de protocoles : `Socket`, `DatagramSocket` et `MulticastSocket`.

<sup>1</sup>Le texte de ce TD s'inspire très largement de ceux des ateliers logiciels écrits jadis par différents collègues. Parmi les TDs que comprend ce cours, *MCD*, *NVS*, *RFS*, *SMB* et *VAK* ont contribué de manière diverses. Certains ont écrits entièrement un TD, d'autres ont contribué ou coécrits un TD, d'autres encore ont fait du travail de relecture. Je (*PBT*) laisse mon empreinte et en profite pour remercier tous ceux qui aiment être remerciés.

<sup>2</sup>Une socket est somme toute un canal de communication entre deux processus, ces deux processus ne s'exécutant pas spécialement sur la même machine.

La classe `Socket` utilise un protocole *orienté connexion*. Une fois la connexion établie, deux applications peuvent s'échanger des données via des flux (voir le TD précédent). La communication entre les machines est alors aussi aisée que la lecture ou l'écriture d'un fichier. Le protocole garantit qu'aucune donnée n'est perdue et qu'elles arrivent dans l'ordre de leur émission. La classe `Socket` utilise le protocole TCP. Dans la suite du TD, ce type seul de connexion est étudié. Dans un soucis de complétude, voyons quand même en quoi consistent les deux autres types de sockets mentionnés plus haut.

La classe `DatagramSocket` utilise un protocole *sans connexion*. Des applications s'envoient de courts messages mais aucune connexion préalable n'est établie entre les machines et rien ne garantit que l'ordre d'arrivée des données est le même que leur ordre d'envoi ni même que tous les paquets envoyés sont reçus ! Le protocole UDP est utilisé par la classe `DatagramSocket`.

La classe `MulticastSocket` est une variante de `DatagramSocket` pour l'envoi de données à plusieurs destinataires.

### 3 Les ports

Afin de permettre à plusieurs serveurs de fonctionner sur une même machine, le système d'exploitation offre plusieurs canaux de communications entre les applications ; on associera à chaque application « serveur » un *numéro de port*. Le client —lorsqu'il interroge un serveur— utilise ce numéro de port.

L'usage des numéros de ports est réglementé par l'**IANA**. Ainsi, l'on peut constater que les numéros de port

- ↪ inférieurs à 1024 (strictement) sont réservés pour des applications tournant avec certains privilèges (*Well Known Ports*),
- ↪ compris entre 1024 et 49152 (non compris) sont réservés pour des applications enregistrées auprès de l'IANA (*Registered Ports*),
- ↪ compris entre 49152 et 65535 sont libres et destinés aux applications privées et aux attributions dynamiques.

### 4 Client : premiers pas

Dans le cadre d'une application réseau, le client est l'application qui initie la communication en se branchant sur le serveur. Ce branchement se fait par le biais de flux (*streams*). Ces flux sont obtenus et fournis par les sockets. Le client est généralement l'application qui utilisera les services offerts par le serveur.

**Exemple** Entrons immédiatement dans le vif du sujet et analysons le code suivant.

```
$ cat PortScanner.java
```

```
package nvs.alg2ir;

import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

/**
 * Tente de se connecter sur tous les ports d'une machine
 *
 * Attention : n'essayer ce programme que sur une machine que vous possédez,
 * il risque en effet d'être considéré hostile !
 *
 * Tiré de : Java Network Programming, 3rd Edition, Elliotte Rusty Harold,
 * O'Reilly, Sebastopol (CA), 2005
 */
public class PortScanner {

    public static void main(String[] args) {

        String host = "localhost";

        if (args.length > 0) {
            host = args[0];
        }

        for (int i = 1; i < 65536; ++i) {
            Socket connection = null;
            try {
                connection = new Socket(host, i);
                System.out.println("There_is_a_server_on_port_"
                    + i + "_of_" + host);
            } catch (UnknownHostException uhe) {
                System.err.println("Nom_d'hôte_incorrect_:_"+uhe);
                break ; // pour interrompre le for
            } catch (IOException e) {
                System.out.println("No_server_on_port_"
                    + i + "_of_" + host);
            } finally {
                try {
                    if (connection != null) connection.close();
                } catch (IOException e) {}
            }
        } // end for
    } // end main
}
```

Ce programme tente de se connecter à une machine distante sur *tous* les ports, les uns à la suite des autres ! Si une connexion est établie, c'est qu'un serveur tourne sur cette machine (sur ce réseau) sur ce port<sup>3</sup>.

---

<sup>3</sup> Ce code sera considéré comme hostile s'il interroge une machine qui n'est pas la votre.

Deux nouvelles classes sont introduite dans cet exemple :

↪ `Socket` et

↪ `UnknownHostException`.

La classe `Socket` est munie de plusieurs constructeurs. Celui utilisé ici prend deux arguments. Le premier est une chaîne de caractères qui représente le nom de l'hôte *avec lequel* le client désire se connecter. Ce nom peut être un nom de machine ou la représentation sous forme d'un `String` d'une adresse IP<sup>4</sup>. Le second argument du constructeur de `Socket` est le numéro de *port* de la connexion désirée.

Plusieurs scénarios d'exécutions sont possibles. Si le nom de l'hôte distant n'est pas correct, une `UnknownHostException` est lancée par le constructeur de `Socket`. Elle est interceptée et la boucle `for` est interrompue. Par contre, si la machine hôte distante peut être atteinte, le scan de ses ports commence. Si aucun serveur n'est à l'écoute sur un port testé, le `Socket` du client ne peut établir de connexion : une `IOException` est lancée, interceptée et on passe au port suivant. Enfin, si un serveur tourne sur la machine hôte distante pour un port donné, le `Socket` client établit une connexion. Comme ici, on désire uniquement tester la présence ou non d'un serveur distant, on coupe immédiatement la connexion en fermant le `Socket` à l'aide de sa méthode `close()`.

## 5 Serveur : premiers pas

Dans le cadre d'une application réseau, le serveur est l'application constamment à l'écoute (sur un port donné) de l'arrivée de clients. Lorsqu'un client se manifeste, la communication s'établit, à nouveau à l'aide de sockets.

**Exemple** Entrons immédiatement dans le vif du sujet en fournissant une contrepartie côté serveur du programme de scan des ports d'une machine de l'exemple 4.

```
$ cat LocalPortScanner.java
```

```
package nvs.alg2ir;

import java.io.IOException;
import java.net.ServerSocket;

/**
 * Tente de créer un serveur sur chaque port de la machine hôte dans le
 * but de tester quels ports sont déjà utilisés
 *
 * Tiré de : Java Network Programming, 3rd Edition, Elliotte Rusty Harold,
 * O'Reilly, Sebastopol (CA), 2005
 */
```

---

<sup>4</sup> La classe `InetAddress` représente une adresse IP dans la JVM. Vous trouverez quelques informations sur la résolution de nom (nom de l'hôte ↔ adresse IP) dans la javadoc de `InetAddress`.

```

*/
public class LocalPortScanner {

    public static void main(String[] args) {

        for (int port = 1; port <= 65535; ++port) {
            try {
                // the next line will fail and drop into the catch block if
                // there is already a server running on the port
                ServerSocket server = new ServerSocket(port);
                System.out.println("Pas_de_serveur_sur_le_port_:" + port + ".");
                try {
                    server.close() ;
                } catch (IOException ioe) {
                    System.err.println("Problème_à_la_fermeture_d'un_ " +
                        "ServerSocket_:" + ioe);
                }
            } catch (IOException ex) {
                System.out.println("There_is_a_server_on_port_ " + port + ".");
            } // end catch
        } // end for
    }
}

```

La classe `ServerSocket` est la classe enveloppant le composant logiciel permettant à un serveur d'être à l'écoute de clients. Le constructeur utilisé ici reçoit en argument le numéro du port d'écoute. Comme, sur une machine donnée, il ne peut y avoir qu'un serveur par port<sup>5</sup>, une `IOException` est lancée si un `ServerSocket` tente d'écouter un port déjà utilisé par une autre application. C'est sur cette propriété que s'appuie le code de l'exemple 5.

Remarquez que dans les exemples 4 et 5, aucune donnée n'est explicitement échangée entre client et serveur. Nous y arrivons maintenant.

## 6 Client / Serveur : premiers dialogues

Considérons maintenant le problème suivant : mettre au point une application réseau constituée d'un serveur se contentant de retourner tel quel au client tout octet que ce dernier lui envoie et d'un client capable de communiquer avec un tel serveur.

### 6.1 Serveur d'écho

**Exemple** Voici une implémentation possible d'un serveur d'écho, avec sa classe de test.

---

<sup>5</sup> C'est d'ailleurs à ça que servent les ports : à discriminer différentes applications réseaux sur une même machine hôte.

Normalement nous devrions utiliser un numéro de port supérieur à 49152 mais le port numéro 7 est justement destiné à recevoir un serveur d'écho. C'est ce numéro de port que nous utiliserons.

```
$ cat EchoServeur.java
```

```
package nvs.alg2ir;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Serveur écho simplissime : un seul client, plusieurs transactions !
 * impossible à arrêter...
 *
 * Attention : véritable écho : retransmet les octets reçus, simplement
 */
public class EchoServeur {

    public final static int ECHO_PORT = 7 ;

    private ServerSocket ss ;

    public EchoServeur() throws IOException , SecurityException {
        ss = new ServerSocket (ECHO_PORT) ;
    }

    /**
     * Méthode pour lancer le serveur : impossible de l'arrêter,
     * sauf si problème (catch) !
     */
    public void tourne() {

        Socket s = null ;

        try {

            while ((s = ss.accept()) != null) {

                // affichages divers
                System.out.println("canonical_hostname_du_client:_"+
                    s.getInetAddress().getCanonicalHostName());
                System.out.println("hostname_du_client:_"+
                    s.getInetAddress().getHostName());
                System.out.println("hostaddress_du_client:_"+
                    s.getInetAddress().getHostAddress());
                System.out.println("toString_du_client:_"+s.getInetAddress());
                System.out.println("local_port:_"+s.getLocalPort());
                System.out.println("remote_port:_"+s.getPort());

                InputStream in = null ; // pour lire les données du client
                OutputStream out = null ; // pour envoyer des données au client

                try {

                    in = s.getInputStream() ;
                    out = s.getOutputStream() ;
                    int c ;
```

```

System.out.print("srv_:");
while ( (c = in.read()) != -1) { // lecture
    System.out.print((char)c); // et si 'c' ne représente pas un
    // char avec le même encodage que la machine hôte du srv...
    out.write(c); // écho...
    if ( ((char)c == '\n') || ((char)c == '\r') ) {
        // double affichage avec \r\n !
        System.out.print("srv_:");
    }
} // fin du while in.read

} catch (IOException ioe) {
    System.err.println(ioe);
} finally { // fermeture de out, in et s
    try {
        if (out != null) {
            out.flush();
            out.close();
            System.out.println("out_fermé");
        }
    } catch (IOException ioe) {
        System.err.println(ioe);
    }
    try {
        if (in != null) {
            in.close();
            System.out.println("in_fermé");
        }
    } catch (IOException ioe) {
        System.err.println(ioe);
    }
    try {
        s.close();
        System.out.println("s_fermé");
    } catch (IOException ioe) {
        System.err.println(ioe);
    }
} // fin de finally
} // fin du while accept
} catch (IOException ioe) { // lancé par accept
    System.err.println(ioe);
} finally {
    try {
        ss.close();
        System.out.println("ss_fermé");
    } catch (IOException ioe) {
        System.err.println(ioe);
    }
} // fin finally
} // fin tourne()
}

```

```
$ cat TestEchoServeur.java
```

```
package nvs.alg2ir;

/**
 * Teste un EchoServeur en le lançant !
 */
public class TestEchoServeur {

    public static void main(String [] args) {

        EchoServeur es = null ;

        try {
            es = new EchoServeur() ;
        } catch (Exception e) {
            System.err.println(e);
            System.exit(1);
        }

        es.tourne() ;
    }
}
```

Quelques remarques :

- ↪ le port d'écoute du serveur d'écho est le port 7 : il s'agit du port du protocole echo ;
- ↪ comme déjà écrit, le `ServerSocket` est en permanence à l'écoute de la connexion d'un client. Sa méthode bloquante `accept()` retourne un `Socket` lorsqu'effectivement un client se connecte ;
- ↪ une instance de `Socket` est munie des méthodes `getInputStream()` et `getOutputStream()` qui retournent un flux en lecture et un flux en écriture, respectivement, permettant aux applications en réseau de communiquer entre elles ;
- ↪ l'`InputStream in` de la méthode `tourne()` de `EchoServeur` est un flux en lecture pour le serveur : il y lit, un par un les octets que lui envoie un client ;
- ↪ l'`OutputStream out` de la méthode `tourne()` de `EchoServeur` est un flux en écriture pour le serveur : il y écrit, un par un, immédiatement après les avoir lus, les octets qu'un client lui a envoyés ;
- ↪ quelques informations et les données envoyées par les clients sont affichées sur la sortie standard ;
- ↪ la partie « écho » en soit est très simple, ce sont les fermetures de flux et sockets et la gestion des exceptions qui sont moyennement délicats et assez verbeux.

## 6.2 Client d'écho

**Exemple** Et voici une implémentation possible d'un client d'écho, avec sa classe de test.

```
$ cat EchoClient.java
```

```
package nvs.alg2ir;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

/**
 * Client d'un serveur écho
 */
public class EchoClient {

    private Socket s ;
    private PrintWriter versSrv ;
    private BufferedReader depuisSrv ;

    public EchoClient(String hote) throws UnknownHostException , IOException {

        s = new Socket (hote,EchoServeur.ECHO_PORT) ;
            // EchoServeur est dans le même paquetage

        // informations diverses
        System.out.println("canonical_hostname_du_serveur:_"+
            s.getInetAddress().getCanonicalHostName());
        System.out.println("hostname_du_serveur:_"+
            s.getInetAddress().getHostName());
        System.out.println("hostaddress_du_serveur:_"+
            s.getInetAddress().getHostAddress());
        System.out.println("toString_du_serveur:_"+s.getInetAddress());
        System.out.println("local_port:_"+s.getLocalPort());
        System.out.println("remote_port:_"+s.getPort());

        versSrv = new PrintWriter(s.getOutputStream(), true);
        depuisSrv = new BufferedReader(new InputStreamReader(s.getInputStream()));

        System.out.println("clt:_connecté");
        // pas vraiment connecté : si serveur mono-client : mise en tampon des
        // entrées... le ServerSocket écoute en continu !
    }

    public void tourne() {

        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));

        String userInput;

        try {
            while ((userInput = stdIn.readLine()) != null) {
                versSrv.println(userInput); // dépend de l'hôte : pas grave car écho !
                String echoSrv = depuisSrv.readLine();
                System.out.println("echo:_"+ echoSrv);
            }
        }
    }
}
```

```

    } // fin du while
  } catch (IOException ioe) {
    System.err.println(ioe);
  } finally {
    versSrv.close();
    System.out.println("out_fermé");
    try {
      depuisSrv.close();
      System.out.println("in_fermé");
      s.close();
      System.out.println("s_fermé");
    } catch (IOException ioee) {
      System.err.println(ioee);
    }
  }
}
}
}
}

```

\$ cat TestEchoClient

```

package nvs.alg2ir.td06clientserveur.enonce;

import java.io.IOException;
import java.net.UnknownHostException;

/**
 * Teste un EchoClient en l'instanciant
 */
public class TestEchoClient {

    public static void main(String[] args) {

        String hote = "localhost" ;
        if (args.length == 1) hote = args[0] ;

        EchoClient ec = null ;
        try {
            ec = new EchoClient(hote) ;
        } catch (UnknownHostException uhe) {
            System.err.print("Hôte_inconnu:_:");
            System.err.println(uhe);
            System.exit(1);
        } catch (IOException ioe) {
            System.err.print("Impossible_de_se_connecter:_:");
            System.err.println(ioe);
            System.exit(1);
        }

        ec.tourne() ;
    }
}

```

Quelques remarques :

↔ le port de connexion du socket client est `EchoServeur.ECHO_PORT`.  
 Hors NetBeans, si votre `CLASSPATH` est bien défini et que vos classes compilées sont bien placées, tout se passe sans problème. Au sein de NetBeans, qui masque la variable d'environnement `CLASSPATH`, il est possible de rendre une classe visible à une autre en modifiant les pro-

- priétés du projet de la seconde et plus précisément les librairies (*libraries*) utilisées lors de la compilation ou de l'exécution ;
- ↪ un `PrintWriter` (avec *autoflush*) est utilisé pour écrire sur le flux en sortie du socket. Cela va à l'encontre de la mise en garde faite dans le TD « Entrées / Sorties » *contre* l'utilisation de ce type de flux dans une application réseau puisque la marque de fin de ligne que `println` introduit dépend de la machine hôte. Cependant, comme ici le dialogue a lieu avec un serveur d'écho, cela ne prêle à aucune conséquence ;
  - ↪ un `BufferedReader` est utilisé pour lire sur le flux en entrée du socket. La méthode `readLine()` est utilisée pour lire les données envoyées par le serveur. Notez que, comme indiqué dans le TD « Entrées / Sorties », cela peut poser problème si le client tourne sur un MAC OS 9 ;
  - ↪ les données envoyées au serveur d'écho sont lues sur l'entrée standard, converties en caractère (`InputStreamReader`), ligne par ligne (`stdin.readLine()`), avant leur envoi ligne par ligne également (`out.println(...)`);
  - ↪ les données reçues du serveur sont lues sur le flux venant du serveur ligne par ligne (`in.readLine()`) puis affichées sur la sortie standard, encore et toujours ligne par ligne (`System.out.println(...)`);
  - ↪ pour signifier la fin de la lecture sur l'entrée standard, l'utilisateur doit fournir la marque de fin de fichier : CTRL-Z sous MS-WINDOWS, CTRL-D sous UNIX.

**Au sujet de la communication client/serveur** Le client et le serveur doivent pouvoir communiquer. À cet effet il est primordial de prévoir un **protocole** de communication entre les deux parties ; l'échange d'information doit être *réglementé*.

### 6.3 Serveur d'écho multiclient

Que se passe-t-il si deux ou plusieurs clients se connectent, ou tentent de se connecter, au serveur d'écho de l'exemple 6.1 ?

Le premier client se connecte normalement au serveur et le dialogue se poursuit jusqu'à ce que le client l'interrompe. Si un second client tente de se connecter, il y arrive en ce sens que des flux d'entrée et sortie sont obtenus par ce second client. Cependant, le serveur reste bloqué dans son dialogue avec le premier client : il exécute la boucle `in.read()` (vers la ligne 60). On en conclut que le serveur de l'exemple 6.1 est bel et bien en permanence à l'écoute de nouveaux clients, même lorsqu'il exécute une autre portion de code que sa méthode `accept()`, *mais* qu'il ne peut répondre qu'à un seul client à la fois.

Comment le serveur peut-il être à l'écoute de connexions de clients et *en même temps* répondre à un client particulier ? En écoutant dans une thread séparée ! Inspirons-nous de cette pratique et mettons-la en œuvre pour permettre au serveur de répondre *simultanément* à plusieurs clients.

**Exemple** Implémentation possible d'un serveur d'écho multiclent, avec sa classe de test.

```
$ cat EchoServeurMulticlient.java
```

```
package nvs.alg2ir;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Serveur écho assez simple : plusieurs clients, plusieurs transactions !
 * impossible à arrêter...
 */
public class EchoServeurMulticlient {

    private ServerSocket ss ;

    public EchoServeurMulticlient() throws IOException , SecurityException {
        ss = new ServerSocket(EchoServeur.ECHO_PORT) ;
    }

    /**
     * Méthode pour lancer le serveur : impossible de l'arrêter,
     * sauf si problème (catch) !
     */
    public void tourne() {

        Socket s = null ;

        try {
            while (true) { // boucle infinie
                s = ss.accept() ;
                System.out.println(s.getInetAddress().getCanonicalHostName()+
                    "_s'est_connecté_au_serveur");
                (new ThreadClient(s)).start() ; // instantiation et démarrage
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        } finally {
            try {
                if (s != null) {
                    s.close() ;
                }
                ss.close() ;
            } catch (IOException ioee) {
                System.err.println(ioee);
            }
        } // fin finally
    }
}
```

```

/**
 * Classe interne nommée : thread lancée à chaque connexion d'un client
 */
private class ThreadClient extends Thread {

    Socket      s ;
    InetAddress ia ;
    InputStream in ;
    OutputStream out ;

    public ThreadClient(Socket s) throws IOException {
        this.s = s ;
        ia = s.getInetAddress() ;
        in = s.getInputStream() ;
        out = s.getOutputStream() ;
    }

    public void run() {
        int c ;
        try {
            System.out.print(ia+"_:_"); // info client
            while ( (c = in.read()) != -1) {
                out.write(c) ;
                System.out.print((char)c);
                if ( (c == '\n') || (c == '\r') ) {
                    System.out.print(ia+"_:_"); // double affichage si '\r'\n'
                }
            }
            return ;
        } catch (IOException ioe) {
            System.err.println(ioe);
        } finally {
            try {
                System.out.println(ia+"_quitte_!");
                out.close() ;
                System.out.println("out_fermé");
                in.close() ;
                System.out.println("in_fermé");
                s.close() ;
                System.out.println("s_fermé");
            } catch (IOException ioe) {
                System.err.println(ioe);
            }
        } // fin finally
    } // fin de run

} // fin de class ThreadClient
}

```

```
$ cat TestEchoServeurMulticlient.java
```

```
package nvs.alg2ir;

/**
 * Teste un EchoServeurMulticlient en le lançant !
 */
public class TestEchoServeurMulticlient {

    public static void main(String[] args) {

        EchoServeurMulticlient esm = null ;

        try {
            esm = new EchoServeurMulticlient() ;
        } catch (Exception e) {
            System.err.println(e);
            System.exit(1);
        }

        esm.tourne() ;
    }
}
```

Quelques remarques :

- ↪ l'écoute de la connexion des clients se fait dans une boucle infinie (`while(true)`) : le serveur ne s'arrête qu'en cas de problème ;
- ↪ chaque fois qu'un client se connecte, une `EchoServeurMulticlient.ThreadClient` est instanciée. Cette thread gère le dialogue entre un client et le serveur et affiche sur la sortie standard les données envoyées par le client ainsi que des informations à son sujet à chaque nouvelle ligne ;
- ↪ comme chaque client « vit » dans sa propre thread, le serveur peut répondre à chacun d'eux en parallèle, le serveur est donc bien multiclient.

## 7 Exercices

**Exercice 1** Écrivez une application réseau constituée :

- ↪ d'un serveur, `ServeurAleatoire`, qui envoie à tout client qui se connecte sur le port 49152 un nombre aléatoire compris entre 0 et `maxVal` (voir plus bas) ; la valeur est envoyée sous la forme d'une chaîne de caractères terminée par la paire « `\r\n` » ; dès la valeur envoyée, la communication est rompue côté serveur ;
- ↪ d'un client, `ClientAleatoire`, qui tente de se connecter à un serveur `ServeurAleatoire` dont le nom de la machine hôte est fourni en argument de la ligne de commande ; une fois la connexion établie, le client envoie au format binaire brut au serveur un `int, maxVal`, lu sur l'entrée

standard, puis attend la réponse du serveur ; une fois celle-ci reçue, le client rompt la connexion.

N'hésitez pas à réaliser des affichages de tests. Le serveur n'est pas multi-client.

**Exercice 2** Reprenez l'exercice du *poulailler* afin d'en faire une application « client / serveur » ;

- ↔ le serveur gère un poulailler, il accepte les connections de plusieurs clients. Ces clients pourront voir l'évolution du poulailler et interagir avec celui-ci.
- ↔ un client —lorsqu'il se connecte au serveur— peut voir le poulailler et interagir avec celui-ci.

En pratique il faudra réfléchir à un *protocole de communication* entre le client et le serveur. Pour ce faire, on peut, par exemple, utiliser une classe `Protocole` constituée d'une série de caractères (ou chaînes de caractères) représentant les échanges d'information entre le client et le serveur.

Par exemple lorsqu'une poule se déplace dans l'enclos, de la position 1,2 à la position 2,3, le serveur peut envoyer le chaîne : « \$d:1:2:2:3\$ ». C'est chaîne serait écrite ;  
`Protocole.DEBUT_LIGNE + Protocole.DEPLACER +`  
`Protocole.DELIMITEUR + 1 + Protocole.DELIMITEUR + 2 +`  
`Protocole.DELIMITEUR + 2 + Protocole.DELIMITEUR + 3 +`  
`Protocole.FIN_LIGNE`

Dans cette approche, le serveur se charge de la gestion du poulailler mais pas du GUI. Le client prend en charge le GUI et reçoit les mises à jours du serveur.

N'hésitez pas à discuter avec votre professeur des différents choix d'implémentations que vous serez amenés à faire.