



## ALG2ir-TD06 : Client / Serveur

### 1 Présentation du TD

Ce TD aborde l'étude des bibliothèques 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 niveau, 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. Leur utilisation en Java est extrêmement simple !

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

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 la protocole TCP. Dans la suite du TD, ce type seul de connexion est étudié. Dans un souci 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 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.

**Exemple 1 :** Entrons immédiatement dans le vif du sujet et analysons le code suivant.

```

1 package nvs.alg2ir.td06clientserveur.enonce;
2
3 import java.io.IOException;
4 import java.net.Socket;
5 import java.net.UnknownHostException;
6
7 /**
8  * Tente de se connecter sur tous les ports d'une machine
9  *
10 * Attention : n'essayer ce programme que sur une machine que vous possédez,
11 * il risque en effet d'être considéré hostile !
12 *
13 * Tiré de : Java Network Programming, 3rd Edition, Elliotte Rusty Harold,
14 * O'Reilly, Sebastopol (CA), 2005
15 */
16 public class PortScanner {
17
18     public static void main(String[] args) {
19
20         String host = "localhost";
21
22         if (args.length > 0) {
23             host = args[0];
24         }
25
26         for (int i = 1; i < 65536; ++i) {
27             //for (int i = 1; i < 100; ++i) {
28             Socket connection = null;
29             try {
30                 connection = new Socket(host, i);
31                 System.out.println("There is a server on port "
32                     + i + " of " + host);
33             } catch (UnknownHostException uhe) {
34                 System.err.println("Nom d'hôte incorrect : "+uhe);
35                 break; // pour interrompre le for
36             } catch (IOException e) {
37                 System.out.println("No server on port "
38                     + i + " of " + host);
39             } finally {
40                 try {
41                     if (connection != null) connection.close();
42                 } catch (IOException e) {}
43             }
44         } // end for
45     } // end main
46 }

```

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.

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'une `String` d'une adresse IP<sup>1</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()`.

## 4 Serveur : premiers pas

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

**Exemple 2 :** 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 1 .

```

1 package nvs.alg2ir.td06clientserveur.enonce;
2
3 import java.io.IOException;
4 import java.net.ServerSocket;
5
6 /**
7  * Tente de créer un serveur sur chaque port de la machine hôte dans le
8  * but de tester quels ports sont déjà utilisés
9  *
10 * Tiré de : Java Network Programming, 3rd Edition, Elliotte Rusty Harold,
11 * O'Reilly, Sebastopol (CA), 2005
12 */
13 public class LocalPortScanner {
14
15     public static void main(String[] args) {
16
17         for (int port = 1; port <= 65535; ++port) {

```

<sup>1</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`.

```

18 //for (int port = 25; port <= 26; ++port) {
19     try {
20         // the next line will fail and drop into the catch block if
21         // there is already a server running on the port
22         ServerSocket server = new ServerSocket(port);
23         System.out.println("Pas de serveur sur le port : "+port+".");
24         try {
25             server.close() ;
26         } catch (IOException ioe) {
27             System.err.println("Problème à la fermeture d'un " +
28                 "ServerSocket : "+ioe);
29         }
30     } catch (IOException ex) {
31         System.out.println("There is a server on port " + port + ".");
32     } // end catch
33 } // end for
34 }
35 }

```

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>2</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 2.

Remarquez que dans les Exemples 1 et 2, aucune donnée n'est explicitement échangée entre client et serveur. Nous y arrivons maintenant.

## 5 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.

### 5.1 Serveur d'écho

**Exemple 3 :** Voici une implémentation possible d'un serveur d'écho, avec sa classe de test.

```

1 package nvs.alg2ir.td06clientserveur.enonce;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.io.OutputStream;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8
9 /**

```

<sup>2</sup> C'est d'ailleurs à ça que servent les ports : à discriminer différentes applications réseaux sur une même machine hôte.



```

65     }
66     } // fin du while in.read
67
68     } catch (IOException ioe) {
69     System.err.println(ioe);
70     } finally { // fermeture de out, in et s
71     try {
72         if (out != null) {
73             out.flush() ;
74             out.close() ;
75             System.out.println("out fermé");
76         }
77     } catch (IOException ioe) {
78     System.err.println(ioe);
79     }
80     try {
81         if (in != null) {
82             in.close() ;
83             System.out.println("in fermé");
84         }
85     } catch (IOException ioe) {
86     System.err.println(ioe);
87     }
88     try {
89         s.close() ;
90         System.out.println("s fermé");
91     } catch (IOException ioe) {
92     System.err.println(ioe);
93     }
94     } // fin de finally
95     } // fin du while accept
96     } catch (IOException ioe) { // lancé par accept
97     System.err.println(ioe);
98     } finally {
99     try {
100        ss.close() ;
101        System.out.println("ss fermé");
102    } catch (IOException ioe) {
103    System.err.println(ioe);
104    }
105    } // fin finally
106    } // fin tourne()
107 }

```

```

1 package nvs.alg2ir.td06clientserveur.enonce;
2
3 /**
4  * Teste un EchoServeur en le lançant !
5  */
6 public class TestEchoServeur {
7
8     public static void main(String [] args) {
9
10        EchoServeur es = null ;
11

```

```

12     try {
13         es = new EchoServeur() ;
14     } catch (Exception e) {
15         System.err.println(e);
16         System.exit(1);
17     }
18
19     es.tourne() ;
20 }
21 }

```

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.

## 5.2 Client d’écho

**Exemple 4 :** Et voici une implémentation possible d’un client d’écho, avec sa classe de test.

```

1 package nvs.alg2ir.td06clientserveur.enonce;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.Socket;
8 import java.net.UnknownHostException;
9
10 /**
11  * Client d'un serveur écho
12  */
13 public class EchoClient {
14
15     private Socket          s ;
16     private PrintWriter    versSrv ;
17     private BufferedReader  depuisSrv ;

```

```

18
19 public EchoClient(String hote) throws UnknownHostException , IOException {
20
21     s = new Socket(hote, EchoServeur.ECHO.PORT) ;
22         // EchoServeur est dans le même paquetage
23
24     // informations diverses
25     System.out.println("canonical hostname du serveur : "+
26         s.getInetAddress().getCanonicalHostName());
27     System.out.println("hostname du serveur : "+
28         s.getInetAddress().getHostName());
29     System.out.println("hostaddress du serveur : "+
30         s.getInetAddress().getHostAddress());
31     System.out.println("toString du serveur : "+s.getInetAddress());
32     System.out.println("local port : "+s.getLocalPort());
33     System.out.println("remote port : "+s.getPort());
34
35     versSrv = new PrintWriter(s.getOutputStream(), true);
36     depuisSrv = new BufferedReader(new InputStreamReader(s.getInputStream()));
37
38     System.out.println("clt : connecté");
39     // pas vraiment connecté : si serveur mono-client : mise en tampon des
40     // entrées... le ServerSocket écoute en continu !
41 }
42
43 public void tourne() {
44
45     BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
46
47     String userInput;
48
49     try {
50         while ((userInput = stdIn.readLine()) != null) {
51             versSrv.println(userInput); // dépend de l'hôte : pas grave car écho !
52             String echoSrv = depuisSrv.readLine();
53             System.out.println("echo : " + echoSrv);
54         } // fin du while
55     } catch (IOException ioe) {
56         System.err.println(ioe);
57     } finally {
58         versSrv.close();
59         System.out.println("out fermé");
60         try {
61             depuisSrv.close();
62             System.out.println("in fermé");
63             s.close();
64             System.out.println("s fermé");
65         } catch (IOException ioee) {
66             System.err.println(ioee);
67         }
68     }
69 }
70 }

```

```

1 package nvs.alg2ir.td06clientserveur.enonce;

```

```

2
3 import java.io.IOException;
4 import java.net.UnknownHostException;
5
6 /**
7  * Teste un EchoClient en l'instanciant
8  */
9 public class TestEchoClient {
10
11     public static void main(String [] args) {
12
13         String hote = "localhost" ;
14         if (args.length == 1) hote = args[0] ;
15
16         EchoClient ec = null ;
17         try {
18             ec = new EchoClient(hote) ;
19         } catch (UnknownHostException uhe) {
20             System.err.print("Hôte inconnu : ");
21             System.err.println(uhe);
22             System.exit(1);
23         } catch (IOException ioe) {
24             System.err.print("Impossible de se connecter : ");
25             System.err.println(ioe);
26             System.exit(1);
27         }
28
29         ec.tourne() ;
30     }
31 }

```

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 propriétés 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 prête 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.

### 5.3 Serveur d’écho multicielient

Que se passe-t-il si deux ou plusieurs clients se connectent, ou tentent de se connecter, au serveur d’écho de l’Exemple 3 ?

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 3 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 5 :** Implémentation possible d’un serveur d’écho multicielient, avec sa classe de test.

```

1 package nvs.alg2ir.td06clientserveur.enonce;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.io.OutputStream;
6 import java.net.InetAddress;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9
10 /**
11  * Serveur écho assez simple : plusieurs clients , plusieurs transactions !
12  * impossible à arrêter...
13  */
14 public class EchoServeurMulticielient {
15
16     private ServerSocket ss ;
17
18     public EchoServeurMulticielient() throws IOException , SecurityException {
19         ss = new ServerSocket(EchoServeur.ECHO.PORT) ;
20     }
21
22     /**
23      * Méthode pour lancer le serveur : impossible de l’arrêter ,
24      * sauf si problème (catch) !
25      */
26     public void tourne() {
27

```

```

28 Socket s = null ;
29
30 try {
31     while (true) { // boucle infinie
32         s = ss.accept() ;
33         System.out.println(s.getInetAddress().getCanonicalHostName()+
34             " s'est connecté au serveur");
35         (new ThreadClient(s)).start() ; // instantiation et démarrage
36     }
37 } catch (IOException ioe) {
38     System.err.println(ioe);
39 } finally {
40     try {
41         if (s != null) {
42             s.close() ;
43         }
44         ss.close() ;
45     } catch (IOException ioee) {
46         System.err.println(ioee);
47     }
48 } // fin finally
49 }
50
51 /**
52  * Classe interne nommée : thread lancée à chaque connexion d'un client
53  */
54 private class ThreadClient extends Thread {
55
56     Socket      s ;
57     InetAddress ia ;
58     InputStream in ;
59     OutputStream out ;
60
61     public ThreadClient(Socket s) throws IOException {
62         this.s = s ;
63         ia = s.getInetAddress() ;
64         in = s.getInputStream() ;
65         out = s.getOutputStream() ;
66     }
67
68     public void run() {
69         int c ;
70         try {
71             System.out.print(ia+" : "); // info client
72             while ( (c = in.read()) != -1) {
73                 out.write(c) ;
74                 System.out.print((char)c);
75                 if ( (c == '\n') || (c == '\r') ) {
76                     System.out.print(ia+" : "); // double affichage si '\r'\n'
77                 }
78             }
79             return ;
80         } catch (IOException ioe) {
81             System.err.println(ioe);
82         } finally {

```

```

83     try {
84         System.out.println(ia+" quitte !");
85         out.close() ;
86         System.out.println("out fermé");
87         in.close() ;
88         System.out.println("in fermé");
89         s.close() ;
90         System.out.println("s fermé");
91     } catch (IOException ioe) {
92         System.err.println(ioe);
93     }
94 } // fin finally
95 } // fin de run
96
97 } // fin de class ThreadClient
98
99 }

```

```

1  package nvs.alg2ir.td06clientserveur.enonce;
2
3  /**
4   * Teste un EchoServeurMulticlient en le lançant !
5   */
6  public class TestEchoServeurMulticlient {
7
8      public static void main(String[] args) {
9
10         EchoServeurMulticlient esm = null ;
11
12         try {
13             esm = new EchoServeurMulticlient() ;
14         } catch (Exception e) {
15             System.err.println(e);
16             System.exit(1);
17         }
18
19         esm.tourne() ;
20     }
21 }

```

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.

## 6 Exercices

**Ex1.** Écrivez une application réseau constituée :

- d'un serveur, **ServeurAleatoire**, qui envoie à tout client qui se connecte sur le port 1234 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 clientrompt la connexion.

N'hésitez pas à réaliser des affichages de tests. Le serveur n'est pas multient.