Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2019 - 2020
Ateliers Logiciels

Ateliers logiciels

Applications client / serveur

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

Les pages qui suivent ne font qu'effleurer le sujet. Nous nous y intéressons, presque exclusivement, 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 utilisons directement. Les fonctionnalités de hauts niveaux, telles celles du paquetage `java.rmi`, ne sont pas étudiées ici.

Les ressources nécessaires sont disponibles sur un dépôt [1] (*repository*) de l'instance GitLab de l'école. Nous vous invitons à le cloner.

1. https://git.esi-bru.be/ATL/ClientServer.git (consulté le 4 octobre 2019).

Version du document : 8 octobre 2019

1 Sockets

> Cette section s'inspire très largement de *Introduction à Java™*, 2e 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 [2]. Leur utilisation en Java est extrêmement simple !

Java utilise des sockets pour ce qui concerne le support de 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. 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, seul ce type 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 soit le même que leur ordre d'envoi ni même que tous les paquets envoyés soient 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.

2 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 associe 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 [3]. Ainsi, on peut constater que les numéros de port :

- inférieurs à 1 024 (strictement) sont réservés pour des applications tournant avec certains privilèges (*Well Known Ports*) ;
- compris entre 1 024 et 49 152 (ce dernier non compris) ne sont pas contrôlés par l'IANA (*Registered Ports*) et peuvent être utilisés par les utilisateurs normaux ;

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

3. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml (consulté le 4 octobre 2019).

— compris entre 49 152 et 65 535 sont libres et destinés aux applications privées et aux attributions dynamiques. Certains de ces ports sont réservés par le système [4].

3 Client

3.1 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 utilise les services offerts par le serveur.

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

```
$ cat PortScanner.java
```

```java
package esi.atl.port;

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 possedez, il
 * risque en effet d'être consideré hostile !
 *
 * Tire 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("Bad hostname : " + uhe);
```

4. Consultez le fichier `/etc/services` à ce sujet.

```
34                  break; // pour interrompre le for
35              } catch (IOException e) {
36                  System.out.println("No server on port "
37                          + i + " of " + host);
38              } finally {
39                  try {
40                      if (connection != null) {
41                          connection.close();
42                      }
43                  } catch (IOException e) {
44                  }
45              }
46          } // end for
47      }  // end main
48  }
```

Ce programme tente de se connecter à une machine hôte 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 port [5].

Deux classes sont introduites 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 *auquel* 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 [6]. 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()`.

3.2 Exercice

Analysez et testez le code de la classe `PortScannerRessources`. Cette classe effectue-t-elle les mêmes actions que la classe `PortScanner` ? Listez les différences entre ces deux classes.

Elles font la même chose, mais elles sont écrites différemment. La classe PortScanner coupe la connexion dans le "finally" et l'autre classe la ferme grâce au contenu des parenthèses suivant le "try" qui est "Socket connection = new Socket(host, i)"

```
$ cat PortScannerRessources.java
```

5. Ce code sera considéré comme hostile s'il interroge une machine qui n'est pas la vôtre.
6. La classe `InetAddress` représente une adresse internet. Vous trouverez quelques informations sur la résolution de nom (nom de l'hôte ⇝ adresse IP) dans la javadoc de `InetAddress`.

```java
package esi.atl.port;

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

public class PortScannerRessources {

    public static void main(String[] args) {

        String host = "localhost";

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

        for (int i = 1; i < 65536; ++i) {
            try (Socket connection = new Socket(host, i)) {
                System.out.println("There is a server on port " + i + " of " + host);
            } catch (UnknownHostException uhe) {
                System.err.println("Bad hostname : " + uhe);
                break; // pour interrompre le for
            } catch (IOException e) {
                System.out.println("No server on port " + i + " of " + host);
            }
        } // end for
    } // end main
}
```

L'expression `try-with-resources`[7] utilisée ci-dessus et dans la suite du TD a été introduite dans le JDK 7.

4 Serveur

4.1 Premiers pas

Dans le cadre d'une application réseau, le serveur est l'application constamment à l'écoute de l'arrivée de clients sur un port donné. 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 contre-partie côté serveur du programme de scan des ports d'une machine de l'exemple de la section 3.1.

```
$ cat LocalPortScannerRessources.java
```

```java
package esi.atl.local.port;

```

7. https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html (consulté le 4 octobre 2019).

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

/**
 * Tente de creer un serveur sur chaque port de la machine hote dans le but de
 * tester quels ports sont deja utilises
 *
 * Tire de : Java Network Programming, 3rd Edition, Elliotte Rusty Harold,
 * O'Reilly, Sebastopol (CA), 2005
 */
public class LocalPortScannerRessources {

    public static void main(String[] args) {

        for (int port = 1; port <= 65535; ++port) {
            try (ServerSocket server = new ServerSocket(port)) {
                System.out.println("Pas de serveur sur le port : " + port + ".");
            } 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 [8], 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 de la section 4.1.

Remarquez que dans les exemples des sections 3.1 et 4.1, 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 Voici une implémentation possible d'un serveur d'écho, avec sa classe de test.

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 donc ce

8. C'est d'ailleurs à ça que servent les ports : à discriminer différentes applications réseaux sur une même machine hôte.

numéro de port que nous utilisons.

```
$ cat EchoServer.java
```

```java
package esi.atl.server;

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

public class EchoServer {

    public static void main(String[] args) {
        // EchoServer server = new EchoServer(7);
        // au 401 et 404 : Error new ServerSocket 7 Address already in use: NET_Bind
        EchoServer server = new EchoServer(7777);
        server.listening();
    }

    private boolean running;

    private final int port;

    public EchoServer(int port) {
        this.port = port;
        this.running = true;
    }

    public void listening() {
        int msg;
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            while (running) {
                System.out.println("Serveur listening ... ");
                try (Socket localSocket = serverSocket.accept();
                        InputStream in = localSocket.getInputStream();
                        OutputStream out = localSocket.getOutputStream()) {
                    while ((msg = in.read()) != -1) {
                        System.out.println("Message from client " + (char) msg);
                        out.write(msg);
                    }
                } catch (IOException ex) {
                    System.out.println("Error listening()" + ex.getMessage());
                    running = false;
                }
            }
        } catch (IOException ex) {
            System.out.println("Error new ServerSocket " + port + " "
                    + ex.getMessage());
            System.exit(-1);
        }
    }
}
```

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 du couple de 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 `listening()` de `EchoServer` 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 `listening()` de `EchoServer` 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 ;

— la partie « écho » en soit est très simple, les fermetures de flux et sockets sont gérée par des `try`-*with-resources* et la gestion des exceptions est moyennement délicate.

5.2 Client d'écho

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

```
$ cat EchoClient.java
```

```java
package esi.atl. client ;

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

public class EchoClient {

    public static void main(String[] args) {
        // EchoClient client = new EchoClient("localhost", 7);
        // pour test au 401 et 404 :
        EchoClient client = new EchoClient("localhost", 7777);
        client . start ();
    }

    private final String host;
    private final int port;

    public EchoClient(String host, int port) {
        this.host = host;
        this.port = port;
    }
```

```
25
26      public void start() {
27          String userInput;
28          String serverAnswer;
29          try (Socket socketClient = new Socket(host, port);
30                  PrintWriter toServer = new PrintWriter(socketClient.
31                          getOutputStream());
32                  BufferedReader fromServer = new BufferedReader(
33                          new InputStreamReader(socketClient.getInputStream()));
34                  BufferedReader keyboard = new BufferedReader(
35                          new InputStreamReader(System.in))) {
36              System.out.println("Client started ... ");
37              while ((userInput = keyboard.readLine()) != null) {
38                  toServer.println(userInput);
39                  toServer.flush();
40                  serverAnswer = fromServer.readLine();
41                  System.out.println("Response from server : " + serverAnswer);
42              }
43          } catch (IOException ex) {
44              System.out.println(ex.getMessage());
45          }
46      }
47
48  }
```

Quelques remarques :

- le port de connexion du socket client est le même que celui du serveur : 7
- un `PrintWriter` (avec *autoflush*) est utilisé pour écrire sur le flux en sortie du socket. Notez qu'en règle générale, l'utilisation de ce type de flux dans une application réseau est déconseillée car la marque de fin de ligne que `println` introduit dépend de la plateforme. 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 ;
- les données sont lues sur l'entrée standard (`System.in`), converties en caractère (`InputStreamReader`), ligne par ligne (`keyboard.readLine()`), avant leur envoi, ligne par ligne également (`toServer.println(...)`) ;
- les données reçues du serveur sont lues sur le flux venant du serveur ligne par ligne (`fromServer.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.

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'informations doit être *réglementé*.

5.3 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 de la section 5.1 ?

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 33). On en conclut que le serveur de l'exemple de la section 5.1 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 *chaque client* dans une thread dédiée à ce client ! 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 multiclient, avec une classe gérant les demandes du client dans une `Thread` séparée.

```
$ cat EchoServeurMulticlient.java
```

```
1  package esi.atl.server;
2
3  import java.io.IOException;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6
7  public class EchoServeurMulticlient {
8
9      public static void main(String[] args) {
10         // EchoServeurMulticlient server = new EchoServeurMulticlient(7);
11         // au 401 et 404 : Error listening ()Address already in use: NET_Bind
12         EchoServeurMulticlient server = new EchoServeurMulticlient(7777);
13         server.listening();
14     }
15
16     private boolean running;
17     private final int port;
18
19     public EchoServeurMulticlient(int port) {
20         this.port = port;
21         this.running = true;
22     }
23
24     public void listening() {
25         try (ServerSocket serverSocket = new ServerSocket(port)) {
26             while (running) {
27                 System.out.println("Serveur listening ... ");
28                 Socket localSocket = serverSocket.accept();
29                 new ThreadClient(localSocket).start();
30             }
```

```
31          } catch (IOException ex) {
32              System.out.println("Error  listening ()" + ex.getMessage());
33          }
34      }
35
36  }
```

```
$ cat ThreadClient.java
```

```
1   package esi.atl.server;
2
3   import java.io.IOException;
4   import java.io.InputStream;
5   import java.io.OutputStream;
6   import java.net.Socket;
7
8   public class ThreadClient
9           extends Thread {
10
11      private final Socket localSocket;
12
13      public ThreadClient(Socket localSocket) {
14          this.localSocket = localSocket;
15      }
16
17      @Override
18      public void run() {
19          int msg;
20          try (InputStream in = localSocket.getInputStream();
21                  OutputStream out = localSocket.getOutputStream()) {
22              while ((msg = in.read()) != -1) {
23                  System.out.println("Message from client " + (char) msg);
24                  out.write(msg);
25              }
26          } catch (IOException ex) {
27              System.out.println("Error run()"
28                      + ex.getMessage());
29          }
30      }
31  }
```

Quelques remarques :

— l'écoute de la connexion des clients se fait indéfiniment (`while(running)` avec `running` toujours `true`) : le serveur ne s'arrête qu'en cas de problème (suite à une exception) ou si l'attribut `running` est mise à jour ;

— chaque fois qu'un client se connecte, une `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 Exercice

Exercice 1 Écrivez une application réseau constituée :

- d'un serveur, `ServeurAleatoire` : il envoie à tout client qui se connecte sur le port 49 152 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` : il commence par tenter de se connecter à un `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 multiclient.

A Écho écho écho...

Pour celles et ceux qui considèrent que verbosité ne rime pas nécessairement avec lisibilité, voici les versions des client et serveur d'écho multiclient mettant à profit l'inférence de type [9] introduite pour les variables locales dans le JDK10 à l'aide du mot clé `var`.

```
$ cat EchoClient.java
```

```java
package esi.atl. client ;

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

public class EchoClient {

    public static void main(String[] args) {
        // EchoClient client = new EchoClient("localhost", 7);
        // pour test au 401 et 404 :
        var client = new EchoClient("localhost", 7777);
        client . start ();
    }

    private final String host;
    private final int port;

    public EchoClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void start() {
        try (var socket = new Socket(host, port);
                var toServer = new PrintWriter(socket.getOutputStream());
                var fromServer = new BufferedReader(new InputStreamReader(socket.
                        getInputStream()));
                var keyboard = new BufferedReader(new InputStreamReader(System.in))) {
            System.out.println("Client started ... ");
            String userInput;
            while ((userInput = keyboard.readLine()) != null) {
                toServer. println (userInput);
                toServer. flush ();
                var serverAnswer = fromServer.readLine();
                System.out.println("Response from server : " + serverAnswer);
            }
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }

```

9. https://developer.oracle.com/java/jdk-10-local-variable-type-inference.html (consulté le 8 octobre 2019).

```
}
```

```
$ cat EchoServeurMulticlient.java
```

```
package esi.atl.server;

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

public class EchoServeurMulticlient {

    public static void main(String[] args) {
        // EchoServeurMulticlient server = new EchoServeurMulticlient(7);
        // au 401 et 404 : Error listening()Address already in use: NET_Bind
        var server = new EchoServeurMulticlient(7777);
        server.listening();
    }

    private final boolean running = true;
    private final int port;

    public EchoServeurMulticlient(int port) {
        this.port = port;
    }

    public void listening() {
        try (var serverSocket = new ServerSocket(port)) {
            while (running) {
                System.out.println("Serveur listening ...");
                var socket = serverSocket.accept();
                new ThreadClient(socket).start();
            }
        } catch (IOException ex) {
            System.out.println("Error listening()" + ex.getMessage());
        }
    }

}
```

```
$ cat ThreadClient.java
```

```
package esi.atl.server;

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

public class ThreadClient extends Thread {

    private final Socket socket;

    public ThreadClient(Socket socket) {
        this.socket = socket;
    }

    @Override
```

```
15      public void run() {
16          try (var in = socket.getInputStream();
17                  var out = socket.getOutputStream()) {
18              int msg;
19              while ((msg = in.read()) != -1) {
20                  System.out.println("Message from client " + (char) msg);
21                  out.write(msg);
22              }
23          } catch (IOException ex) {
24              System.out.println("Error run()" + ex.getMessage());
25          }
26      }
27  }
```