**Programmation réseau**

# A. Introduction

A priori, les classes **Socket**, **ServerSocket**, **DatagramPacket** et **DatagramSocket** travaillent sur la couche Transport du modèle OSI. Mais les classes **InetAdress**, **URLConnection** et **URLEncoder** peuvent travailler sur les couches supérieures de ce même modèle.

## A.1. Classe InetAddress

Dans certains cas de programmation réseau il est uniquement nécessaire d’envoyer des données d’un ordinateur à un autre sans avoir à se soucier de la bonne réception de ces données par le destinataire.

Dans de tels cas il suffit de grouper les données dans une entité qu’on appelle datagramme et qu’on envoi sur le réseau. Un datagramme peut être définit comme étant un message qu’on envoie sur le réseau et dont l’arrivée, le temps d’arrivée et le contenu ne sont pas garantis.

Dans l’Internet le protocole UDP (User Datagram Protocol) assure cette fonctionnalité. Le paquetage java.net fourni deux classes pour faciliter la programmation réseau en mode datagramme : **DatagramPacket** et **DatagramSocket**. Les données sont mises dans un objet de type **DatagramPacket** qui est envoyé sur le réseau par le biais d’un objet de type **DatagramSocket**.

Avant d’aborder en détail la programmation en mode datagramme, nous allons introduire la classe **InetAddress** et les services qu’elle nous fournit. Cette classe met à notre disposition un ensemble de méthodes permettant la résolution des noms de machines en adresses IP et vice versa.

import java.net.\* ; import java.io.\* ; public class ResoudreNom

{ public static void main(String[] args)

{ InetAddress adresse ; try

{ adresse=InetAddress.getByName(args[0])

System.out.println(args[0]+adresse.getHostAddress()) ;

}

catch(UnknownHostException e) { }

}

}

Dans cet exemple nous fournissons au programme sur la ligne de commande comme argument le nom d’une machine. Le programme se chargera alors de trouver l’adresse IP correspondant à ce nom de machine.

Deux méthodes de la classe InetAddress sont illustrées dans cet exemple :

getByName() et getHostName(). La première permet de résoudre l’adresse IP d’une machine étant donné son nom. La seconde permet de retourner sous forme de chaîne de caractères l’adresse IP mémorisé dans l’objet de type InetAddress.

Ainsi dans cet exemple nous trouvons l’adresse IP d’une machine dont le nom est passé en argument, mémorisons cette adresse dans un objet de type InetAddress et enfin l’imprimons sur l’écran.

Revenons au mode datagramme proprement dit.

Pour envoyer des datagrammes en Java nous devons suivre les étapes suivantes :

* Obtenir l’adresse du destinataire et la mettre dans une instance de la classe **InetAddress**.
* Mettre les données et l’adresse dans une instance de la classe **DatagramPacket**
* Créer une instance de la classe **DatagramSocket** et lui confier l’envoi du datagramme.

Pour recevoir des datagrammes en Java nous devons suivre les étapes suivantes :

* Créer une instance de la classe de la classe **DatagramSocket** qui devra attendre l’arrivée de données à travers le réseau.
* Créer une instance de la classe **DatagramPacket** qui recevra données qui lui seront passées par l’instance de **DatagramSocket**.

## Exemple

import java.net.\*; import java.io.\*; public class CoteEnvoi

{ static int port=5000; // Le port UDP de destination static InetAddress adr=null; static DatagramSocket sock=null;

static String message="Salut!, mode datagramme en illustration";

static byte[] tampon=new byte[message.length()]; public static void main(String[] args) throws SocketException,

IOException

{ if(args.length!=1)

{ System.out.println("Donnez comme argument le nom du destinataire"); System.exit(1);

} try

{ adr=InetAddress.getByName(args[0]);

}

catch(UnknownHostException e)

{ System.out.println("resolution impossible");

System.exit(2);

}

tampon=message.getBytes();

DatagramPacket paquet=new DatagramPacket(tampon,tampon.length,adr,port); sock=new DatagramSocket(); sock.send(paquet);

}

}

La première chose que nous faisons est de résoudre le nom de la machine destinatrice en adresse réseau. Cette adresse sera placée dans un objet de type InetAddress.

Ensuite nous transformons notre chaîne de caractères en une suite d’octets afin de les transmettre sur le réseau. N’oublions pas que le réseau ne comprend pas les caractères Unicode (ni toute autre forme d’objet ou de variable). En effet tout ce qu’un réseau comprend est une suite d’octets qui transitent. Cette transformation se fait par la méthode getBytes() de la classe String.

Il nous faut ensuite construire un datagramme en indiquant l’emplacement des données à transmettre, leur longeur et enfin l’adresse et le port de destination.

Enfin nous ouvrons une DatagramSocket et utilisons sa méthode send() pour envoyer notre datagramme.

**Passons au programme côté récepteur.**

import java.net.\*; import java.io.\*; public class CoteReception { static int port=5000;

public static void main(String[] args) throws IOException,

SocketException

{ byte[] tampon=new byte[200];

String message;

DatagramSocket sock=new DatagramSocket(port);

DatagramPacket paquet=new

DatagramPacket(tampon,tampon.length); sock.receive(paquet); message=new String(tampon);

System.out.println(message);

}

}

Côté réception les choses sont plus faciles. Nous préparons une zone mémoire tampon pour recevoir les données. Ensuite nous créons un DatagramSocket pour écouter sur le port de destination en attente de données. L’ordre d’attente se fait par la méthode receive().

Cette méthode se charge de placer les données dans un DatagramPacket gérant lui même notre tampon. Enfin les données sont extraites du tampon sous forme de chaîne de caractères et imprimées à l’écran.

# B. Implémenter des sockets

## B.1. Introduction

Les sockets sont des outils permettant d'effectuer une transmission en mode connecté entre deux entités d'un réseau (Un client et un serveur par exemple). Les deux entités possèdent des ports qu'ils peuvent utiliser pour envoyer et recevoir des données. Chaque entité possède également une ou plusieurs adresses **IP.**

Donc, avec un port source, un port destination, une adresse **IP** source et une adresse **IP** destination, une connexion est possible entre deux entités.

## B.2. Classe Socket

La classe Socket se situe dans la librairie java.net

 **Créer des sockets**

La classe Socket associée aux flux de données, permet de lire et d'écrire des données sur le réseau. La plupart du temps, les sockets sont utilisés pour se connecter à un serveur. En fait, les serveurs proposent des services (HTTP, FTP, TELNET…) que les clients peuvent utiliser.

La classe Socket permet à un client de se connecter à un hôte :

**hôte** représente le nom de l'hôte et port représente le port sur lequel la socket va se connecter.

Par exemple, si vous vous connectez sur le port 80, vous aurez toutes les chances de créer une connexion vers un serveur web qui pourra renvoyer des pages HTML.

### I- Constructeurs de Socket

|  |  |
| --- | --- |
| **Socket**(InetAddress address, int port) | Crée une connexion vers l'hôte ayant l'adresse et le port donnés en arguments |
| **Socket**(String host, int port) | Crée une connexion avec le nom de l'hôte spécifié ainsi que le port. L'appel à cette  méthode sous entend qu'il y a une résolution de nom en adresse IP. |

 **Lire et écrire des données**

A partir de là, il est possible de créer un flux d'entrée ou un flux de sortie qui vous permettrons de communiquer dans les deux sens avec le serveur. Chaque objet Socket dispose de deux méthodes intéressantes pour le faire :

***getInputStream( )*** : Retourne le flux permettant de lire les octets provenant du serveur pour cette connexion.

***getOutputStream()*** : Retourne le flux permettant d'écrire des octets à destination du serveur pour cette connexion.

Il est souvent plus utile, surtout sur Internet, de lire et d'écrire des caractères sur le réseau.

Socket s = **new** Socket(hote, port ); BufferedReader in = **new** BufferedReader ( **new** InputStreamReader(s.getInputStream()));

Dans notre exemple, nous obtenons un flux de caractères bufférisé provenant de l'hôte. Il nous reste à effectuer une boucle de lecture des caractères :

**String** line;

**while** ( (line = in.readLine()) != **null**

 **Temps d'attente**

Sur un réseau, il y a des raisons pour lesquelles la communication peut échouer. Une socket qui tente de communiquer avec une autre machine non disponible peut bloquer un programme indéfiniment. C'est pourquoi une socket peut attendre un certain temps défini par le programmeur, puis abandonner la connexion si celle-ci ne peut se faire avant le délai imparti.

Ce délai se fixe grâce à la méthode **setSoTimeout** :

s.setSoTimeout(10000); // Timeout de 10s

Après ce délai, la socket est fermée. Toute opération de lecture ou d'écriture déclenche alors une interruption de type **InterruptedIOException**. Par conséquent, il peut être vraiment utile de lever cette exception au cas où :

try {

Socket s = new Socket(hote, port );

s.setSoTimeout(10000);

BufferedReader in = new BufferedReader ( new InputStreamReader(s.getInputStream())); String line;

while ( (line = in.readLine()) != null )

System.out.println(line);

} catch (InterruptedIOException ie) {

System.out.println("Délai d'attente dépassé");

}

L'exemple précédent ne résout pas pour autant le problème puisque l'appel au constructeur peut geler le programme si la connexion n'abouti pas. Pour résoudre ce problème, il vous faudra créer la socket dans un thread séparé.

 **Socket et Thread**

Pour illustrer le point qui vient d'être soulevé, voici une classe nommée **SocketOpener** permettant d'ouvrir une socket dans un thread séparé et de renvoyer la socket lorsqu'elle est ouverte et si elle n'a pas atteint la valeur de **timeout**:

**import** java.net.\*;

**import** java.io.\*;

**class** SocketOpener **implements** Runnable {

**private** String host;

**private int** port;

**private** Socket socket;

// Méthode statique ouvrant la socket

**public static** Socket openSocket(**String** aHost, **int** aPort,**int** timeout)

{

SocketOpener opener = **new** SocketOpener(aHost, aPort);

Thread t = **new** Thread(opener);

t.start();

**try**

{ t.join(timeout);}

**catch** (InterruptedException exception){}

**return** opener.getSocket();

}

**public** SocketOpener(**String** aHost, **int** aPort) {socket = **null**; host = aHost; port = aPort;}

// Exécution dans un thread séparé **public void** run()

{ **try**

{ socket = **new** Socket(host, port);}  **catch** (IOException exception){}

}

**public** Socket getSocket() { **return** socket;}

}

Pour utiliser cette classe :

Socket s = SocketOpener.openSocket(host,port,10000);

**if** (s==**null**)

System.out.println("La socket n'a pas pu être ouverte"); else

// Travail avec la socket

B.4. Exemple de connexion à un serveur **import** java.net.\*;

**import** java.io.\*;

**public class** ClockClientSocket

{

public static void main ( String [] args )

{ try {

Socket s = **new** Socket("localhost",8189); BufferedReader in = **new** BufferedReader( **new** InputStreamReader(s.getInputStream())); PrintWriter out = **new** PrintWriter(

s.getOutputStream(),true);

String line;

out.println("quit");

**while** (

!(line=in.readLine()).toUpperCase().equals("STOP

") )

System.out.println(line);

System.out.println("Fermeture de la connexion

...");

s.close(); }

**catch** (Exception e) {System.out.println(e);}

} }

### C. Implémenter un serveur

#### C.1. Types de serveurs

Maintenant que nous avons vu la manière de se connecter à un serveur, nous allons passer aux choses sérieuses en étudiant ce qui se passe côté serveur. En fait, un serveur est à l'écoute du client qui se connecte et réagit en fonction des données qu'il reçoit.

Il existe 2 types de serveurs : le serveur capable de gérer un seul client et le serveur multiclients.

#### C.2. Classe ServerSocket

##### Exemple d'un serveur mono-client

Cette classe permet de gérer l'envoi et la réception de données vers un client. Une fois activée, la socket serveur autorise le client à se connecter sur un port déterminé.

Pour créer une telle socket :

ServerSocket s = **new** ServerSocket (8189);

Ensuite, il faut créer une socket de type *Socket* permettant de gérer les flux vers et depuis le client :

Socket client = s.accept();

L’objet *client* ne sera crée que si un client se présente sur le port 8189.

ATTENTION : La méthode *accept()* est bloquante. Ce qui signifie qu'elle ne se termine que lorsqu'un client se connecte au port concerné. La conséquence directe est que le programme ne se terminera jamais sans l'intervention d'un utilisateur sur le serveur. La plupart du temps, il est indispensable de créer la socket serveur dans un thread séparé, de manière à ne pas trop monopoliser le serveur qui a sûrement d'autres tâches à effectuer.

Lorsque le client est connecté, le dialogue peut commencer entre le client et le serveur :

BufferedReader in = new BufferedReader (new

InputStreamReader(client.getInputStream()));

PrintWriter out = new PrintWriter(client.getOutputStream());

out et sa méthode print ou println permettra d'envoyer des caractères au client, alors que la méthode readLine de in permettra une lecture bufferisée des requêtes du client.

Pour écrire au client :

out.println("Résultat de la requête"); out.flush();

Pour écouter le client :

String line = in.readLine();

Voici un programme d'exemple d'un serveur répondant au client connecté :

**import** java.net.\*;

**import** java.io.\*;

**import** java.util.\*;

**public class** ServerSocketTest {

public static void main(String[] args)

{ try

{

ServerSocket s = new ServerSocket(8189); while (true)

{

Socket client = s.accept();

BufferedReader in = new BufferedReader( new InputStreamReader( client.getInputStream())); PrintWriter out = new PrintWriter ( client.getOutputStream() ,true);

out.println( "Bienvenue sur le serveur" ); String line;

boolean fin=false;

while ( (line=in.readLine())!=null && !fin)

{

out.println("Echo :" + line);

if (line.trim().toUpperCase().equals("QUIT"))

{

fin = TRUe; out.println("STOP");

} } client.close();

} }

catch (SocketException se)

{

System.out.println("Interruption de type SocketException");

} catch (Exception e)

{

System.out.println(e);

}

}

}

##### Exemple d'un serveur multi-clients

Le défaut du programme précédent est que seul un client peut se connecter en même temps. Dans le monde du client/serveur, il est très courant qu'un serveur puisse accepter plusieurs clients. Il faut donc que chaque client puisse être géré par le serveur dans un thread différent.

**public class ThreadedSocket extends Thread**

{

private Socket client;

public ThreadedSocket(Socket client)

{

this.client = client;

}

public void run()

{ try

{

BufferedReader buffer = new BufferedReader( new InputStreamReader(client.getInputStream()));

PrintWriter writer = new PrintWriter( client.getOutputStream(),true);

writer.println( "Bienvenue sur le serveur" ); String line; boolean fin=false;

while ( (line=buffer.readLine())!=null && !fin)

{

writer.println("Echo :" + line);

if (line.trim().toUpperCase().equals("QUIT"))

{

fin = true; writer.println("STOP");

} }

client.close();

} catch (SocketException se) {System.out.println("Interruption de type

SocketException");}

catch (Exception e) {System.out.println(e);}

} }

**public class MultiServerSocket**

{

public static void main(String[] args)

{ try

{

ServerSocket s = new ServerSocket(8189); while(true)

{

ThreadedSocket th = new ThreadedSocket(s.accept()); th.start(); }

} catch (SocketException se) {System.out.println("Interruption de type

SocketException");}

catch (Exception e) {System.out.println(e);}

}

}

### D. Connexion à des serveurs http

#### D.1. Classe URL

La classe URL permet de manipuler des adresses Internet sur différents protocoles et de récupérer des informations sur des parties de l'adresse. En effet une adresse Internet est constituée :

Du protocole ( HTTP, FTP …)

Du nom DNS du domaine ( comme *www.gita.greta.fr* )

Du chemin de la ressource ( comme */formations* )

Du nom de la ressource ( comme *fiche.asp* )

Des informations sur la requête ( comme *id=76)*

Du nom du signet au sein de la page ( comme *dates*)

Une telle adresse ressemblerait à ceci :

<http://www.gita.greta.fr/formations/fiche.asp?id=76#dates>

#####  Méthode openConnection

La classe **URL** permet aussi, grâce à sa méthode **openConnection** d'effectuer une connexion vers l'adresse qu'elle représente.

URL url = new URL ("http://www.gita.greta.fr/index.htm");

URLConnection con = url.openConnection();

Les constructeurs de la classe *URL* génèrent une exception de type *MalformedURLException* si le protocole spécifié n'existe pas ou si aucun protocole n'est spécifié. Il faut donc lever cette exception systématiquement.

#####  Méthode openStream

Cette méthode permet d'ouvrir une connexion et de récupérer un flux d'entrée de données depuis le serveur correspondant à cette connexion :

InputStream in = url.openStream(); Cette méthode est équivalente à :

InputStream in = url.openConnection().getInputStream();

EXEMPLE 1

**import** java.io.\*;

1. **import** java.net.\*;
3. **public** **class** URLopenConnectionExample2 {
5. **public** **static** **void** main(String[] args)
6. {
7. **try**
8. {
9. URL url = **new** URL("https://www.javatpoint.com/java-jdbc");
10. URLConnection urlcon=url.openConnection();
11. System.out.println("------------------------------------");
13. //get the inputstream of the open connection.
14. BufferedReader br = **new** BufferedReader(**new** InputStreamReader (urlcon.getInputStream()));
15. String i;
16. **while** ((i = br.readLine()) != **null**)
17. {
18. System.out.println(i);
19. }
20. }
21. **catch** (Exception e)
22. {
23. System.out.println(e);
24. }
25. }
26. }

-----------------------------------

<!DOCTYPE html><html lang="en"><head><meta charset="utf-8"><title>JDBC Tutorial | What is Java Database Connectivity(JDBC) - javatpoint</title><link rel="SHORTCUT ICON" href="/images/logo/icon.png" />

<link rel="stylesheet" type="text/css" href="/link.css"/>

<meta name="keywords" content="java, jdbc, tutorial, api, drivers, odbc"/><meta name="description" content="Java JDBC Tutorial or What is Java Database Connectivity(JDBC) with examples on Driver, DriverManager, Connection, Statement, ResultSet, PreparedStatement, CallableStatement, ResultSetMetaData, DatabaseMetaData, RowSet, Store Image, Fetch Image, Store file, Fetch file etc."/><meta name="viewport" content="width=device-width, initial-scale=1.0"><meta name="apple-mobile-web-app-capable" content="yes"><meta name="apple-mobile-web-app-status-bar-style" content="black"><link rel="canonical" href="https://www.javatpoint.com/java-jdbc" />

EXEMPLE 2

1. **import** java.net.URL;
2. **import** java.net.URLConnection;
3. **import** java.util.Scanner;
4. **public** **class** URLopenConnectionExample1{
6. **public** **static** **void** main(String args[]) **throws** Exception
7. {
8. URL url = **new** URL("http://www.javatpoint.com");
9. URLConnection connection = url.openConnection( );
10. String mimeType = connection.getContentType( );
11. System.out.println(" The mime type is : "+mimeType);
12. System.out.println(" The time out time of connection is : "+connection.getConnectTimeout());
13. }
14. }

The mime type is : text/html; charset=iso-8859-1

The time out time of connection is : 0