Université Claude Bernard Lyon 1

miage

ISTIL
INSTITUT DES SCIENCES ET TECHNIQUES DE L'INGENIEUR DE LYON

Concepts JAVA orienté réseau : les Sockets UDP

TABLE DES MATIERES

1. Introduction

Les sockets définissent les interfaces de programmation pour la communication selon les protocoles TCP et UDP.
Le paquetage java.net possède deux classes DatagramSocket et Socket pour représenter les sockets selon les protocoles UDP et TCP. Une 3° classe MulticastSocket permet de définir des sockets pour envoyer simultanément des datagrammes à plusieurs machines. Nous ne traiterons pas cette classe
Dans ce document, nous ne présentons que les méthodes destinées à utiliser UDP.

2. La gestion des sockets : la classe DatagramSocket

Cette classe permet de gérer l'ouverture d'un socket utilisant UDP, d'échanger des datagrammes (**DatagramPacket)** et de fermer le socket.

Public class DatagramSocket extends Object

Tous les objets DatagramSocket sont reliés à un port local sur lequel on écoute pour récupérer les données entrantes et sur lequel on envoie les datagrammes.

2.1 Ouverture d'un socket pour DTG

Le constructeur crée un socket qu'il lie à un numéro de port libre et anonyme.

Public DatagramSocket() throws SocketException

Par exemple:

```
Try{
        datagramSocket cli = new datagramSocket();
        // suite du programme
        }
        catch (SocketException ex) {
        System.err.println("Port déjà occupé")}
        }
```

Un socket *cli* est crée sur un port libre de la machine locale. Une SocketException est générée en cas de problème sur la création du socket.

D'autres constructeurs
Le constructeur *Public DatagramSocket(int Port) throws SocketException* crée un port désigné. En cas de problème, c'est Socketexception qui traite.
Le constructeur Public *DatagramSocket(int Port, InetAddress interface) throws SocketException* crée le socket sur la machine et le port désigné (utilisé pour des multi hosts).

2.2 Les sockets d'envoi et de réception de DTG

Un socket UDP peut à la fois recevoir et envoyer des datagrammes. Ceux-ci peuvent provenir d'un seul ou de plusieurs hôtes.

2.2.1 La méthode send().

Lorsque le DatagramSocket est ouvert et que le DatagramPacket est crée, la méthode send() envoie le datagramme *dp* sur le socket crée. S'il y a un problème lors de l'envoi du datagramme, une IOException est créée en cas de problème: envoi d'unn DTG de taille supérieure à celle supporté par le logiciel réseau.

Public void send(DatagramPacket dp) throws IOException

2.2.2 La méthode receive()

La méthode receive() permet d'attendre un seul datagramme, de le recevoir et de le placer dans un objet DatagramPacket.
Cette méthode bloque le processus (thread) appelant jusqu'à ce que le datagramme arrive. Si vous désirez que le programme fasse autre chose pendant qu'il attend le DTG, alors il faut utiliser un thread séparé.

Public void receive(DatagramPacket dp)throws IOexception;

Le datagramme reçu sera place dans l'objet datagramPacket *dp* qui a été crée au préalable.

Remarque sur le DTG préexistant
Le buffer de réception du DTG doit être suffisamment grand pour que tout le DTG puisse être copié sinon le reste est perdu.
La limite théorique est 65 535 octets mais les protocoles limitent la taille des données généralement à 8192 octets. Ne pas oublier qu'un dtg UDP possède 8 octets d'entête.

2.3 La fermeture de socket

L'appel de la méthode **close()** de l'objet DatagramSocket ferme le socket.
Il est particulièrement recommandé de fermer les sockets inutilisés.

3. Les datagrammes UDP : la classe DatagramPacket

En Java, un datagramme est représenté par une instance de la classe DatagramPacket de la manière suivante:

Public final class DatagramPacket extends Object.

La classe DatagramPacket utilise des constructeurs différents selon que l'on veuille envoyer ou recevoir le datagramme construit.
La longueur théorique d'un datagramme est 65535 octets. Cette valeur est théorique car dans la pratique la limite est plutôt de 8K (8192 octets). Les paquets plus gros sont tronqués à 8K et d'un point de vue **des performances, il vaut mieux choisir des tailles de 512 octets ou moins**.

3.1 Les constructeurs de création d'un DTG à envoyer

Un constructeur crée un nouveau DatagramPacket pour envoyer à un hôte. Trois constructeurs pour créer les « paquets datagrammes »:

Public DatagramPacket (byte[ ] data, int length, InetAdress destination, int port)
Public DatagramPacket(byte[ ] data, int length, SocketAdress destination, int port)
Public DatagramPacket(byte[ ] data, int offset, int length, SocketAdress destination, int port)

Le paquet est rempli en utilisant *length* octets en commençant à *Offset ou à 0*. Le *port* désigne le port de l'hôte. Les *InetAddress* ou *SocketAddress* pointent vers l'hôte distant. Reportez-vous au §4 pour créer des adresses au format voulu.

3.2 Les constructeurs pour recevoir les datagrammes

Deux constructeurs créent des objets pour recevoir des datagrammes en provenance du réseau.

Public DatagramPacket (byte[ ] buffer, int lenght)
Public DatagramPacket (byte[ ] buffer, int offset, int length)//Java1.2

Premier constructeur : lorsque le socket recevra un datagramme, il le stockera dans la zone *buffer* en commençant à *buffer[0]* et jusqu'à ce que le paquet soit complètement stocké ou jusqu'à ce que le nombre d'octets précisé soit atteint.
Si le second constructeur est utilisé, le stockage commencera dans *buffer [offset]*. La taille du buffer ne sera pas testée.
Bien dimensionner la zone de réception de l'objet DatagramPacket
Il faut dimensionner la zone de réception en tenant compte du fait que l'entête UDP d'un datagramme contient 8 octets: les numéros de port source et destination, la longueur de la zone qui suit l'entête UDP et un checksum optionnel.

3.3 Les méthodes Get

Elles permettent de retrouver des informations dans un DatagramPacket. Nous citerons 5 méthodes pour retrouver les différentes parties d'un datagramme.

1. La méthode **getAddress()**
   Public InetAddress getAddress()

La méthode getAddress() retourne une InetAddress qui donne l'adresse de l'hôte distant.

2. La méthode **getPort()**
   Public int getPort()

La méthode retourne un entier qui spécifie **le numéro du port distant**

3. La méthode **getSocketaddress()**
   Public SocketAdress getSocketaddress()

Retourne l'adresse et le numéro du port.

4. La méthode **getData()**
   Public Byte getData()

Cette méthode retourne une zone mémoire de type octet contenant les données du datagramme. Il est souvent utile de convertir ces données en une chaîne de caractères de type string.

5. La méthode **getLength()**
   Public int getLength()

La méthode getLength() retourne sous forme d'un entier le nombre d'octets de données contenus dans le DTG.

La plupart du temps, les méthodes ci-dessus sont suffisantes pour manipuler les DTG. Il existe aussi des méthodes **set** qui permettent de fixer des valeurs dans un datagramme.

3.4 La Conversion de données

Il est parfois intéressant de convertir les données contenues dans un datagramme (que vous venez de recevoir) en une chaîne de caractères qui vous permettra de les afficher.

Public String (byte[ ] bufRec, String encoding)

Le premier argument *bufRec* désigne la zone de données qui contient les données. Le 2° argument désigne le type de codage des données.

Exemple 1.
On peut convertir la zone Data DatagramPacket *dp* en une chaîne de caractère *S* de la manière suivante:

String S= new(String (dp.getData(),"ascii");

Exemple 2.

```
String s = "this is a test";
Byte[ ] chaine = s.getBytes("ascii");
```

La méthode getBytes() de **java.lang.String** est utilisée pour convertir des données et les placer dans la zone *chaine* de type octet.

Exemple 3
Pour construire un DTG, il faut convertir les données à envoyer sous forme « ascii » et les placer dans une zone de type Bytes [].

```
String s = "this is a test";
Byte[ ] chaine = s.getBytes("ascii");

Try{
        InetAddress ia = InetAddress.getByName(www.ibiblio.org);
Int porthost = 7;
DatagramPacket dp = new datagramPacket(chaine, chaine.length, ia, porthost);
}
Catch(IOException ex)
```

Remarque. La classe java.io.Byte ArrayOutputStream peut également être utilisée pour préparer des données à inclure dans le DTG.

4. Les adresses Internet

La plupart des machines hôtes ont un nom (sauf les PC qui ont des adresses temporaires). Un nom d'hôte est plus stable qu'une adresse IP. Il existe de configurations où on associe plusieurs adresses IP (donc plusieurs machines) à un même nom de machine. Dans d'autres cas, plusieurs machines ont plusieurs noms correspondant à la même adresse IP.
Pour éviter de manipuler les valeurs IP des adresses et faciliter le nommage des machines, les concepteurs Internet ont inventé les DNS (Domain Name Systems). Un serveur DNS associe un nom de machine (hostname) que les humains peuvent mémoriser (ex : le serveur bat710) à son adresse IP (que les machines utilisent). Le rôle du serveur DNS est aussi d'affecter, de manière judicieuse, les demandes de connexion aux différentes machines de même nom et d'adresses IP différentes. Cette possibilité est très souvent utilisée lorsqu'il y a un gros trafic.
Chaque machine connectée à Internet a accès à un serveur DNS sur lequel « tourne » un logiciel DNS qui peut résoudre les associations entre les noms et les adresses. Les classes InetAddress intègrent cette gestion des noms et l'accès au serveur DNS

4.1 La classe des adresses Inet

Java.net.InetAddress est la classe de représentation des adresses IP dans Java pour IPV4 et IPv6. Elle est utilisée pour représenter les adresses internet lorsqu'on doit les utiliser dans les autres classes réseau: Socket, ServerSocket, URL, DatagramSocket, DatagramPacket et autres

4.1.1 Créer de nouvelles Inet AddressObject

Il n'y a pas de constructeur public dans la classe **InetAddress**. Toutefois la classe InetAddress possède trois méthodes statiques qui retournent des **objets InetAddress** initialisés.

public static InetAddress getByName (string hostname) throws UnknownHostException

public static InetAddress getAllByName (string hostname) throws UnknownHostException

public static InetAddress getLocalHost() throws UnknownHostException

Les trois méthodes réalisent une connexion avec le serveur DNS pour remplir les informations des InetAddress. Les exceptions se produisent si la connexion au serveur DNS n'est pas autorisée, ou impossible.
La méthode **InetAddress.getByName** est la plus utilisée. Elle prend comme argument le nom de l'hôte recherché. Elle utilise le serveur DNS pour obtenir son adresse IP.

On récupère une adresse *adr1* de la manière suivante :

```
java.net.InetAddress adr1 = java.net.InetAddress.getByName("www.oreilly.com") ;
```

Si la classe a déjà été importée alors il suffit de faire:

```
InetAddress adr1 = java.net.InetAddress.getByName("www.oreilly.com") ;
```

La méthode fait une exception si le serveur portant le nom demandé n'existe pas ; aussi faut-il exécuter un try.

```
try{
InetAddress adr1 = java.net.InetAddress.getByName("www.oreilly.com") ;
system.out. println(adr1);
}
catch ( unknownHostException ex){
system.out.println("could not find server");
}
```

Le résultat est imprimé sur OutputStream du système.

Remarque utile pour le TP

Dans les cas, peu fréquents, où la machine ne possède pas de nom ou bien s'il n'y a pas de serveur DNS; on peut utiliser la méthode **getByName** avec une chaine de caractères contenant l'adresse IP de la machine. Dans ce cas, il n'y aura pas d'accès au serveur DNS mais l'adresse sera construite à partir de la chaîne de caractères.

```
InetAddress adr1 = java.net.InetAddress.getByName("192.68.1.10");
```

L'adresse de bouclage ou loopback ("127.0.0.1" en IPV4) permet de tester un programme sans réseau car il y a bouclage du port de sortie sur l'entrée. Pour utiliser le test en loopback, il faut ouvrir le socket local sur le même n° de port que le port distant demandé.

4.1.2 Le timer S0_TIMEOUT

Normalement, lorsqu'on lit des informations sur un socket, l'appel est bloquant jusqu'à ce que les informations désirées soient arrivées. Toutefois, afin d'éviter le blocage en cas de non réception des octets lus, l'activation de **SO_Timeout** permet de ne pas bloquer indéfiniment l'appel. Quand le timer expire, une InterruptIOException est lancée. Toutefois le socket est encore actif et on pourra toujours lire sur ce socket.
Les timeout s'expriment en millisecondes. Zero est interprété comme un délai infini: c'est la valeur par défaut.

Public void setSoTimeout(int milliseconds) throws SocketException
Public int getSoTimeout() throws socketException

If (s.getSoTimeout() == 0) s.setSoTimeout(180000;

Si le timer associé à l'objet socket s est égal à 0, alors il est positionné à 180000 millisecondes soit 3mn.

Bibliographie

Elliote Rusty Harold, Java Network Programming -Third Edition, Oreilly Edition.