Département INFORMATIQUE

Concepts JAVA orientés réseau

les Sockets-Inet UDP et TCP

Introduction

Les sockets du domaine INET permettent à deux applications situées sur deux machines du réseau Internet d'échanger des données. Les sockets INET sont à la frontière entre l'espace utilisateur des applications et l'ensemble des logiciels de communication de l'Internet niveau transport compris. Autrement dit, les sockets Inet constituent **une interface de programmation permettant aux applications d'utiliser les services de TCP et UDP pour communiquer avec les applications distantes**.

En JAVA, le système de sockets est implanté sous forme de paquetages. **Java.net** possède trois classes **DatagramSocket, ServerSocket et Socket** pour représenter les sockets selon les protocoles UDP et TCP..

Définition de la communication au sens des sockets

Ce chapitre est identique au chapitre 9 du polycop Réseaux d'ordinateurs (S6).

La communication par sockets est établie pour le compte d'une application distribuée, entre 2 machines distantes, toutes deux reliées au même réseau Internet.

Le système par sockets considère que la communication n'est pas symétrique entre les 2 machines, (au moins dans sa première phase). En effet, il faut attribuer à chaque extrémité le rôle de client ou de serveur.

Le client correspond à l'extrémité qui émet la demande d'ouverture, il est **initiateur**. Le serveur correspond à l'extrémité qui attend les demandes d'ouvertures, il est **accepteur**.

Pour une communication, chaque machine gère, en local, 3 valeurs: le protocole, l'adresse IP appelée InetAddress, et le numéro de port. Au sens des sockets, une communication est entièrement définie par 5 paramètres:

- Le protocole TCP ou UDP utilisé,
- Inet Address de la première machine,
- Le numéro de port associé au processus (application) s'exécutant sur la première machine,
- Inet Address de la deuxième machine,
- le numéro de port associé au processus (application) s'exécutant sur la deuxième machine.

1. Gestion de la communication avec utilisation du protocole UDP.

1. Création de sockets avec UDP.

Pour envoyer ou recevoir un datagramme UDP, il s'agit pour chaque extrémité client ou serveur, d'ouvrir au préalable un socket sur sa propre machine. Chaque socket pour datagrammes (Datagram Socket) est lié à un port local (local port) à partir duquel se font les envois et les réceptions des datagrammes de l'application (port source du protocole). Si on écrit un programme client, le numéro de port n'a aucune importance: dans ce cas, il est recommandé de laisser le système de sockets choisir le numéro de port (Anonymous port). Il attribue généralement le premier port libre au-dessus de 1024. Si on écrit un programme serveur, alors celui-ci communique sur un port connu des programmes clients et dans ce cas, on doit préciser le numéro de port qui sera lié à l'application serveur.

1. Les échanges de données

On crée une communication en utilisant, à chaque extrémité, le protocole UDP qui fonctionne sans connexion préalable entre les applications. Toutefois, les échanges de données ne pourront se faire correctement sans un minimum d'organisation: définition d'un dialogue entre client et serveur. En général, lorsque l'on crée une application utilisant UDP pour communiquer, on prévoit un premier échange de datagrammes initié par le client. Celui-ci envoie un datagramme du style « Hello serveur XXX, ici client YYY», le serveur s'il a été démarré, et s'il reconnait ce message peut répondre « Hello serveur XXX écoute ». La figure 1 montre un exemple d'échange UDP entre client et serveur.

1. Le serveur ouvre « son» port local et se met en attente du premier datagramme (DTG),
2. Le client ouvre un port local et envoie le premier DTG (généralement prédéfini) puis il se met en attente d'un DTG réponse.
3. Le serveur reçoit le DTG et renvoie au client sa réponse (DTG visant à s'identifier) et il se met en attente d'un nouveau DTG client.
4. Le client et le serveur échangent ainsi des DTG selon les besoins de l'application.
5. A la fin de l'échange de données; le client doit fermer le socket local pour libérer le port.

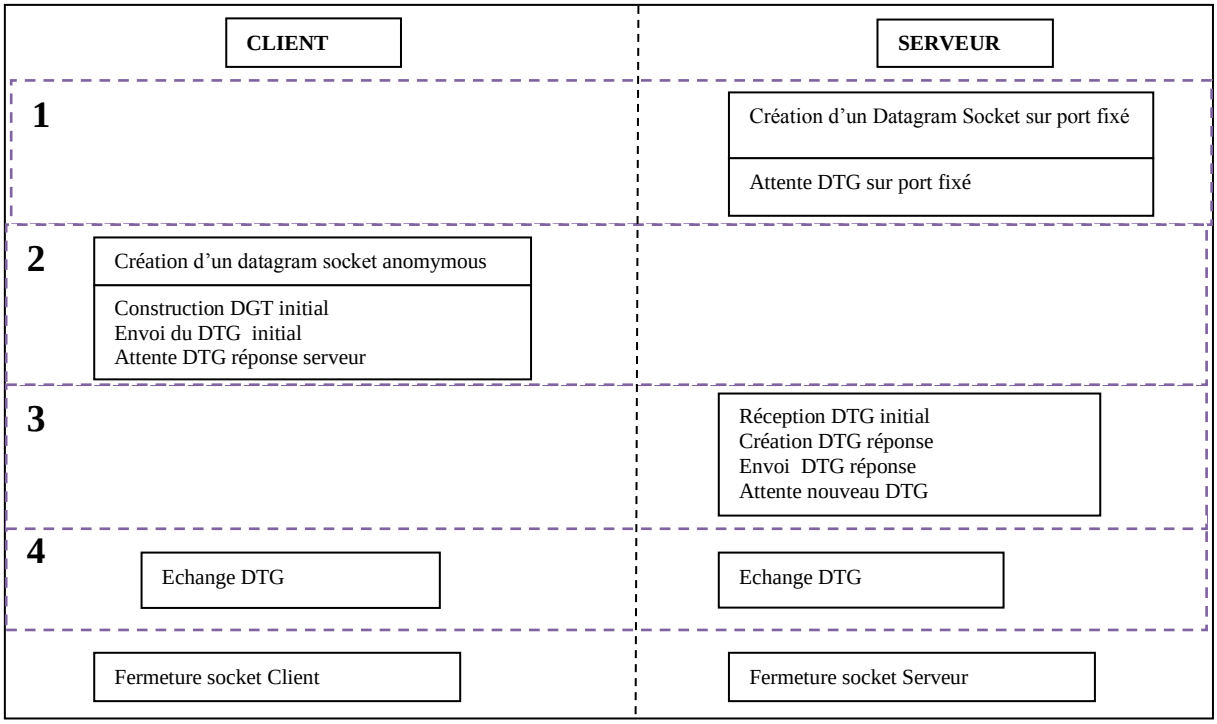

Figure 1. Exemple de dialogue en mode client-serveur avec UDP

2. Gestion de la communication avec utilisation du protocole TCP.

Dans ce cas, il faut **créer une connexion** entre les deux extrémités par le biais de TCP avant d'échanger les données.
Selon que l'on spécifie une extrémité client ou serveur fonctionnant avec TCP, on utilise 2 classes différentes. La classe **java.net.ServerSocket** est utilisée pour programmer la partie «serveur de l'application». La classe **java.net.Socket** est utilisée pour programmer la partie «client de l'application»

1. Principe d'une application utilisant TCP.

Le travail d'un serveur est de veiller en attente d'une demande de connexion entrante TCP arrivant d'un client. Chaque serveur attend sur un port particulier qui est forcément connu du client. La figure 2 montre la chronologie entre client et serveur fonctionnant sous TCP:

1. Créer un objet ServerSocket sur le port d'écoute du serveur,
2. Attendre les demandes de connexion client en utilisant la méthode accept(). La méthode accept() est bloquante jusqu'à ce qu'une demande de connexion soit faite par le client. Accept() retourne **un objet socket()représentant la connexion client-serveur**.
3. Le client et le serveur échangent les données de l'application à l'aide des méthodes getInputStream() et GetOutputStream() de l'objet socket,
4. Enfin, client ou serveur ou les deux peuvent décider de clôturer la connexion. Alors que la méthode close de l'objet Socket() ne ferme que la connexion client/serveur. Clôturer une connexion libère les ports utilisés.
5. La méthode close de l'objet ServerSocket() ferme le serveur.

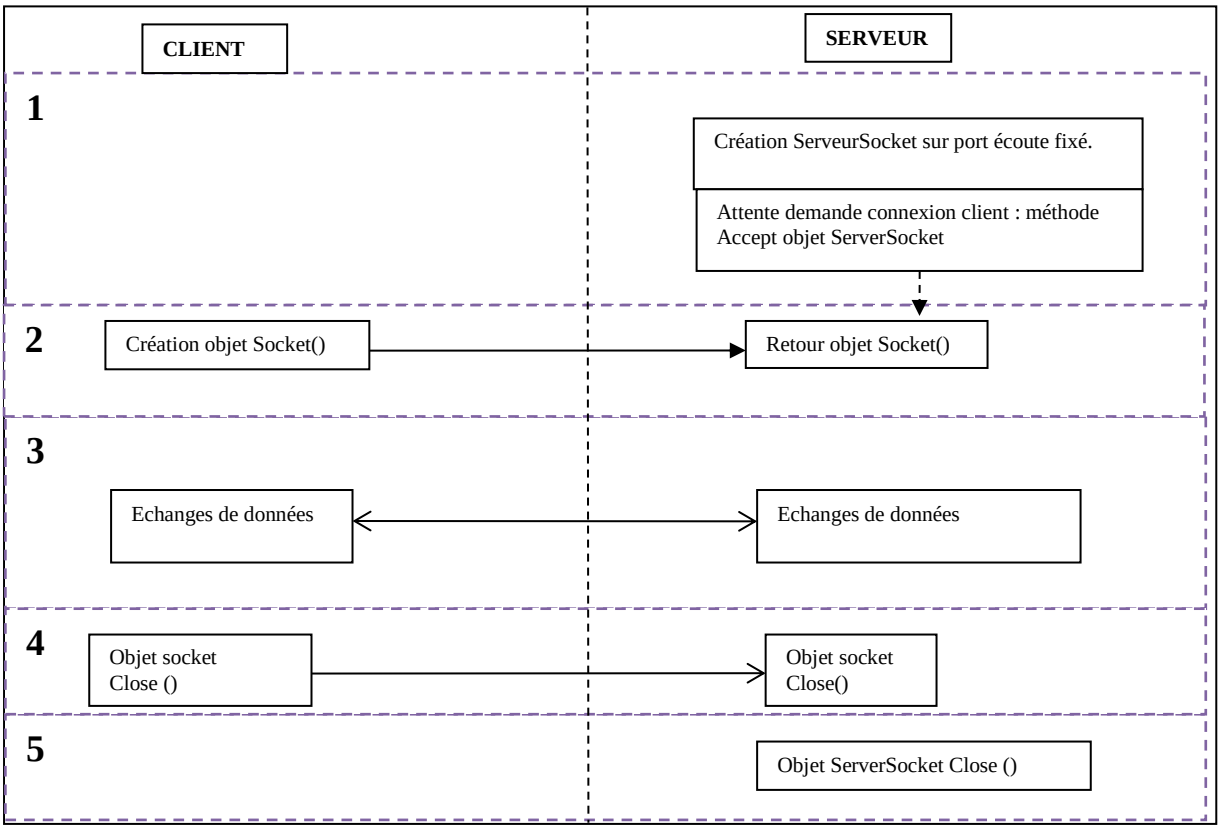

Figure 2. Chronologie d'une application fonctionnant sous TCP

2. Les échanges de données sur la connexion.

A partir du moment où la connexion est réalisée, l'application peut échanger les données avec le site distant utilisant deux méthodes de l'objet Socket:

- **getInputStream()** pour effectuer une lecture de données (flux entrant)
- **getOutputStream()** pour effectuer une écriture de données (flux sortant). Les données peuvent être stockées en attente d'envoi (bufferisées) et/ou converties (voir les filtres java).

3. La fermeture de la connexion.

C'est la méthode close() de l'objet Socket qui ferme la connexion coté client comme coté serveur.
La méthode close() de l'objet ServerSocket() libère le port d'écoute du serveur.

Programmer une application réseau en Java

1. Les adresses Internet

La gestion des adresses internet sera utilisée dans toutes les classes réseau aussi bien avec le protocole UDP qu'avec le protocole TCP.

1.1 Les serveurs de noms (DNS) pour les machines hôtes

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 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.

Dans le cas, 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 (voir §1.2.2).

1.2 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.

1.2.1 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.

1.2.2 Remarque utile pour les TP

Dans les cas, peu fréquents (hors configuration de TP), 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.

2. Communiquer avec le protocole UDP

2.1 La gestion des sockets sous UDP.

La classe **DatagramSocket** permet de gérer l'ouverture d'un socket utilisant UDP, d'échanger des datagrammes 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.

2.1.1 Ouverture d'un socket pour DTG

Création d'un socket sur port anomyme (anonymous port)
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.

Création d'un socket sur port fixé
Ce constructeur crée un socket sur port désigné.

Public DatagramSocket(int Port) throws SocketException

En cas de problème, c'est SocketException qui traite. Deux problèmes principaux peuvent apparaitre : le port demandé est déjà occupé ou bien l'application ne dispose pas de droits suffisants pour demander l'ouverture d'un port de numéro inférieur à 1024.

D'autres constructeurs
Le constructeur Public *DatagramSocket(int Port, InetAddress interface) throws SocketException* crée le socket sur la machine et le port désignés (utilisé pour des multi hosts).

2.1.2 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.

La méthode send().

Lorsque le DatagramSocket est ouvert et que le Datagramme est crée (DatagramPacket), 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

La méthode receive()

La méthode receive() permet d'attendre un seul datagramme, de le recevoir et de le placer dans un objet DatagramPacket crée au préalable (voir § 2.2.2).
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 placé dans l'objet datagramPacket dp qui a été crée au préalable.

Remarque sur le DTG qui doit préexister
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.

2.1.3 La fermeture du socket

Public void close() throws exception

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

2.2 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**.

2.2.1 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 §1 pour créer des adresses au format voulu.

2.2.2 Les constructeurs pour recevoir les datagrammes

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

Public DatagramPacket (byte[ ] buffer, int length)
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.

2.3 Accéder aux informations continues dans un datagramme ou un segment

2.3.1 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 datagrammes. Il existe aussi des méthodes **set** qui permettent de fixer des valeurs dans un datagramme.

3. Communiquer avec le protocole TCP.

3.1 La gestion des sockets sous TCP

Selon que l'on spécifie une extrémité client ou serveur fonctionnant avec TCP, on utilise 2 classes différentes. La classe **java.net.Socket** est utilisée pour programmer la partie « client de l'application », et la classe **java.net.ServerSocket** est utilisée pour programmer la partie « serveur de l'application ».

3.1.1 La création d'un socket coté client : la classe java.net.Socket .

La classe **java.net.Socket** est la classe fondamentale pour effectuer les opérations TCP du coté client. Les constructeurs sont simples. Les adresses peuvent utiliser des objets InetAddress ou des chaînes de caractères. Les ports sont spécifiés sous la forme 0 à 65 535. D'autres classes Java permettent d'utiliser les sockets mais celle-ci utilise du java natif pour communiquer avec le protocole TCP et le système.

Le constructeur Socket crée **un socket TCP sur le port désiré de l'ordinateur hôte serveur et tente une connexion sur cet hôte.**

Public Socket (InetAddress host, int Port) throws IOException

Coté client, ce constructeur utilise le premier port libre. Notez bien que l'adresse est spécifiée sous forme d'un **objet InetAddress.**

```
try {
InetAddress oreilly = InetAddress.getbyname (www.oreilly.com);
Socket toReilly = new Socket (oreilly,80);
Catch (IOException ex){
        system.err.println(ex);
        }
```

Public Socket (stringhost, int port) throws UnknownHostException, IOException
Dans ce constructeur, le nom de l'hôte est passé sous forme d'une chaîne de caractères. Si l'appel au DNS ne peut résoudre ce problème, alors le constructeur exécute le **HostException.** Si le socket ne peut être ouvert pour d'autres raisons alors, le constructeur exécute **IOException.**
Par exemple:

```
try {
Socket toReilly = new Socket(www.reilly.com,80);
}
catch(UnknownHostexception ex){
        system.err.println(ex)
 }
catch(IOException ex){
        system.err.println(ex);
}
```

3.1.2 La création de socket coté serveur : la classe ServerSocket Class.

Le travail d'un serveur est de veiller en attente d'une demande de connexion entrante TCP arrivant d'un client. Chaque serveur attend sur un port particulier qui est forcément connu du client qui demande la connexion. Java fournit la classe ServerSocket qui permet à un serveur de différencier les connexions client selon les étapes suivantes :

6. Créer un objet ServerSocket sur le port d'écoute du serveur,
7. Attendre les demandes de connexion client en utilisant la méthode accept(). La méthode accept() est bloquante jusqu'à ce qu'une demande de connexion soit faite par le client. Accept() retourne un objet socket() connectant client et serveur.

8. Le client et le serveur échangent les données de l'application à l'aide des méthodes getInputStream() et GetOutputStream()
9. Enfin, client ou serveur ou les ceux peuvent décider de clôturer la connexion.

A l'étape 2, Java permet de créer un thread pour traiter chaque nouvelle connexion, libérant ainsi le serveur qui peut traiter une nouvelle connexion arrivant sur le port d'écoute.
Il existe 4 constructeurs de ServerSocket. Nous ne présenterons que le plus courant.

Public ServerSocket(int port, int queueLenght) throws IOException,BindException.

Le constructeur permet de spécifier le port d'écoute, la taille de la file d'attente utilisée pour mettre en attente les demandes de connexion entrantes. Par défaut, les systèmes ont une taille de file d'attente typiquement 5.
Pour créer un objet ServerSocket sur le port 80 avec 6 connexions entrantes :

```
try{
ServerSocket myconnex = new ServerSocket(80,6) ;
}
catch(IOException ex){
system.err.println(ex);
}
```

Le constructeur renvoie une IOException (une Bind exception) si le ServerSocket ne peut pas être créé et lié au port spécifié. La valeur 0 sur le port laisse le système choisir un port libre.
Un serveur travaille généralement dans une boucle qui accepte et ferme les connexions. Chaque passage dans la boucle exécute une méthode Accept(). Cette méthode retourne un objet Socket qui représente la connexion entre le serveur et le client distant. Toute la transaction entre le client et le serveur utilisera les méthodes de cet objet Socket. Une fois la transaction terminée, le serveur invoquera la méthode close() de l'objet Socket pour clôturer la connexion.

Public Socket accept() throws IOException.
La méthode accept() bloque le serveur en attente d'une demande de connexion distante.

```
try{
ServerSocket server = new ServerSocket(5776);
While (true){
        Socket connexion=server.accept();

        // transaction client distant…
        connexion.close() ;
        }
```

Ci-dessus, lorsque le client distant se connecte, la méthode accept() retourne un Socket qui est stocké dans la variable locale connexion et le programme continue. Il faut toujours clôturer un socket lorsqu'on n'en a plus besoin

Lorsque l'on rajoute les exceptions, il faut distinguer celles qui clôturent la connexion de celles qui arrêtent le serveur. Généralement les exceptions de l'objet Socket() ne clôturent que la connexion.
Si le client ferme la connexion alors que le serveur exécute des opérations d'entrées sorties, une InterruptedIOException est renvoyée dès l'exécution d'une méthode getInputStream() ou getOutputStream().
Si le serveur veut arrêter son fonctionnement, il devra utiliser la méthode close() de l'objet ServerSocket.

La méthode close().
Public void close() throws exception
La méthode close() libère le port pour d'autres programmes qui peuvent alors l'utiliser

Fermer un ServerSocket ne doit pas être confondu avec fermer un Socket. La fermeture d'un ServerSocket ferme le port ouvert à la création de l'objet ServerSocket.

Gestions des exceptions sur le serveur

Lorsque l'on crée un serveur, il faut s'assurer que les sockets ouverts durant son fonctionnement seront tous fermés à l'arrêt du serveur. En effet, la plupart des protocoles spécifient que le client est responsable de la fermeture de la connexion mais dans la réalité, les clients ne respectent pas toujours cette règle. L'appel à la méthode close() doit être placé dans un bloc try{} qui permet de traiter les IOException (catch).

```
try{
ServerSocket server = new ServerSocket(5776);
While (true){
        Socket connexion = server.accept();
        try {
        // transaction client distant…
        connexion.close();
} //end while

        } // end try
        catch(IOException ex){
        // on traite ici les erreurs apparues durant le déroulement de la transaction client ainsi
que  la clôture prématurée de la connexion par le client
        } // end catch
finally{
try{
if (connexion !=null) connection.close() ;
} //end try
Catch (IOException ex){
System.err.println(ex);
} //end catch
} // end finally
} // end main
```

3.2 Les échanges d'information à l'aide des sockets.

Ainsi, l'application peut utiliser les méthodes getInputStream() et getOutputStream() de l'objet Socket pour réaliser ses échanges de données sous TCP.

3.2.1 La méthode getInputStream( )

La méthode getInputStream( ) de l'objet socket retourne un objet InputStream (flux d'entrée) qui permet de lire les données sur la connexion, c'est-à-dire sur le socket ouvert.

public InputStream getInputStream( ) throws IOException

3.2.2 La classe BufferedInputStream

Habituellement, les programmes doivent lire plusieurs caractères pour déterminer la signification des données reçues. Ce sont les protocoles d'application qui fixent la structure des données. Ils doivent également rechercher un caractère spécial qui indique que les données reçues forment une suite complète et peuvent être analysées.

Par exemple, un serveur WEB doit identifier une requête Web pour répondre. Or, le rythme d'arrivée des octets de données sur la connexion est variable. Pour pallier ces difficultés, la classe BufferedInputStream stocke les données d'un flux d'entrée (InputStream) dans les octets d'une zone mémoire protégée nommée bufindata.

Public BufferedInputStream(InputStream in)

Public BufferedInputStream(InputStream in, int bufferSize)

Le premier argument désigne l'objet flux d'entrée, le second, s'il existe, désigne la taille de la zone de stockage. Par défaut la taille de la zone de stockage entrée est de 2048 octets.

Exemple

Si on suppose que connect1 représente l'objet socket :

InputStream indata = connect1.getInputStream();
BufferedInputStream bufindata =new BufferedInputStream(indata);

Indata représente l'objet InputStream et l'objet bufindata permet d'utiliser la zone de stockage (buffer).

Les méthodes de lecture read() des objets InputStream et BufferedInputstream.

Les méthodes de lecture s'appliquent à l'objet InputStream et également à l'objet BufferedInputStream.

Plusieurs méthodes read() permettent de lire les données à partir de l'objet InputStream ou de l'objet BufferedInputStream. Nous ne présenterons que la méthode basique qui permet de lire 1 octet à la fois.

Public int read() throws exception.

Lorsque la méthode read() est utilisée sur un objet BufferedInputStream, elle provoque la lecture de l'octet suivant de la zone de stockage. Lorsque cette zone est vide, la méthode lit sur le flux d'entrée source (InputStream).

3.2.3 La méthode getOutputStream()

La méthode getOutputStream(), de l'objet socket, retourne un objet OutputStream (flux de sortie) qui permet **d'écrire les données** depuis l'application vers le socket.

Public OutputStream getOutputStream() throws Exception

Pour des raisons applicatives et/ou de performances, on évite d'écrire 1 caractère à la fois sur la connexion. On préfère écrire un message complet, par exemple une requête Web. Pour cela, on va utiliser une zone de stockage (BufferedOutputStream). On va écrire caractère par caractère dans la zone de stockage puis on déclenchera l'écriture de tous les caractères de la zone sur la connexion.

3.2.4 La classe BufferedOutputStream

La classe BufferedOutputStream stocke les données dans une zone mémoire protégée (suite d'octets nommée bufoutdata). Le stockage a lieu jusqu'à ce que la zone soit pleine ou bien un ordre de vidage intervienne (voir méthode flush). Les octets sont alors tous envoyés sur la connexion.

Public BufferedOutputStream(OutputStream out)
Public BufferedOutputStream(OutputStream out, int bufferSize)

Le premier argument désigne l'objet flux de sortie, le second s'il existe désigne la taille de la zone de stockage (buffer). Par défaut la taille de la zone de stockage sortie est de 512 octets.

Exemple

Si connect1 représente l'objet socket :

OutputStream outdata = connect1.getOutputStream();
BufferedOutputStream bufoutdata =new BufferedOutputStream(outdata);

outdata représente l'objet OutputStream et l'objet bufoutdata permet d'utiliser la zone de stockage.

Les méthodes d'écriture des objets OuputStream et BufferedOutputStream.

Les méthodes d'écriture s'appliquent à l'objet OutputStream et également à l'objet BufferedOutputStream.

Plusieurs méthodes write () permettent d'écrire les données à partir de l'objet OutputStream ou de l'objet BufferedOutputStream. Nous ne présenterons que la méthode basique qui permet d'écrire un octet à la fois.

Public write (int b) throws exception.

Lorsque cette méthode est utilisée sur un objet BufferedOutputStream, elle provoque l'écriture de l'octet suivant de la zone de stockage.

La méthode flush()

Cette méthode provoque l'envoi immédiat des données qui seraient stockées, en attente d'autres octets envoyés sur le flux de connexion.
.Le buffer d'envoi de données TCP est également vidé (flag «Push » positionné dans le segment TCP envoyé). T

Public void flush() throws exception

Notions utiles en programmation réseau

1. La Conversion de données

1.1 Conversion d'une liste d'octets en une chaine de caractères

La methode getData() retourne une liste d'octets provenant du datagramme.
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.
Conversion en chaines de caractères
On utilise le constructeur String pour transformer la liste d'octets en chaîne de caractères.

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 demandé pour la chaine (ASCII ou 8859-1).

Exemple
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");

1.2 Conversion d'une chaine de caractères en une liste d'octets

La méthode getBytes() permet de convertir une chaine de caractères en une liste d'octets.

Exemple
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[ ] Loct = s.getBytes("ASCII");

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

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

2. Programmer les temporisations (timer S0_TIMEOUT)

Normalement, lorsqu'on lit des informations sur un DatagramSocket, 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 attendus, l'activation de la temporisation **SO_Timeout** permet de définir le temps maximum d'attente, pour ne pas bloquer indéfiniment le programme. **SO_Timeout** est une option du datagramSocket. Quand le timer expire, une InterruptIOException est lancée, c'est une sous-classe de IOException.Toutefois le socket est encore actif et on pourra toujours lire sur ce socket.

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

Les timeout s'expriment en millisecondes. La méthode set*SoTimeout()* permet d'initialiser la temporisation. La valeur doit être positive ou nulle. Par défaut la valeur est initialisée à zéro (délai infini). Pour stopper une temporisation, il suffit de l'initialiser à zéro.

Exemple d'expiration de temporisation

```
try{
byte[] buffer= new byte [2056] ;
DatagramPacket dp= newDatagramPacket ( buffer, buffer.length) ;
DatagramSocket ds= new DatagramSocket(;
ds.setSoTimeout(3000); // Tempo lancée pour de 3 secondes
try{
ds.receive(dp) ;
….
}
catch SocketTimeoutException ex){
System.err.println ("Tempo expirée en 3s"+ex) ;
}
catch(SocketException ex){
System.err. println (ex);
}
```

…

L'instruction qui suit les blocs catch est exécutée après que la temporisation a expirée.

La méthode *getSoTimeout() permet de tester la valeur courante de la temporisation.*

Bibliographie

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