Programmation réseau en java : les sockets

M. Belguidoum

Université Mentouri de Constantine
Département Informatique

Plan

1. Rappel sur les entrées/sorties
2. Introduction et rappels sur les sockets
3. Socket en mode flux
4. Socket en mode datagram
5. Socket en mode multicast
6. Conclusion

Rappel sur les entrées/sorties

- lecture d'informations émises par une source externe, ou envoi d'informations à une destination externe
- sur le réseau, dans un fichier du disque dur local, dans un autre programme s'exécutant en parallèle, ou encore dans la mémoire.
- en java les entrées sorties sont gérées par les objets de flux.

Rappel sur les entrées/sorties : les flux

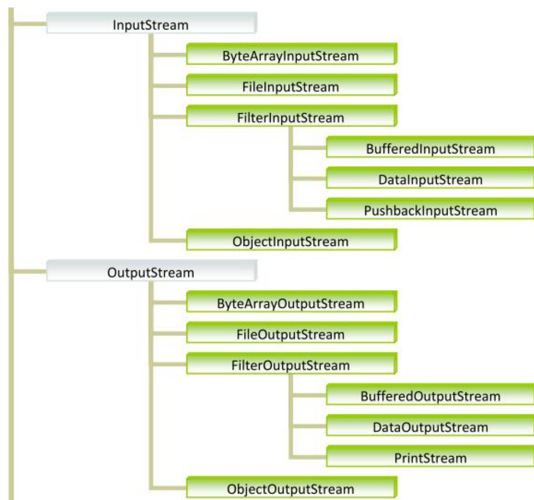
- un flot est un canal de communication dans lequel les données sont écrites ou lues de manière séquentielle.
- un *flux en lecture* permet de lire les informations de manière séquentielle.
- un *flux en écriture* permet d'écrire les informations de manière séquentielle.
- les classes d'entrées/sorties sont définies dans le paquetage java.io.
  - ceux qui manipulent des **octets** (`InputStream`, `OutputStream` et leurs classes dérivées). Les flux binaires (octets) peuvent servir à charger en mémoire des images, les enregistrer sur le disque ou enregistrer des objets (procédé nommé sérialisation) ou les charger (désérialisation).
  - ceux qui manipulent **des caractères** (`Reader`, `Writer` et leurs classes dérivées), les caractères en java sont codés sur 16 bits (Unicode). Ces flux servent à gérer les jeux de caractères (conversion, etc.).
- Les classes `InputStream`, `OutputStream`, `Reader` et `Writer` sont abstraites, c.a.d. qu'elles définissent des méthodes communes à tout leurs héritiers.

Rappel sur les entrées/sorties : les flux

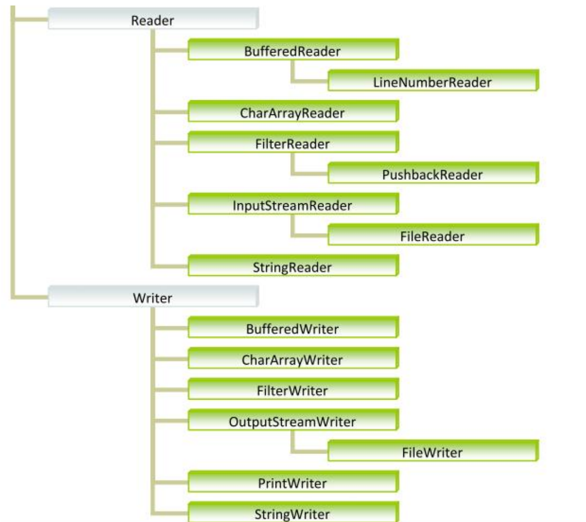
Class Hierarchy

- java.lang.Object
  - java.io.InputStream
    - java.io.FilterInputStream
      - java.io.BufferedInputStream
  - java.io.OutputStream
    - java.io.ByteArrayOutputStream
    - java.io.FilterOutputStream
      - java.io.PrintStream
  - java.io.Reader
    - java.io.BufferedReader
    - java.io.InputStreamReader
  - java.lang.Throwable
    - java.lang.Exception
      - java.io.IOException
  - java.io.Writer
    - java.io.OutputStreamWriter
    - java.io.PrintWriter

Rappel sur les entrées/sorties : les flux binaires



Rappel sur les entrées/sorties : les flux caractères



Rappel sur les entrées/sorties : les flux

- Les classes d'entrées/sorties

	Entrée	**Sortie**
Binaire	InputStream	OutputStream
Texte	Reader	Writer

- Les classes d'entrées/sorties les plus utilisées

	Entrée	**Sortie**
Binaire	FileInputStream	FileOutputStream
Texte	BufferedReader, FileReader	BufferedWriter, FileWriter

- Les méthodes d'entrées/sorties les plus courantes

	Entrée	**Sortie**
Binaire	read()	write()
Texte	readLine()	println

Rappel sur les entrées/sorties : les flux binaires

InputStream :
lecture de données

int read();

Lit un octet du flux

Bloquant

Return -1 si fin du flux

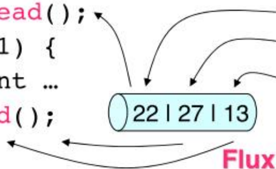
Bloquant, binaire, par flux

OutputStream :
écriture de données

void write(int b)

Écrit un octet dans le flux

```
InputStream is = …
int i = is.read();
while(i != -1) {
   … traitement …
   i = is.read();
}
is.close();
```




```
OutputStream os = …
os.write(22);
os.write(27);
os.write(13);
os.close();
```

Rappel sur les entrées/sorties : les fichiers binaires

FileInputStream : lecture de données à partir d'un fichier (hérite de InputStream)

int read();

Lit un octet du fichier

Bloquant, binaire, par flux

FileOutputStream : écriture de données dans un fichier (hérite de OutputStream)

void write(int b)

Écrit un octet dans le fichier

```
InputStream is = new FileInputStream(
   "/tmp/toto");
int i = is.read();
while(i != -1) {
   … traitement …
   i = is.read();
}
is.close();
```

```
OutputStream os = new FileOutputStream(
   "/tmp/toto");
os.write(22);
os.write(27);
os.write(13);
os.close();
```

Rappel sur les entrées/sorties : les fichiers texte

- Pour écrire dans un fichier, il faut disposer d'un flux d'écriture. On peut utiliser la classe FileWriter ou la classe PrintWriter :
  - `FileWriter(String fileName)` : crée le fichier de nom fileName
  - `PrintWriter(Writer out)` : l'argument est de type `Writer`, c.a.d. un flux d'écriture (dans un fichier, sur le réseau,...)
- Pour lire le contenu d'un fichier, il faut disposer d'un flux de lecture associé au fichier. On peut utiliser pour cela la classe `FileReader` ou `BufferedReader` :
  - `FileReader(String nomFichier)` : ouvre un flux de lecture à partir du fichier indiqué. Lance une exception si l'opération échoue.
  - `BufferedReader(Reader in)` : ouvre un flux de lecture bufferisé à partir d'un flux d'entrée `in`. Ce flux de type Reader peut provenir du clavier, d'un fichier, du réseau, ...

Rappel sur les entrées/sorties : E/S standard

- Utiliser deux flux du type InputStream et OutputStream qui s'appellent System.in et System.out.
- Le flux de données provenant du clavier est désigné par l'objet System.in de type InputStream. Il permet de lire des données caractère par caractère.
- Le type InputStream ne permet pas de lire d'un seul coup une ligne de texte. Le type BufferedReader le permet avec la méthode readLine().

```
import java.io.*;
public class Hello{
  public static void main(String []args) throws IOException
  {
    BufferedReader in=
        new BufferedReader (new InputStreamReader (System.in));
    System.out.println("Entrez votre nom : ");
    String name=in.readLine();
    System.out.println("Hello, "+name);
  } }
```

Rappel sur les entrées/sorties : Flux entrants

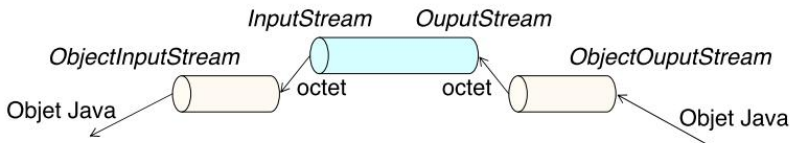
Caractère	Définition	Octet
Reader	Classe abstraite pour les flux entrant	InputStream
BufferedReader	Tampons d'entrée, peut lire une ligne	BufferedInputStream
LineNumberReader	Compte les numéros de ligne	LineNumberInputStream
CharArrayReader	Lit depuis un tableau	ByteArrayInputStream
InputStreamReader	Transforme un flux d'octets en un flux de caractères	(rien)
FileReader	Lit un fichier	FileInputStream
FilterReader	Classe abstraite pour filter le flux	FilterInputStream
PushbackReader	Permet de remettre les données lues dans le flux ce qui permet de les relire	PushbackInputStream
PipedReader	Lit depuis un "pipe"	PipedInputStream
StringReader	Lit depuis un String	StringBufferInputStream

Rappel sur les entrées/sorties : Flux sortants

Caractère	Définition	Octet
Writer	Classe abstraite pour les flux sortant	OutputStream
BufferedWriter	Tampon de sortie, utilise le `return` de du système d'exploitation	BufferedOutputStream
CharArrayWriter	Ecrit dans un tableau	ByteArrayOutputStream
FilterWriter	Classe abstraite pour filter le flux	FilterOutputStream
OutputStreamWriter	Transform un flux de caractères en flux d'octets	(rien)
FileWriter	Ecrit en binaire sur un fichier	FileOutputStream
PrintWriter	Ecrit en ASCII sur un fichier	PrintStream
PipedWriter	Ecrit dans un "pipe"	PipedOutputStream
StringWriter	Ecrit dans un String	(rien)

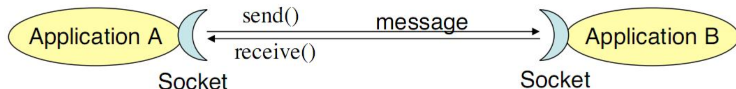
Rappel sur les entrées/sorties : la sérialisation

- **Sérialisation** : consiste à écrire des données présentes en mémoire vers un flux de données binaires, c'est donc la représentation sous forme binaire d'un objet Java
- Utilisé pour échanger des objets (envoi d'objets, persistance, ...)
- Ne marche qu'avec des Objets implantant l'interface `serializable`
- Java a introduit des outils permettant de sérialiser les objets de manière transparente et indépendante du système d'exploitation.
- La sérialisation peut s'appliquer facilement à tous les objets.



Rappels sur les sockets

- Une **socket** represente un point de communication entre un processus et un réseau.
- Un processus client et un processus serveur, lorsqu'ils communiquent, ouvrent donc chacun une socket.
- A chaque socket est associé un port de connexion. Ces numéros de port sont uniques sur un système donné, une application pouvant en utiliser plusieurs (un serveur par exemple exploite une socket par client connecté).
- Un port est identifié par un entier (16 bits).



Rappels sur les sockets

- Les ports numérotés de 0 à 511 sont les "well known ports" de l'architecture TCP/IP. Ils donnent accès aux services standard de l'interconnexion : transfert de fichiers (FTP port 21), terminal (Telnet port 23), courrier (SMTP port 25), serveur web (HTTP port 80)
- De 512 à 1023, on trouve les services Unix.
- Au delà, (1024 ...) ce sont les ports "utilisateurs" disponibles pour placer un service applicatif quelconque.
- Une connexion est identifiée de façon unique par la donnée de deux couples, une adresse IP et un numéro de port, un pour le client et un autre pour le serveur.
- Une communication client/serveur n'a pas forcément lieu via un réseau. Il est en effet possible de faire communiquer un client et un serveur sur une même machine, via l'interface de loopback, représentée par convention par l'adresse IP 127.0.0.1.

Rappels sur les sockets

- Il existe deux modes de communication : suivant si elles sont précédées ou pas d'une ouverture de communication et suivies ou pas d'une fermeture
  - **mode connecté** : la communication entre un client et un serveur est précédée d'une connexion et suivi d'une fermeture
    - Facilite la gestion d'état
    - Meilleurs contrôle des arrivées/départs de clients
    - Uniquement communication unicast
    - Plus lent au démarrage
  - **mode non connecté** : les messages sont envoyés librement
    - Plus facile à mettre en œuvre
    - Plus rapide au démarrage
- la connexion au niveau transport $\neq$ au niveau application

Mode	**couche transport**	**couche application**
Connecté	TCP	FTP, Telnet, SMTP, POP, JDBC
Non connecté	UDP	HTTP, NFS, DNS, TFTP

Rappels sur les sockets

- Une socket est donc identifiée par
  - **Une adresse IP** : une des adresses de la machine
  - **Un port** : attribué automatiquement ou choisi par le programme

=> **Adresse de Socket = Adresse IP + port**

- Une socket communique avec une autre socket via son adresse
  - **Flux** : une socket **se connecte** à une autre socket via son adresse de socket
  - **Datagram** : une socket **envoie/reçoit** des données à/d'une autre socket identifiée par son adresse de socket

Rappels sur les sockets

- Sockets en Java : uniquement orientée transport (couche 4)
- Deux API pour les sockets
  - `java.net` : API bloquante (abordée dans ce cours)
  - `java.nio.channels` (> 1.4) : API non bloquante (pas abordée)

Couche 7	Applicative	Logiciels	NFS
Couche 6	Présentation	Représentation indépendante des données	XDR
Couche 5	Session	Établit et maintient des sessions	RPC
Couche 4	Transport	Liaison entre applications de bout en bout, fragmentation, éventuellement vérification	TCP, UDP, Multicast
Couche 3	Réseau	Adressage et routage entre machines	IP
Couche 2	Liaison	Encodage pour l'envoi, détection d'erreurs, synchronisation	Ethernet
Couche 1	Physique	Le support de transmission lui-même	

Principe de fonctionnement

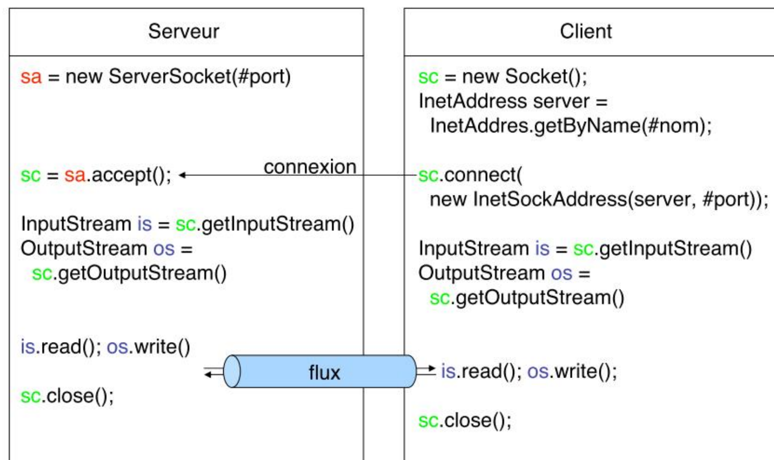
- **le serveur enregistre son service** sous un numéro de port, indiquant le nombre de clients qu'il accepte de faire buffériser à un instant T (`new serverSocket(...)`)
- **le serveur se met en attente d'une connexion** (méthode `accept()` de son instance de `ServerSocket`)
- **le client** peut alors **établir une connexion** en demandant la **création** d'une socket (`new Socket()`) à destination du serveur pour le port sur lequel le service a été enregistré.
- `le serveur` **sort** de son `accept()` et **récupère** une Socket de communication avec le client
- le client et le serveur peuvent alors utiliser des `InputSteam` et `OutputStream` pour échanger les données

Socket en mode flux

Liaison par flux : Socket/ServerSocket (TCP)

- **Connecté** : protocole de prise de connexion (lent), communication uniquement point à point
- **Sans perte** : un message arrive au moins un fois
- **Sans duplication** : un message arrive au plus une fois
- **Avec fragmentation** : les messages sont coupés
- **Ordre respecté** : Communication de type téléphone

Socket en mode flux



TCP/IP : le serveur (java.net.ServerSocket)

- il utilise la classe `java.net.ServerSocket` pour accepter des connexions de clients
- quand un client se connecte à un port sur lequel un `ServerSocket` écoute, `ServerSocket` crée une nouvelle instance de la classe `Socket` pour supporter les communications côté serveur

```
int port =...;
ServerSocket server = new ServerSocket(port);
Socket connection = server.accept();
```

TCP/IP : le serveur (java.net.ServerSocket)

- les constructeurs et la plus part des méthodes peuvent générer une IOException
- la méthode accept() est dite bloquante ce qui implique de la mettre dans une boucle infinie qui se termine seulement si une erreur grave se produit

```
final int PORT = ...;
try {
ServerSocket serveur = new ServerSocket(PORT,5);
while (true) {
Socket socket = serveur.accept();
}
}
 catch (IOException e){
  ... }
```

TCP/IP : le serveur (java.net.ServerSocket)

- Constructeur : n port

```
ServerSocket s = new ServerSocket(8080);
```

- Méthodes principales
  - **adresse IP** : `InetAddress getInetAddress()`
  - **port** : `int getLocalPort()`
  - **attente de connexion** : `Socket accept()` méthode bloquante
  - **fermeture** : `void close()`
  - **Options TCP** : `timeOut, receiveBufferSize`

TCP/IP : le client (java.net.Socket)

- le client se connecte au serveur en créant une instance de la classe `java.net.Socket` : connexion synchrone

```
String host = ...;
int port = ...;
Socket connection = new Socket (host,port);
```

- la socket permet de supporter les communications côté client
- la méthode `close()` ferme (détruit) le socket
- les constructeurs et la plupart des méthodes peuvent générer une `IOException`
- le serveur doit être démarré avant le client. Dans le cas contraire, si le client se connecte à un serveur inexistant, une exception sera levée après un *time-out*

TCP/IP : le client (java.net.Socket)

```
final String HOST = "...";
final int PORT = ...;
try {
  Socket socket = new Socket (HOST,PORT);
}
finally {
    try {socket.close();} catch (IOException e){}
 }
```

TCP/IP : le client (java.net.Socket)

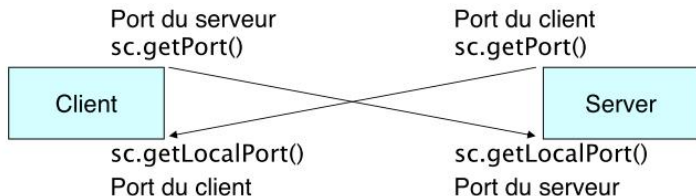
- Constructeur : adresse + nř port

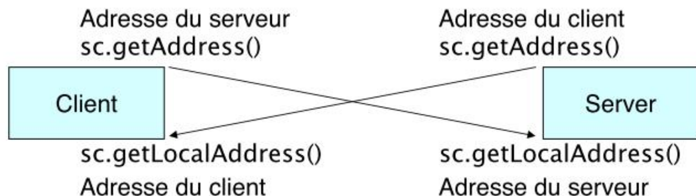
```
Socket s = new Socket("www.lifl.fr",80);
Socket s = new Socket(inetAddress,8080);
```

- Les méthodes principales
  - **adresse IP** : `InetAddress getInetAddress()`, `getLocalAddress()`
  - **port** : `int getPort()`, `getLocalPort()`
  - **flux in** : `InputStream getInputStream()`
  - **flux out** : `OutputStream getOutputStream()`
  - **fermeture** : `close()`
- Options TCP : `timeOut`, `soLinger`, `tcpNoDelay`, `keepAlive`

TCP/IP : le client (java.net.Socket)

Retrouver les adresses IP et les ports





TCP/IP : le flux de données

- une fois la connexion réalisée, il faut obtenir les streams d'E/S (`java.io`) auprès de l'instance de la classe `Socket` en cours
- **Flux entrant**
  - obtention d'un stream simple : définit les opérations de base
    `InputSteam in = socket.getInputStream();`
  - création d'un stream convertissant les bytes reçus en char
    `InputSteamReader reader = new InputStreamReader(in);`
  - création d'un stream de lecture avec tampon : pour lire ligne par ligne dans un stream de caractères
    `BufferedReader istream = new BufferedReader(reader);`
  - lecture d'une chaîne de caractères
    `String line = istream.readline();`

TCP/IP : le flux de données

- **Flux sortant**
  - obtention du flot de données sortantes : bytes

```
OutputSteam out = socket.getOutputStream();
```

  - création d'un stream convertissant les bytes en chaînes de caractères `PrintWriter ostream = new PrintWriter(out);`
  - envoi d'une ligne de caractères `ostream.println(str);`
  - envoi effectif sur le réseau des bytes (important) `ostream.flush();`

TCP/IP : le flux de données

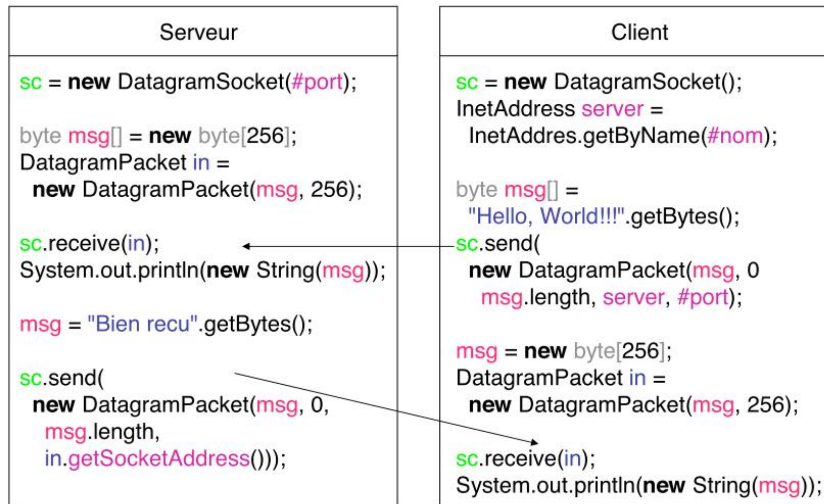
```
try {
 Socket socket = new Socket(HOST,PORT);
//Lecture du flux d'entrée en provenance du serveur
InputStreamReader reader = new
InputStreamReader(socket.getInputStream());
BufferedReader istream = new BufferedReader(reader);
String line = istream.readLine();
//Echo la ligne lue vers le serveur
PrintWriter ostream = new PrintWriter(socket.getOutputStream());
ostream.println(line);
ostream.flush();
}catch (IOException e) {...}
finally {try{socket.close();}catch (IOException e){}}
```

Socket en mode datagram

Liaison par datagram : DatagramSocket/DatagramPacket (UDP)

- **Non connecté** : pas de protocole de connexion (plus rapide)
- **Avec perte** : l'émetteur n'est pas assuré de la délivrance
- **Avec duplication** : un message peut arriver plus d'une fois
- **Sans fragmentation** : les messages envoyés ne sont jamais coupés : soit un message arrive entièrement, soit il n'arrive pas
- **Ordre non respecté** : Communication de type courrier

Socket en mode datagram



Socket en mode datagram

- Il faut utiliser les classes `DatagramPacket` et `DatagramSocket`
- Ces objets sont initialisés différemment selon qu'ils sont utilisés pour envoyer ou recevoir des paquets

Socket en mode datagram : envoi d'un datagram

1. créer un DatagramPacket en spécifiant :
    - les données à envoyer
    - leur longueur
    - la machine réceptrice et le port
2. utiliser la méthode `send(DatagramPacket)` de `DatagramSocket`
    - pas d'arguments pour le constructeur car toutes les informations se trouvent dans le paquet envoyé

Socket en mode datagram : envoi d'un datagram

```
//Machine destinataire
InetAddress address = InetAddress.getByName("␣rainbow.essi.fr␣");
 static final int PORT = 4562;
//Création du message à envoyer
String s = new String ("␣Message␣à␣envoyer");
int longueur = s.length();
byte[] message = new byte[longueur];
//Initialisation du paquet avec toutes les informations
DatagramPacket paquet = new DatagramPacket(message,longueur,
address,PORT);
//Création du socket et envoi du paquet
DatagramSocket socket = new DatagramSocket();
socket.send(paquet);....
```

Socket en mode datagram : réception d'un datagram

1. créer un `DatagramSocket` qui écoute sur le port de la machine du destinataire
2. créer un `DatagramPacket` pour recevoir les paquets envoyés par le serveur : dimensionner le buffer assez grand
3. utiliser la méthode `receive()` de `DatagramPacket` : cette méthode est bloquante

Socket en mode datagram : réception d'un datagram

```
//Définir un buffer de réception
byte[] buffer = new byte[1024];
//On associe un paquet à un buffer vide pour la réception
DatagramPacket paquet =new
DatagramPacket(buffer,buffer.length());
//On crée un socket pour écouter sur le port
DatagramSocket socket = new DatagramSocket(PORT);
while (true) {
//attente de réception
socket.receive(paquet);
//affichage du paquet reçu
String s = new String(buffer,0,paquet.getLength());
System.out.println("Paquet reçu : "+ s);
}
```

Socket en mode datagram : java.net.DatagramSocket

- Constructeur

```
DatagramSocket(port)          socket UDP sur port
DatagramSocket()              socket UDP sur port qqconque
```

- envoi : send(DatagramPacket)
- réception : receive(DatagramPacket)
- Options UDP : timeOut,...
- remarque : possibilité de "connecter" une socket UDP à une (@IP,port)

```
connect(InetAddress,int)
```

pas de connection réelle, juste un contrôle pour restreindre les send/receive

Socket en mode datagram : java.net.DatagramPacket

- Constructeur

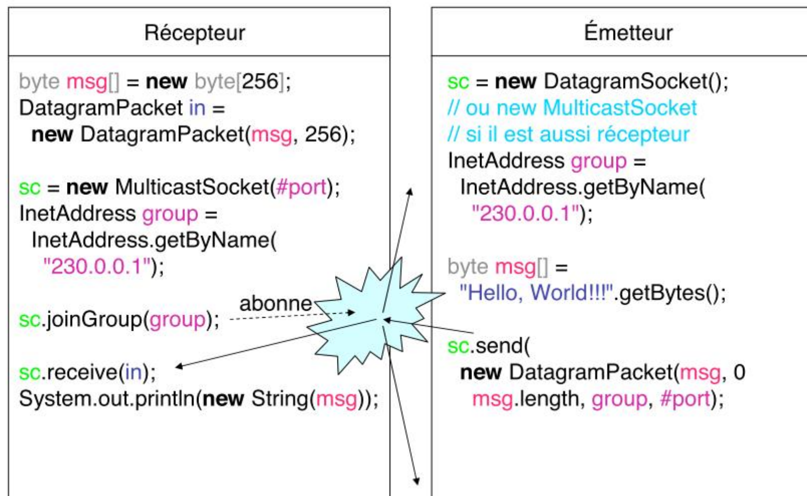
```
DatagramPacket( byte[] buf, int length )
DatagramPacket( byte[] buf, int length, InetAddress, port )
```

- getPort() : port de l'émetteur pour une réception ou port du récepteur pour une émission
- getAddress() : idem adresse
- getData() : les données reçues ou à envoyer
- getLength() : idem taille

Socket en mode multicast

- Muslticast IP : Diffusion de messages vers un groupe de destinataires
- messages émis sur une adresse
- messages reçus par tous les récepteurs "écoutant" sur cette adresse
- plusieurs émetteurs possibles vers la même adresse
- les récepteurs peuvent rejoindre/quitter le groupe à tout instant
- l'adresse IP de classe D (de 224.0.0.1 à 239.255.255.255) indépendante de la localisation des émetteurs/récepteurs
- Même propriétés que UDP : taille des messages limitée à 64 K, perte de messages possible, pas de contrôle de flux, ordre des messages non garanti,pas de connexion

Socket en mode multicast



Socket en mode multicast

- API : `java.net.MulticastSocket`
- Constructeur

```
MulticastSocket(port)          sur #port
MulticastSocket() sur port qqconque
( + ... )
```

- envoi : `send(DatagramPacket)`
- réception : `receive(DatagramPacket)`
- se lier à un groupe : `joinGroup(InetAddress)`
- quitter un groupe : `leaveGroup(InetAddress)`
- Limiter la portée des messages multicast : en fixant le *TTL* (setTimeToLive(int), le nombre de routeurs que le paquet peut traverser avant d'être arrêté
  - 0 : ne dépasse pas la machine
  - 1 : ne dépasse pas le réseau local
  - 127 : monde entier

Bibliographie

- APPRENTISSAGE DU LANGAGE JAVA, Serge Tahé - ISTIA - Université d'Angers, juin 2002
- basé sur les cours de : Gaël Thomas, Lionel Seinturier, Karima Boudaoud, etc.
- Kenneth L. Calvert, Michael J. Donahoo, TCP/IP Sockets in Java, pratical for programmers
- tutorial socket : `http://java.sun.com/docs/books/tutorial/networking/sockets/index.html`
- la paquetage `java.net` : `http://download.oracle.com/javase/1.5.0/docs/api/`