PROGRAMMATION RÉSEAU EN JAVA

*** Programmation réseau et client/serveur**

- communication => deux programmes qui s'échangent des données
  - donner des exemples
- peer to peer
- client-serveur
  - serveur, qui est à l'écoute
    - il écoute toutes les adresses
    - continue (se remet à l'écoute) même après avoir traité la demande
      - en général, il crée un processus/thread (il "fork") qui traite la demande et lui, en parallèle, se remet à l'écoute => il peut communiquer simultanément avec plusieurs clients
    - généralement, c'est lui qui transmet les données
      - contre-exemples : serveur FTP et on y dépose des fichiers
  - client, qui fait la demande auprès du serveur
    - la demande se fait à une certaine machine (exception : si adresse broadcast ou multicast)
    - finit après que la demande ait été traitée

*** Montrer l'exemple en UDP plus bas**

*** Perte de type lors du transfert**

- int a, b;
  a = b;
  => pas de problème
- sur le réseau, si on envoie a on ne sait plus de l'autre côté qu'il s'agit d'un entier !
  => c'est à nous (le protocole application) de gérer cela
  - par ex., on envoie et on reçoit toujours un entier, ensuite un float, ensuite une chaîne de caractères finissant par \0
- => on envoie et on reçoit toujours des byte[]

*** Conversion byte[] <-> String et autres**

- String -> byte[] :
  String s = "hello";
  byte[] b = s.getBytes ();
- byte[] -> String :
  String s = new String (byte[] data, int offset, int count);
  offset (indice à partir duquel on commence) est généralement 0
  new String (byte[] data) existe aussi, mais il convertit tout le tableau !!
- s'il y a d'autres types :

```
DataInputStream dis = new DataInputStream (new
                ByteArrayInputStream (packet.getData ())));
short v1 = dis.readShort ();
float v2 = dis.readFloat ();
char c = dis.readChar ();  // pour String aussi, avec une boucle for
```

- ASN.1 (Abstract Syntax Notation), network order : en java, pas de souci, le langage utilise les conventions réseau
  - ex. : int est stocké sur 4 octets, indépendamment de la machine
- int Integer.parseInt (String)
  String String.valueOf (int)

*** Adressage, classe InetAddress**

- sans constructeur
- création d'un objet, par méthodes statiques :
  getByName (String), le String est soit adresse IP, soit nom
- (Inet4Address, Inet6Address : spécifiques à IPv4 et IPv6)
- récupération du nom et de l'adresse IP d'un InetAddress : ...

*** Introduction aux transferts UDP et TCP**

- de quelles informations a-t-on besoin pour envoyer un paquet ?
  - données ?
  - longueur des données ?
  - protocole MAC ?

  - protocole réseau ?
  - protocole transport ?
  - protocole application ?
  - autres, analyser les différents champs des en-têtes
- on précise donc :
  - données
  - adresse IP destination
  - port destination
  - Q : pourquoi ne précise-t-on pas l'@ et le port source ?
  - Q : préciser en gros le contenu du paquet à envoyer, avec les différents en-têtes
- différences entre UDP et TCP :
  - UDP est en mode non connecté (les paquets sont indépendants)
  - TCP est en mode connecté (les paquets forment un tout)
  - UDP est orienté paquet (pour un grand volume de données, c'est à nous de le découper en paquets)
  - TCP est orienté flux (c'est TCP qui découpe les données en morceaux/paquets de la bonne taille)
  - en UDP les paquets peuvent être :
    - perdus
    - dupliqués
    - désordonnés (les paquets, pas les données, arrivent en désordre)
  - au contraire, TCP garantit :
    - fiabilité
    - non-duplication
    - ordonnancement

*** Transfert en mode non connecté (UDP)**

**** Exemple, étapes**

- analogie avec une conversation entre personnes
  - on s'appelle d'abord (décrocher), onsuite on communique, enfin on se sépare (raccrocher)
- (*) signifie que cette étape peut être répétée plusieurs fois
- serveur :
  - s'enregistre auprès du SE pour écouter sur un certain port (en Java : crée une socket)
  - (*) se met en attente de paquets
  - (*) quand un paquet lui arrive, fait le traitement nécessaire (en général, renvoi d'une réponse)
  - (*) éventuellement, envoi de paquet
  - avant la fin, libère le port
- client :
  - choisit un port (en Java : crée une socket)
  - (*) crée un paquet pour une certaine destination
  - (*) envoie le paquet
  - (*) éventuellement attendre des paquets
  - libère le port

**** Exemple en Java**

```
import java.net.*;
import java.io.*;

public class Serveur{
    public static void main (String args[]) throws IOException, SocketException{
        byte[] msg = new byte[1500];
        DatagramSocket s = new DatagramSocket (33000);  // le port sur lequel
                                                        // on ecoute
        DatagramPacket p = new DatagramPacket (msg, msg.length);  // l'objet
                        // ou le paquet est stocke
        s.receive (p);  // blocage jusqu'a reception d'un paquet
        s.close ();  // ferme la socket, libère le port
        System.out.println (new String (p.getData (), 0, p.getLength ()));
    }
}
```

```
----------

import java.net.*;
import java.io.*;

public class Client{
    public static void main (String args[])
                  throws IOException, SocketException, UnknownHostException{
        String m = "Hello world!";
        byte[] msg = m.getBytes ();
        DatagramSocket s = new DatagramSocket ();  // le port est choisi
                                                   // au hasard par le SE
        DatagramPacket p = new DatagramPacket (msg, m.length (),
                           InetAddress.getByName ("localhost"), 33000);
        s.send (p);
        s.close ();  // ferme la socket, libère le port
    }
}
```

**** DatagramSocket, discussion**

- s.receive (p) :
  - si aucun paquet n'existe dans la file de la couche UDP, se bloque jusqu'à réception d'un paquet
  - récupère le plus vieux paquet de la file UDP et le stocke dans p
    - attention, d'autres paquets peuvent se trouver dans la file UDP, en attente
- pour le client :
  - réservation d'un port aléatoire libre : new DatagramSocket ();
- la même socket peut être utilisée pour l'envoi et la réception
- réservation des ports :
  - http://www.iana.org/assignments/port-numbers
  - mieux vaut entre 33000 et 65000
  - sous linux, on ne peut pas se lier à un port <1024 si on n'est pas root
  - on ne peut pas se "lier" ("to bind") à un port plusieurs fois ! On reçoit BindException à l'exécution
  - fermeture du port recommandé (.close)

**** DatagramPacket, discussion**

- pour le récepteur des données :
  - automatiquement rempli avec les données reçues
  - longueur des données : p.getLength (), car on ne peut pas se fier au msg.length !
  - si msg.length < p.getLength (), la fin des données est perdue
- pour l'émetteur des données :
  - on met nous-mêmes les données à envoyer, l'adresse et le port destination
  - données : sans type, donc byte[] et longueur
- pour les deux :
  - p.getAddress () retourne l'InetAddress de l'autre extrémité

*** Transfert en mode connecté (TCP)**

**** Exemple, étapes**

- serveur :
  - s'enregistre auprès du SE pour écouter sur un certain port (en Java : crée une socket)
  - se met en attente de connexion
  - quand une connexion est créée, fait le traitement nécessaire (par ex. le renvoi d'un fichier)
  - ferme la socket
  - libère le port
- client :
  - choisit un port et se connecte à une certaine machine + port (en Java : crée une socket)

  - reçoit/envoie des données
  - ferme la socket et libère port

**** Exemple en Java**

```
import java.net.*;
import java.io.*;

public class Serveur{
    public static void main (String args[]) throws IOException{
        byte[] msg = new byte[3000];
        for (int i=0 ; i<msg.length ; i++)  // remplissage des donnees
            msg[i] = (byte)(i % 10);
        ServerSocket ss = new ServerSocket (40000);  // erreur si port utilise
        Socket s = ss.accept ();  // bloque jusqu'a l'arrivee d'une connexion
        OutputStream os = s.getOutputStream ();  // recupere le canal de transm
        os.write (msg);
        s.close ();  // fermeture de la connexion => TCP informe le peer
                     // de la fin des donnees
        ss.close ();  // liberation du port
    }
}

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

import java.net.*;
import java.io.*;

public class Client{
    public static void main (String args[])
                  throws IOException, UnknownHostException{
        int nb;  // nombre d'octets lus
        byte[] msg = new byte[1000];
        Socket s = new Socket ("localhost", 40000);
        InputStream is = s.getInputStream ();
        while ((nb = is.read (msg)) != -1)
            for (int j=0 ; j<nb ; j++)  // affiche les donnees
                System.out.print (msg[j]);
        s.close ();
    }
}
```

**** Communication, entrées/sorties, discussion**

- Socket = extrémité d'une connexion TCP
  - s.getInetAddress () retourne l'InetAddress de l'autre extrémité
- InputStream, OutputStream : flux binaires
- OutputStream :
  - write (byte[] b) écrit TOUS les bytes de la variable b
  - write (b, offset, length) écrit length bytes à partir d'offset (offset est par ex. 0)
- InputStream :
  - la méthode read bloque jusqu'à réception d'au moins 1 octet
  - read (byte[] b) remplit la variable b, sauf s'il y a moins de données
    - => voir son code de retour : -1 si EOF, ou bien le nombre octets lus
- socket.close () est obligatoire pour indiquer la fin des données à transmettre !!
- question ouverte : comment s'échanger des données de manière interactive sans couper la communication ?
  - exemple en HTTP : le client envoie une requête, ensuite attend une réponse, le serveur reçoit la requête, ensuite il envoie la réponse, le client reçoit la réponse et renvoie une requête, le serveur ...
- lire/écrire les types primitifs dans les flux binaires : les classes DataOutputStream et DataInputStream
  - création de l'objet : DataInputStream dis = new DataInputStream (is);

  - readInt (), writeInt (int) : lit, écrit un int
  - readDouble (), writeDouble (double) : lit, écrit un double
  - readUTF (), writeUTF (String) : lit, écrit un String
  - etc. : lancent EOFException si EOF

*** Transfert en utilisant un protocole au niveau application**

- HTTP et FTP fournis pour l'instant
  - d'autres peuvent être définis par le programmeur
- la classe URL :
  - crée la connexion TCP
  - crée l'en-tête de niveau application
  - envoie la requête

**** Exemple en Java**

```
import java.net.*;
import java.io.*;

public class ClientHTTP{
    public static void main (String args[]) throws IOException{
        URL u = new URL ("http://www.univ-fcomte.fr");
        DataInputStream br = new DataInputStream ((InputStream)u.getContent ());
        for (int i=0 ; i<10 ; i++)  // affiche 10 premieres lignes de la page
            System.out.println (br.readLine ());
    }
}
```