Chapitre IX : Introduction à la programmation réseau

Eric.Leclercq@u-bourgogne.fr

Département IEM
http://ufrsciencestech.u-bourgogne.fr
http://ludique.u-bourgogne.fr/~leclercq

24 mars 2011

Notion de socket

Définition :

Apparu à l'origine dans les systèmes UNIX, un socket est une abstraction logicielle qui réalise une interface logicielle avec les services réseau du système d'exploitation, grâce à laquelle un programme peut communiquer de manière uniforme avec d'autre programme situés sur des machines différentes.

- Ainsi les socket permettent l'échange de données entre 2 processus sur deux machines distinctes ;
- Chaque machine (programme) crée un socket ;
- Chaque socket est associé à un port (différent) ;
- Les deux socket devront éventuellement être connectés explicitement ;
- Les programmes lisent et écrivent dans les sockets.

Notion de port

- Un service réseau TCP/IP est accessible par un port ce qui permet d'aiguiller les données vers le bon processus dans le SE.
- Un port est identifié par un entier (16 bits).
- Les ports numérotés de 0 à 511 sont qualifié *well known ports* : ils donnent accès aux services standards (transfert de fichiers FTP port 21, terminal Telnet port 23, courrier SMTP port 25, serveur web port 80)
- De 512 à 1023, on trouve les services Unix.
- Au delà de 1024 ce sont les ports utilisateurs : disponibles pour une application quelconque.
- Un service est souvent connu par un nom logique : la correspondance entre nom et numéro de port est donnée par le fichier `/etc/services`

API Java

L'API (*Application Programming Interface*) `sockets` est une bibliothèque de classes de communication pour TCP/IP (`java.net`). Elle offre deux modes de communication :

- le mode connecté correspond au protocole TCP :
  - le protocole établit une connexion virtuelle
  - se charge alors de maintenir l'intégrité de la communication
  - gére les erreurs de transmission
- Le mode non connecté correspond au protocole UDP : l'envoi est fait au mieux (*best effort*). C'est à l'application de gérer la qualité de la transmission.

Les classes socket

Java propose 4 types de classes :

- Pour UDP :
  - `DatagramSocket()` pour un client
  - `DatagramSocket(ServerPort)` pour un serveur
- Pour TCP :
  - `Socket(ServerName,ServerPort)` pour un client
  - `ServerSocket(ServerPort)` pour un serveur
- Les sockets TCP doivent être associé à des flux

Classe Socket

- Constructeur : `Socket (String host, int port)` création du socket vers le port et la machine hôte spécifiés
- Méthodes :
  - `close()` : ferme le socket ;
  - `OutputStream getOutputStream()` : renvoie un flux de sortie pour le socket ;
  - `IntputStream getIutputStream()` : renvoie un flux de d'entrée pour le socket.

Classe `ServerSocket`

- Constructeur : `ServerSocket (int port)` creation du socket serveur sur le port spécifié
- Méthodes :
  - `close()` : ferme le socket ;
  - `OutputStream getOutputStream()` : renvoie un flux de sortie pour le socket ;
  - `IntputStream getIutputStream()` : renvoie un flux de d'entrée pour le socket ;
  - `Socket accept()` : écoute si une connexion est demandée pour ce socket et l'accepte.

La résolution de nom

- La mise en œuvre de la résolution des noms se fait à travers un service DNS ou un fichier local (`/etc/hosts`).
- Ce service est accessible via la classe `InetAddress`.
- Cette classe représente les adresses IP et un ensemble de méthodes pour les manipuler.
- les applications doivent utiliser les méthodes `getLocalHost`, `getByName`, ou `getAllByName` pour construire une nouvelle instance de `InetAddress`
- `public class InetAddress implements Serializable`
  - `InetAddress getByName(String host)` : construit un nouvel objet `InetAddress` à partir d'un nom textuel sous forme symbolique ou sous forme numérique
  - `String getHostName()` : obtient le nom complet correspondant à l'adresse IP
  - `String getHostAddress()` :obtient l'adresse IP sous forme numérique
  - `byte[] getAddress()` :obtient l'adresse IP sous forme d'un tableau d'octets

Exemple

```
1  import java.net.*;
2    public class WhoAmI {
3      public static void main(String[] args) throws Exception {
4    if(args.length != 1) {
5      System.err.println("Usage:␣WhoAmI␣MachineName");
6      System.exit(1);
7  }
8    InetAddress a = InetAddress.getByName(args[0]);
9    System.out.println(a);
10 }
```

Clients multiples

```
1  public class ServeurConcurrent {
2   final static int port = 1314;
3    public static void main(String[] args)
4      throws IOException {
5      ServerSocket serveur = new ServerSocket(port);
6      while (true) {
7        Connexion connexion = new Connexion(serveur.accept());
8        connexion.start();}
9   }
10 }
11 class Connexion extends Thread {
12   Socket connexion;
13   Connexion (Socket s) {
14     connexion = s;}
15   public void run() {
16     String requete;
17     try {PrintWriter out =
18   new PrintWriter(connexion.getOutputStream());
19       BufferedReader in =new BufferedReader(new InputStreamReader(
20             connexion.getInputStream()));
21       out.println("=>");
22       while (!(requete = in.readLine()).equals("FIN")) {
23   out.println(requete + "␣depuis␣" + connexion.getInetAddress() + ":" +
         connexion.getPort());
24         out.flush();
25       }
26     connexion.close();
27     }
28     catch (IOException e) {System.err.println(e);}
29   }
30 }
```

Datagrammes

- le protocole UDP est beaucoup plus simple que TCP :
  - ne permet pas de reconstituer l'ordre d'envoi des messages ;
  - donc plus efficace en bande passante ;
  - mais moins fiable puisqu'il n'est pas doté d'accusé de réception (automatique)
- les données sont placées dans un datagramme UDP, muni d'un en-tête comportant les numéros de port d'origine et de destination, la taille du datagramme et une somme de contrôle ;
- ce datagramme est lui-même placé dans un datagramme IP (ou paquet IP), muni d'un en-tête comportant entre autre les adresses IP d'émission et de réception ;
- la taille des données est limitée à 65 507 octets ;
- implémentés en Java par la classe `DatagramPacket` (les données sont contenues dans un tableau d'octet).

Datagrammes

Côté Client :

```
1  String s = "DATADATADTATA";
2  byte[] donnees = s.getBytes();
3  InetAddress adresse = InetAddress.getByName("le␣nom␣de␣l'hote");
4  int port = 5555;
5  DatagramPacket paquet =
6    new DatagramPacket(donnees, donnees.length, adresse, port);
```

En réception côté serveur : il faut construire un datagramme càd tableau d'octets qui recevra les données

```
1  byte[] donnees = new byte[4000];;
2  DatagramPacket d =
3    new DatagramPacket(donnees, donnees.length);
```

Socket UDP

- Les datagrammes sont émis et reçus par l'intermédiaire d'un objet de la classe `DatagramSocket`
- Côté client : on utilise un port pour construire le `DatagramSocket`, puis la méthode `send`

```
1 DatagramSocket client = new DatagramSocket();
2 client.send(paquet);
```

- Côté serveur : le port doit être passé en argument au constructeur `DatagramSocket(int)`, et la méthode `receive` remplit le tableau d'octets avec les données du paquet reçu

```
1 DatagramSocket serveur = new DatagramSocket(port);
2 serveur.receive(paquet);
```

Serveur ECHO UDP

```
1  import java.io.*;
2  import java.net.*;
3  class ServeurEchoUDP {
4    public static void main(String[] args)
5      throws UnknownHostException, IOException {
6      final int port = 8080;
7      DatagramPacket paquetRequete, paquetReponse;
8      DatagramSocket serveur = new DatagramSocket(port);
9      byte[] donneesRequete = new byte[4000];
10     while (true) {
11       paquetRequete = new DatagramPacket(donneesRequete, donneesRequete.length);
12       serveur.receive(paquetRequete);
13       paquetReponse =
14   new DatagramPacket(paquetRequete.getData(),
15           paquetRequete.getLength(),
16           paquetRequete.getAddress(),
17           paquetRequete.getPort());
18       serveur.send(paquetReponse);
19     }
20   }
21 }
```

Client ECHO UDP

```
1  class ClientEchoUDP {
2    public static void main(String[] args)
3      throws UnknownHostException, IOException {
4      String nomHote = "localhost";
5      InetAddress adresse = InetAddress.getByName(nomHote);
6      final int port = 8080;
7      String requete, reponse;
8      DatagramPacket paquetRequete, paquetReponse;
9      DatagramSocket client = new DatagramSocket();
10     BufferedReader entree =  new BufferedReader(new InputStreamReader(System.in)
           );
11     byte[] donneesReponse = new byte[4000];
12     while (!(requete = entree.readLine()).equals("END")) {
13       donneesRequete = requete.getBytes();
14       paquetRequete =new DatagramPacket(donneesRequete,
15   donneesRequete.length, adresse, port);
16       paquetReponse = new DatagramPacket(donneesReponse, donneesReponse.length);
17       client.send(paquetRequete);
18       client.receive(paquetReponse);
19       reponse = new String(paquetReponse.getData());
20       System.out.println(paquetReponse.getAddress() + "␣:␣" + reponse);
21     }
22     client.close();
23   }
24 }
```

Options des sockets

Il est possible de définir le comportement des sockets avec les options via une méthode set suivie du nom de l'option (grand nombre) :

- `SO_TIMEOUT` (minuterie sur les méthodes bloquantes) : à tester avec Interrupted Exception ;
- `TCP_NODELAY` : envoi de paquet très rapidement (attention aux impacts en terme de performance dans le cas de petits paquets) ;
- `SO_LINGER` : gestion des données non transmises ;
- `TCP_KEEPALIVE` : gestion de l'inactivité.

Protocoles d'objets sérialisés

- Définir les classes d'objets sérialisables :

```
1  // cette définition doit être acessible au client et au serveur
2  class Objet implements Serializable {
3     public String nom;
4     public Objet(String n) {
5        nom = new String(n);
6     }
7     public String toString() {
8        return "Objet : nom";
9     }
10 }
```

- Côté client :

```
1  public class Client {
2     static final int port = 8080;
3     public static void main(String[] args) throws Exception {
4           Socket socket = new Socket(args[0], port);
5           System.out.println("SOCKET = " + socket);
6           ObjectOutputStream oss = new ObjectOutputStream(socket.
                 getOutputStream());
7           oss.writeObject(new Objet("mon objet"));
8           System.out.println("END");
9     // attention aux commandes de type protocole
10          oss.writeObject("END") ;
11          oss.close();
12          socket.close();
13    }
14 }
```

Protocoles ad-hoc versus objets sérialisés

- Côté serveur :

```
1  public class Serveur{
2     static final int port = 8080;
3     public static void main(String[] args) throws Exception {
4          ServerSocket s = new ServerSocket(port);
5          Socket soc = s.accept();
6          ObjectInputStream ois =  new ObjectInputStream(
7        soc.getInputStream());
8          while (true) {
9             Object o = ois.readObject();
10         if (o.equals("END")) break;
11         System.out.println(o);   // tester l'objet
12         }
13         ois.close();
14         soc.close();
15    }
16 }
```

- L'utilisation de l'objet nécessite de connaître le type de l'objet reçu et de transtyper
- Comment transtyper dynamiquement ?