Les sockets avec Java

Auteur: Zouheir HAMROUNI

1 Les sockets en Java

Les sockets servent à communiquer entre deux hôtes à l'aide d'une adresse IP et d'un port. Elles permettent de gérer des flux entrants et sortants en deux modes :

- en mode connecté (TCP), assurant une communication fiable mais coûteuse en termes de ressources
- en mode non connecté (UDP) plus rapide, mais nécessitant de gérer soit même les erreurs de transmission.

Les sockets permettent de mettre en oeuvre des applications de type client/serveur :

- Un serveur est un programme (/machine) offrant un service sur un réseau
- Un client est un programme qui demande un service en émettant des requêtes

Les sockets sont utilisées dans différents types d'applications (jeux en lignes, applications distribuées, services de messagerie, gestion de fichiers à distance, etc) ; et sont supportés par différents lanaguages (C, C++, PHP, Java, etc).

Les exemples ci-dessous illustrent le fonctionnement des sockets en mode client/serveur en Java.

1.1 La classe InetAddress

Le "package" java.net fournit une classe `InetAddress` qui permet de récupérer et manipuler l'adresse internet d'un hôte sous différentes formes. En voici les principales méthodes :

- getLocalHost() : retourne un objet qui contient les références internet de la machine locale.
- getByName(String nom_machine) : retourne un objet qui contient les références internet de la machine dont le nom est passé en paramètre

De cet objet, on peut extraire différentes informations en appliquer les méthodes :

- getHostName() : retourne le nom de la machine dont l'adresse est stockée dans l'objet.
- getAddress() : retourne l'adresse IP stockée dans l'objet
- toString() : retourne un String contenant le nom de la machine et son adresse.

Le code suivant en donne une illustration.

```
1  import java.net.InetAddress ;
2  import java.net.UnknownHostException ;
3
4  public class AdressageInet {
5
6    public static void main(String[] args) {
7       InetAddress adresse ;
8       try {
9          adresse = InetAddress.getLocalHost() ;
10         System.out.println("L'adresse locale est : " + adresse ) ;
11         System.out.println("Le nom local est : " + adresse.getHostName() ) ;
12         adresse= InetAddress.getByName("www.enseeiht.fr") ;
13         System.out.println("L'adresse du site n7 est : " + adresse) ;
14      } catch (UnknownHostException e) {
15         e.printStackTrace() ;
16      }
17   }
18 }
```

Et affiche les informations suivantes :

```
$ L'adresse locale est : luke/147.127.133.193
$ Le nom local est : luke
$ L'adresse du site n7 : www.enseeiht.fr/193.48.203.34
```

1.2 Etablissement d'une connexion

Un socket est un point de terminaison d'une communication bidirectionnelle entre deux machines : un client et un serveur.

L'application Serveur est liée à un numéro de port, par exemple 8080, sur lequel elle se met à l'écoute des demandes de connexion des clients, via un `ServerSocket` (package java.net).

```
ServerSocket listenSock = new ServerSocket(numero_port);
```

Après ouverture du socket d'écoute, le serveur se met en état attente des demandes de connexion via la méthode accept(), et reste bloqué dedans jusqu'à réception d'une demande de connexion de la part d'un client :

```
Socket comSock = listenSock.accept();
```

Côté client, la demande connexion est effectuée via un `Socket`

```
Socket comSock = new Socket(adresse_serveur, numero_port_serveur);
```

Lorsque la demande de connexion du client atteint le serveur, celui-ci crée son socket de communication et sort de l'appel bloquant listenSock.accept(). A partir de cet instant, les échanges entre le client et le serveur peuvent commencer.

Voici ci-dessous un exemple de serveur :

```
1  import java.io.* ;
2  import java.net.* ;
3
4  public class Server1 {
5
6    static int port = 8080 ;
7    static int numc ;
8
9    public static void main(String[] args) {
10
11      ServerSocket listenSock ; // pour écouter les demandes de connection
12      Socket comSock ;
13      numClient = 1 ;
14
15      try {
16         // ouverture d'un socket d'écoute sur le port 8080
17         listenSock = new ServerSocket(port) ;
18         while (numClient < 100) {
19            comSock = listenSock.accept() ; // attente d'une requête de connexion
20            System.out.println("Une nouvelle connexion : " + numClient) ;
21            System.out.println(comSock) ;
22            // Echanges avec le client via comSock
23            // ....
24            comSock.close() ;
25            numClient++ ;
26         }
27         listenSock.close() ;
28      } catch (IOException e) {
29         e.printStackTrace() ;
30      }
31   }
32 }
```

Et un client.

```
1  import java.io.* ;
2  import java.net.* ;
3
4  public class Client1 {
5
6    static final int port = 8080 ;   //port de connection du serveur
7
8    public static void main(String[] args) {
9
10      Socket clientSock ;          //socket de communication
11
12      try {
13         // demande d'ouverture d'un socket sur la machine locale et le port
14         clientSock = new Socket(InetAddress.getLocalHost(), port) ;
15
16         // Echanges avec le serveur via clientSock
17         // ....
18         clientSock.close() ;      // fermeture du socket
19         System.out.println("Fin") ;   // message de terminaison
20      } catch (UnknownHostException e) {
21         e.printStackTrace() ;
22      } catch (IOException e) {
23         e.printStackTrace() ;
24      }
25   }
26 }
```

Il est important de capter les exceptions qui peuvent être levées par les différents appels. Par la suite, et pour alléger la présentation, le code de traitement des exceptions ne sera pas mis.

1.3 Communication

Une fois la connexion établie, le client et le serveur peuvent communiquer en gérant les flux entrants et les flux sortants via un `InputStream` et un `OutputStream`, et les méthodes associées getInputStream() et getOutputStream. La gestion de ces échanges peut se faire dans différents modes :

En mode byte

La communication se fait en mode "byte" : l'écriture (envoi) et la lecture (réception) se font via un tableau de "byte", respectivement sur un OutputStream et un InputStream. L'exemple suivant illustre ce mode de communication.

Un client qui envoie 3 fois le message "bonjour", avec une temporisation de 3 secondes entre les messages (code à ajouter dans le fichier du client).

```
1     static private void combasicStream(Socket cs) {
2        try {
3           OutputStream outStream = cs.getOutputStream() ;
4           String str = "bonjour" ;
5           byte[] b = str.getBytes() ;
6           int len = str.length() ;
7           for (int i = 0 ; i < 3 ; i++) {
8              outStream.write(b, 0, len) ;      // envoi du message
9              Thread.sleep(3000) ;
10          }
11          outStream.close() ;
12       }
13       catch ...
14    }
```

Le serveur affiche le numéro du client et les messages reçus

```
1     static private void combasicStream(Socket cs, int numc) {
2        try {
3           InputStream rdStream = cs.getInputStream() ;
4           byte[] buf = new byte[512] ;
5           int n = 0 ;
6           while ((n = rdStream.read(buf)) > 0) {
7              System.out.print("reçu du client "+ numc +" : "n+" octets : ") ;
8              System.out.write(buf, 0, n) ;
9              System.out.println() ;
10          }
11          System.out.println("FIN RECEPTION "+ n) ;
12          rdStream.close() ;
13       }
14       catch ...
15    }
```

Dans cet exemple, il faut remarquer que :

- ce mode de communication n'est pas bien adapté aux échanges de messages texte, et a nécessité des conversions pour passer aux bytes.
- lorsque le client ferme son socket, l'action de lecture sur le socket du serveur en mode stream renvoie la valeur -1, et le fait sortir de la boucle de lecture.

En mode texte

Pour l'envoi (écriture), la classe `PrintStream` ajoute à un flux la possibilité de faire des écriture, sous forme de texte, des types primitifs java, et des chaînes de caractères. Il s'agit d'une écriture bufférisée (l'envoi n'est effectué que lorsque le buffer est plein ou lorsqu'il est activé (flush) en ajoutant un "\n" ou en utilisant la méthode println.

Pour la réception (lecture) la classe `InputStreamReader` établit un pont entre les flux d'octets et les flux de caractères, mais ne permet pas à elle seule de lire des chaines.

Les classes `BufferedReader` et `LineNumberReader` permettent de lire dans un flux tamponné de caractères, dans un tampon de taille paramétrable (8192 par défaut).

On transforme le code précédent. Ce qui donne :

Côté Client :

```
1     static private void comText(Socket cs) {
2        try {
3           PrintStream wrBuffer = new PrintStream(cs.getOutputStream()) ;
4           String str = "bonjour" ;
5           for (int i = 0 ; i < 10 ; i++) {
6              wrBuffer.println(str) ;      // envoi du message
7              Thread.sleep(3000) ;
8           }
9           wrBuffer.println("END")  ;
10          wrBuffer.close() ;
11       }
12       catch ...
13    }
```

Il faut noter que si le serveur effectue une lecture sur le socket de communication alors que le client a fermé le sien, cette lecture engendre une "IOException". Pour éviter cela, le client informe le serveur de la fin des échanges en envoyant un message spécifique, "END" par exemple.

Côté serveur :

```
1  static private void comText(Socket cs, int numc) {
2     try {
3        BufferedReader rdBuffer = new BufferedReader(
4              new InputStreamReader(cs.getInputStream()) ;
5        while (true) {      // On peut mieux faire
6           String str = rdBuffer.readLine() ;   // lecture du message
7           if (str.equals("END")) break ;
8           System.out.println("RECU de : "+ numc +"= "+ str) ;
9        }
10       rdBuffer.close() ;
11    } catch ...
12 }
```

Avec ObjectInputStream et ObjectOutputStream

Les classes `ObjectInputStream` et `ObjectOutputStream` permettent l'échange d'objets sur des flux (lecture, écriture), via un mécanisme qui repose sur la sérialisation. Elles permettent d'échanger directement des String et des types natifs.

Le code dessous implémente les échanges en utilisant ces deux classes.

Côté Client :

```
1     static private void comObjectStream(Socket cs) {
2        try {
3           ObjectOutputStream oos = new ObjectOutputStream(cs.getOutputStream()
4           String str = "bonjour" ;
5           for (int i = 0 ; i < 3 ; i++) {
6              oos.writeObject(str) ;      // envoi du message
7              Thread.sleep(3000) ;
8           }
9           oos.writeObject("END")  ;
10          oos.close() ;
11       }
12       catch ...
13    }
```

Et côté serveur :

```
1     static private void comObjectStream(Socket cs, int numc) {
2        try {
3           ObjectInputStream ois = new ObjectInputStream(cs.getInputStream()) ;
4           while (true) {
5              String str = (String)ois.readObject() ;
6              if (str.equals("END")) break ;
7              System.out.println("RECU de : "+ numc +" = "+ str) ;
8           }
9           ois.close() ;
10       }
11       catch ...
12    }
```

1.4 Plusieurs clients en parallèle

L'exemple du serveur traité dessus ne permet de gérer qu'un seul clinet à la fois, car il est basé sur un seul thread qui s'occupe des échanges avec le client courant. Durant ce temps, les autres clients qui tentent de se connecter voient leur demande bloquée jusqu'à la fin de la communication avec le client courant.

Pour remédier à cela, il s'avère intéressant de faire travailler le serveur avec plusieurs threads :

- le thread initial se charge uniquement de l'écoute des demandes de connexion
- à chaque nouvelle connexion, un nouveau thread est lancé pour prendre en charge la communication avec le nouveau client.

Le code suivant permet d'implanter un serveur multi-threads.

```
1  public class ServerM extends Thread {
2
3     static int port = 8080 ;
4     static int numc = 0 ;
5     Socket comSock ;
6
7     public ServerM(Socket cs) {
8        comSock = cs ;
9     }
10
11    public void run() {
12       System.out.println("Une nouvelle connexion : " + numc) ;
13       comBuffer(comSock, numc) ;
14    }
15
16    public static void main(String[] args) {
17
18       ServerSocket listenSock ;
19       Socket comSock  ;
20
21       try {
22          listenSock = new ServerSocket(port) ;
23          while (numc < 100) {
24             Thread th = new ServerM(listenSock.accept()) ;
25             numc++ ;
26             th.start() ;
27          }
28          listenSock.close() ;
29       } catch ...
30    }
31 }
```