1. Introduction et définition

Les Socket sont des objets permettant la gestion de deux flux de données: un flux d'entrée (***InputStream***), garantissant la réception des données, et un flux de sortie (***OutputStream***), servant à envoyer les données. Un socket est un point de terminaison d'une communication bidirectionnelle, c'est-à-dire entre un client et un serveur en cours d'exécution sur un réseau donné. Les deux sont liés par un même numéro de port. En Java, nous distinguerons deux types de socket: ***les sockets simples*** (dits "clients") et ***les sockets serveurs***. Un socket client est tout simplement un socket qui va se connecter sur un socket serveur pour lui demander d'effectuer des tâches. Un serveur fonctionne sur une machine bien définie et est lié à un numéro de port spécifique. Le serveur se met à l'écoute d'un client, qui demande une connexion.

La communication par socket est un modèle permettant la communication inter processus afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau. On distingue deux modes de communication :

- **Le mode connecté** (comparable à une communication téléphonique), utilisant le protocole TCP. Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que l'adresse de destination n'est pas nécessaire à chaque envoi de données.
- **Le mode non connecté** (analogue à une communication par courrier), utilisant le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

2. Rôle des sockets:

- Connexion à une machine distante.
- Permettent l'échange de messages entre 2 processus, entre 2 machines.
- Envoi/Réception de données.
- Fermeture d'une connexion.
- Attachement à un port.
- Acceptation d'une demande de connexion à un port local.
- Attente de demandes de connexion.
- etc…

3. Principe de fonctionnement

- Le serveur crée un "socket serveur" (associée à un port) et se met en attente.
- Le client se connecte au socket serveur ; deux sockets sont alors crées : un " socket client", côté client, et un "socket service client" côté serveur. Ces sockets sont connectés entre elles.
- Le client et le serveur communiquent par les sockets. L'interface est celle des fichiers (read, write).
- Le socket serveur peut accepter de nouvelles connexions.
- Une fois terminée chaque machine ferme son socket.

4. Les sockets en JAVA

4.1. Etablissement de la connexion

Java.net comprend la classe **ServerSocket**, qui met en œuvre une sorte de prise que les serveurs peuvent utiliser pour écouter et accepter les connexions des clients. Pour créer un Socket type serveur on doit exécuter l'instruction suivante :

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

Ainsi on obtient un objet de la classe ***ServerSocket*** sur un port spécifique.

Il existe deux autres constructeurs, l'un à deux paramètres, le premier est bien sûr le numéro de port et le nombre total de connexions simultanées acceptées.

```
ServerSocket socketserver = new ServerSocket(numer_port,nbr_max);
```

nbr_max est le nombre maximal de connexions traitées simultanément. Par exemple au-delà de cinq tentatives de connexion consécutives autorisées, les connexions sont refusées.

Quant au client, celui-ci connaît le nom de la machine sur laquelle le serveur est en exécution et le numéro de port sur lequel il écoute. Le client va demander une connexion au serveur en s'identifiant avec son adresse IP ainsi que le numéro de port qui lui est lié.

Pour cela Java.net fournit une classe **Socket** qui met en œuvre un côté d'une connexion bidirectionnelle entre votre programme Java et un autre programme sur le réseau. La création d'un socket pour le client nécessite le constructeur suivant :

```
Socket socket = new Socket (param1, param2);
```

Le premier paramètre correspond à l'identité du serveur, il peut être une chaine de caractère ou de type **InetAddress**, param2 correspond au numéro de port sur lequel on souhaite se connecter sur le serveur.

Après tentative de connexion, si tout va bien, le serveur accepte la connexion du client, et reçoit un nouveau socket qui est directement lié au même port local. Il a besoin d'une nouvelle prise de sorte qu'elle puisse continuer à écouter le socket d'origine pour les demandes de connexion, tout t'en satisfaisant les besoins du client connecté. Voici comment accepter une connexion d'un client :

```
Socket connection = socketserver.accept();
```

4.2. Échange de message

Une fois la connexion établie, il est possible de récupérer le flux d'entrée et de sortie de la connexion TCP vers le serveur. Il existe deux méthodes pour permettre la récupération des flux :

- getInputStream() de la classe InputStream. Elle nous permet de gérer les flux entrant.
- getOutputStream() de la classe OuputStream. Elle nous permet de gérer les flux sortant.

Ces deux méthodes nous permettent donc de gérer les flux en entrée et en sortie. En général le type d'entrée et sortie utilisé est BufferedReader et InputStreamReader pour la lecture, PrintWriter pour l'écriture. Mais on peut utiliser tous les autres flux.

4.3. Exemple d'application

4.3.1. LE CLIENT

1. Création de la classe « client ».

```
import java.io.*;
import java.net.*;
public class Client{
        public static void main(String args[]) {
         try{  // La gestion des exceptions
```

2. Création de socket pour se connecter a un serveur.

Nous devons nous connecter à un serveur en utilisant TCP/IP. Pour résoudre ce problème, nous devons créer un Socket, en passant le nom du serveur et un numéro de port dans son constructeur. Pour créer un objet de la classe Socket, il est nécessaire d'utiliser le constructeur suivant : ***public Socket (String machine, int port).***

```
Socket SocketClient = new Socket("localhost", 2004);
```

La création d'un objet Socket entraîne la création d'un point de connexion (socket) et la connexion vers une autre socket (le serveur). L'adresse du serveur est composée de l'adresse d'une machine et d'un numéro de port. L'adresse locale et le port local peuvent également être spécifiés.

3. ***Affichage d'un message qui indique que la connexion est établie.***

```
System.out.println("Connected to localhost in port 2004");
```

Une fois connecté, nous aimerions transférer du texte. Pour résoudre ce problème, il faut construire un BufferedReader ou un PrintWriter depuis les getInputStream() ou getOutputStream() du socket. Afin de lire ce que le serveur envoie au client on utilise le socket.getInputStream() et afin d'envoyer du texte au serveur on utilise le socket.getOutputStream().
Une fois la connexion établie, il est possible de récupérer le flux d'entrée et le flux de sortie de la connexion TCP vers le serveur au moyen des méthodes : getInputStream () et getOutputStream ().

```
ObjectOutputStream out = new ObjectOutputStream(SocketClient.getOutputStream());
ObjectInputStream in= new ObjectInputStream(SocketClient.getInputStream());
```

Il est alors possible d'échanger des données avec le serveur au moyen de toutes les primitives de lecture et d'écriture des différentes classes du package java.io.

4. Communication avec le serveur.

Par défaut les primitives de lecture, tel que read(), sont bloquantes tant que rien n'est lisible sur le flux.

```
String message = "Message Client";
out.writeObject(message);
System.out.println("client>" + message);
message = (String)in.readObject();
System.out.println("server>" + message);
```

5. *Fermeture* de la connexion.

```
in.close(); out.close(); SocketClient.close(); }
catch(Exception e){ System.out.println("Exception !!! "+e.toString());} }}
```

4.3.2. LE SERVEUR.

Le programme Serveur.java est un simple serveur qui ne fait que d'afficher le texte reçu.

```
import java.io.*;
import java.net.*;
public class Serveur {
        public static void main(String args[]) {
                try{
```

1. Création d'un ServerSocket.

Cette classe est utilisée pour créer un socket du coté serveur. La classe **ServerSocket** permet de créer un point de communication, sur un port particulier, en attente de connexions en provenance de clients. Contrairement à la classe Socket elle n'ouvre pas de connexion.

```
ServerSocket SocketServeur = new ServerSocket(2004);
```

Une fois le socket créé, on attend les connexions de clients avec une méthode bloquante, le socket attend qu'une connexion se fasse puis l'accepte.

```
System.out.println("Waiting for connection");
```

La méthode accept() retourne un nouvel objet de la classe Socket qui est connecté avec un client particulier.

```
Socket connection = SocketServeur.accept();
System.out.println("Connection received and Accepted");
```

2. Les opérations get Input and Output streams.

```
ObjectOutputStream out = new ObjectOutputStream(connection.getOutputStream());
ObjectInputStream in  = new ObjectInputStream(connection.getInputStream());
```

3. Communication entre le client et le server via input & output streams.

```
String message = (String)in.readObject();
System.out.println("client>" + message);
message = "Message serveur";
out.writeObject(message);
System.out.println("server>" + message);
```

4. Fermeture de la connexion.

```
            out.close(); in.close(); SocketServeur.close(); }
catch(Exception e){ System.out.println("Exception !!! "+e.toString());}}}
```

TP RMI - Remote Method Invocation

L'objectif est de mettre en œuvre l'utilisation des RMI en langage Java, il s'agit de réaliser deux programmes (un client et un serveur) utilisant les RMI pour manipuler des objets sur une machine distante.

1. Introduction

RMI est une technologie développée et fournie par Sun. RMI un ensemble de classes permettant de manipuler des objets sur des machines distantes de manière similaire aux objets sur la machine locale. C'est un peu du "RPC orienté objet". Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant (figure 1). Cette machine virtuelle peut être sur une autre machine différente accessible par le réseau.

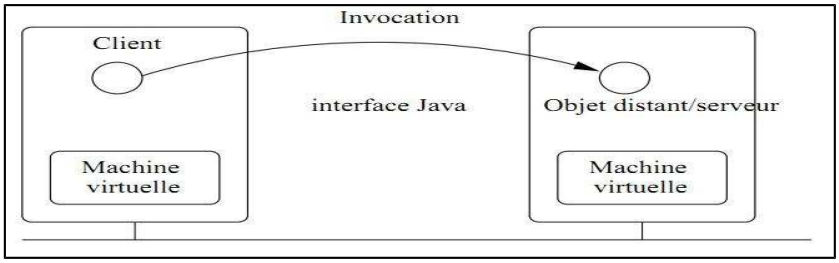

Figure 1. Principe des RMI.

2. Développement d'une application Java RMI

Le développement d'une application Java RMI passe par les étapes suivantes :

A. Définir une Interface distante.

Il s'agit dans cette étape de définir une interface distante qui sera implémentée par la suite au niveau du serveur RMI. Dans notre exemple l'interface contient deux méthodes : ***receiveMessage*** qui prend en entrée un Texte et l'affiche sur l'écran et la fonction ***calcul*** qui affiche la somme de deux entiers passés en entrée de la fonction.

```
import java.rmi.*;
public interface InterfaceRMI extends Remote
{  void receiveMessage(String x) throws RemoteException;
   int calcul(int a, int b) throws RemoteException; }
```

B. Créer une application Serveur.

La machine sur laquelle s'exécute la méthode distante est appelée serveur. A ce nivaux on va développer une classe qui implémente l'interface distante "**InterfaceRMI**" avec un constructeur et deux méthodes : ***receiveMessage*** et ***calcul***. Cette classe doit obligatoirement hériter de la classe ***UnicastRemoteObject*** qui contient les différents traitements élémentaires pour un objet distant.

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

class ServeurRMI extends java.rmi.server.UnicastRemoteObject implements InterfaceRMI {
String Address;
static int Port=1099;
```

1. Constructeur de la classe.

```
public ServeurRMI() throws RemoteException
 {System.out.println("Adresse Serveur = "+Address+",Port = "+Port);}
```

2. Méthodes implémentant l'interface distante.

```
   public void receiveMessage(String x) throws RemoteException
   {System.out.println("Invoked method : "+ x);}
```

```
public int  calcul(int a,int b) throws RemoteException
{int som=a+b;  System.out.println("Invoked method clacul: " + som); return som;}
```

3. Méthode principale : C'est la méthode main principale nécessaire pour lancer notre serveur RMI. Cette méthodes instanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de nom RMI (RMI Registry).

```
static public void main(String args[])
{ try{
      ServeurRMI s=new ServeurRMI();
      System.out.println("Hello Server is ready.");
```

La classe ***java.rmi.registry.LocateRegistry*** et sa méthode ***getRegistry***() est un niveau plus bas (elle est utilisée par Naming). Cette classe permet de créer dynamiquement un ***registry*** avec ***createRegistry***(). L'opération suivante consiste à enregistrer l'objet créé dans le registre du nom en lui affectant un nom. Ce nom sera utilisé par le client pour obtenir une référence sur l'objet distant. L'enregistrement se fait en utilisant la méthode ***rebind*** de la classe ***Naming***.

```
      Registry registry = LocateRegistry.createRegistry(Port);
      System.out.println("Created RMI registry on port "+Port);
      registry.rebind("rmiServer", s);
} catch (Exception e) {System.out.println(e.toString());System.exit(1);}
} }
```

C. Créer une application cliente qui invoque les méthodes du serveur à distance.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence. Le développement côté client se compose de :

1. L'obtention d'une référence sur l'objet distant à partir de son nom.
2. L'appel à la méthode à partir de cette référence.

```
import java.rmi.*;
import java.rmi.registry.*;
import java.net.*;
public class ClientRMI
{
   static public void main(String args[])
   {  InterfaceRMI rmiServer;
      Registry registry;
      String Address="127.0.0.1";
      String Port="1099";
      String text="Hello from client";
      System.out.println("sending from client>>>"+text+" to "+Address+":"+Port);
```

La méthode ***getRegistry()*** essaye d'obtenir le registre s'exécutant sur le port 1099.

```
      try{    registry=LocateRegistry.getRegistry("127.0.0.1",1099);
              System.out.println("RMI registry found on port"+Port);
```

Avant d'invoquer une méthode de l'interface, il faut obtenir une référence. Pour obtenir une référence sur l'objet distant à partir de son nom, il faut utiliser la méthode ***lookup***().

```
         rmiServer= (InterfaceRMI)(registry.lookup("rmiServer"));
         rmiServer.receiveMessage(text);
         int x = rmiServer.calcul(10, 15); }
      catch(Exception e){System.out.println(e.toString());}
   } }
```

(Socket Java - UDP)

```
import java.io.*;
import java.net.*;
class UDPServer
{
   public static void main(String args[]) throws Exception
```

```
        {
           DatagramSocket serverSocket = new DatagramSocket(9876);
              byte[] receiveData = new byte[10];
              byte[] sendData = new byte[10];
              while(true)
                 {
                    DatagramPacket receivePacket = new DatagramPacket(receiveData,
                    receiveData.length);
                    serverSocket.receive(receivePacket);
                    String sentence = new String( receivePacket.getData());
                    InetAddress IPAddress = receivePacket.getAddress();
                    int port = receivePacket.getPort();
                    System.out.println("RECEIVED: " + sentence+" - From Address:
                    "+IPAddress+" - Port: "+ port);
                    String capitalizedSentence = sentence.toUpperCase();
                    sendData = capitalizedSentence.getBytes();
                    DatagramPacket sendPacket = new DatagramPacket(sendData,
                    sendData.length, IPAddress, port);
                    serverSocket.send(sendPacket);
                 }
        }
}
```

```
import java.io.*;
import java.net.*;
import java.util.Scanner;
class UDPClient
{
   public static void main(String args[]) throws Exception
   {
      DatagramSocket clientSocket = new DatagramSocket();
      InetAddress IPAddress = InetAddress.getByName("localhost");
      byte[] sendData = new byte[10];
      byte[] receiveData = new byte[10];
      System.out.println("Sentence : ");
      Scanner sc = new Scanner(System.in);
      String sentence = sc.next();
      sendData = sentence.getBytes();
      DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
      IPAddress, 9876);
      clientSocket.send(sendPacket);
      DatagramPacket receivePacket = new DatagramPacket(receiveData,
      receiveData.length);
      clientSocket.receive(receivePacket);
      String modifiedSentence = new String(receivePacket.getData());
      System.out.println("ORIGINAL SENSTENCE:" + sentence);
      System.out.print("FROM SERVER (IP Server Address" +
      receivePacket.getAddress().toString()+" Port Server:
      "+receivePacket.getPort()+ ") - AFTER MODIFICATION : " + modifiedSentence);
      clientSocket.close();
   }
}
```

Référence :

- https://www.java.com/fr/
- http://www.oracle.com/
- https://openclassrooms.com/courses/apprenez-a-programmer-en-java/
- Développons en Java - Les EJB3: https://www.jmdoudoux.fr/java/dej/chap-ejb3.htm
- http://www.eclipsetotale.com/articles/Introduction_EJB3_avec_Eclipse.html
- http://www.oracle.com/technetwork/articles/java/ejb-3-1-175064.html
- J2EE / Java EE - Jean-Michel Doudoux : https://www.jmdoudoux.fr/java/dej/chap-j2ee-javaee.htm
- Développons en Java - JDBC (Java DataBase Connectivity) : https://www.jmdoudoux.fr/java/dej/chap-jdbc.htm
- http://www.objis.com/formation-java/tutoriel-java-acces-donnees-jdbc.html