Cours - Intergiciels - S7

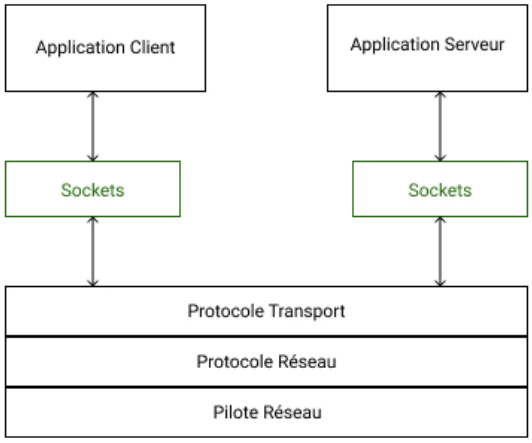
> Écrit par Clément Lizé

1 - Sockets

1.1 - Contexte

Sockets = `API` (Interface de programmation)

Point de communication → couche applicative



1.2 - Deux modes

Mode connecté : TCP

➡️ Le plus utilisé

✔️ Connexion interrompue → applications informées

👌 Très fiable

😞 Plus lent

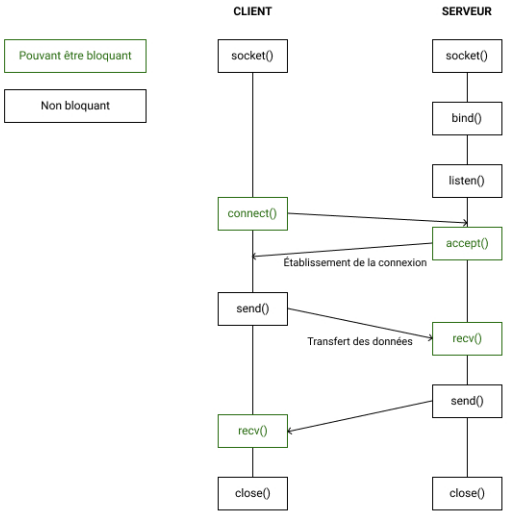
Mode non connecté : UDP

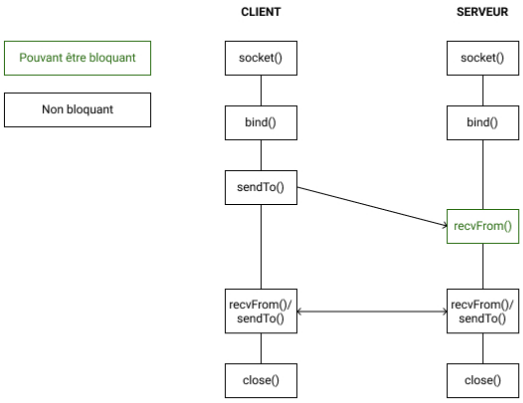
➡️ Moins utilisé que TCP

✔️ Données envoyées en paquets indépendants

👌 Plus rapide car pas de connexion

😞 Moins fiable



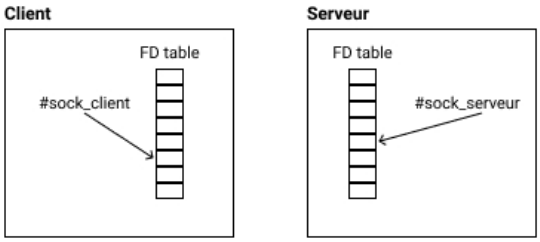


1.3 - Fonctions en C

1.3.1 - socket()

➡️ Créer un socket

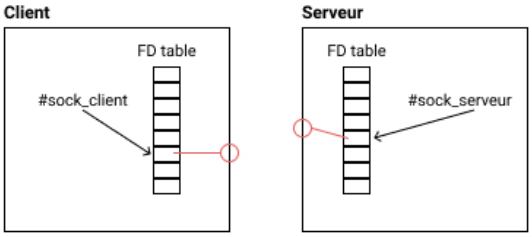
`int socket (int family, int type, int protocol)`



1.3.2 - bind()

➡️ Créer un lien entre un socket et un port machine

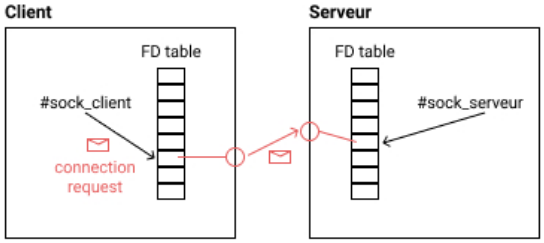
`int bind (int sock_desc, struct sockaddr *my_@, int lg_@)`



1.3.3 - connect()

➡️ Envoyer une demande de connexion au serveur (depuis le client)

`int connect (int sock_desc, struct sockaddr *my_@, int lg_@)`



1.3.4 - listen()

➡️ Utiliser le socket (côté serveur) pour recevoir des demandes de connexion

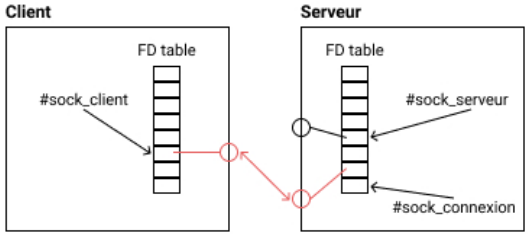
📄 `int connect (int sock_desc, int nbr)`

🚨 `nbr` = nombre de connexions qui peuvent être en attente

1.3.5 - accept()

➡️ Côté serveur. Bloque les demandes de connexion, crée un nouveau socket de communication puis ré-autorise les demandes de connexions. Le nouveau socket sert pour la communication entre le client et le serveur et sera détruit à la fin de celle-ci.

📄 `int accept (int sock_desc, struct sockaddr *client, int lg_@)`



1.4 - Les sockets en Java

1.4.1 - DNS

```
import java.net.*;

public class Enseeiht1 {

  public static void main (String[] args) {

    try {
      InetAddress address = InetAddress.getByName("www.enseeiht.fr");
      System.out.println(address);
      // OU
      InetAddress a = InetAddress.getLocalHost();
      System.out.println(a.getHostName() + " / " + a.getHostAddress());
    }
    catch (UnknownHostException e) {
      System.out.println("cannot find www.enseeiht.fr");
    }
  }
}
```

➡️ InetAddress = classe de java.net

➡️ getByName : transforme un nom de machine en adresse IP

➡️ getLocalHost : récupère l'adresse Inet de localhost

1.4.2 - Mode connecté TCP

a - Code client

```
class Client {
  try {
    Socket s = new Socket("www.enseeiht.fr",80);
    InputStream is = s.getInputStream();
    OutputStream os = s.getOutputStream();
  }
  catch (UnknownHostException u) {
    System.out.println("Unknown host");
  }
  catch (IOException e) {
    System.out.println("IO exception");
  }
}
```

➡️ Socket : objet Socket, initialisé avec une adresse (nom ou IP) et un port. Équivalent en C de *socket()* et *connect()*

➡️ InputStream : permet de lire des octets

➡️ OutputStream : permet d'écrire des octets

b - Code serveur

```
class Serveur {
  try {
    ServerSocket ss = new ServerSocket(80);
```

➡️ ServerSocket : Socket qui a pour objectif de recevoir des demandes de connexion. Équivalent en C de *socket()*, *bind()* et *listen()*

```
    Socket s = ss.accept();
    InputStream is = s.getInputStream();
    OutputStream os = s.getOutputStream();
  }
  catch (UnknownHostException u) {
    System.out.println("Unknown host");
  }
  catch (IOException e) {
    System.out.println("IO exception");
  }
}
```

➡️ accept : Retourne un socket de communication avec le client. Ce nouveau socket est indépendant de *ss*, ainsi le serveur peut à nouveau recevoir des demandes de connexion pendant qu'il échange avec le client sur le socket s.

➡️ InputStream : permet de lire des octets

➡️ OutputStream : permet d'écrire des octets

c - Fonctions utiles

➡️ `InputStreamReader` ou `OutputStreamReader` : convertit un stream d'octets en un stream de caractères

➡️ `BufferedReader` : implémente la notion de tampon : permet la lecture caractère par caractère, ligne par ligne, ..

➡️ `PrintWriter` : permet d'afficher (avec println par ex)

1.4.3 - Mode non connecté UDP

a - Code client

```
class Client {
  try {
    int p = 8888; // for receiving a response
    byte[] t = new byte[10];

    FileInputStream f = new FileInputStream("data.txt");
    int r = f.read(t);

    DatagramSocket s = new DatagramSocket(p);
    DatagramPacket d = new DatagramPacket(t, r,
      InetAddress.getByName("thor.enseeiht.fr"), 9999);
    s.send(d);
  }
  catch (Exception e) {
    System.err.println(e);
  }
}
```

➡️ DatagramSocket : permet de créer un socket associé à un port. Équivalent en C de *socket()* et bind*()*

➡️ DatagramPacket : un packet prêt à l'envoi, créé à partir d'un tableau d'octets. *r* est la taille des données qu'on lit du fichier.

➡️ send : envoi du packet

b - Code Serveur

```
class Serveur {
  try {
   int p = 9999;
   byte[] t = new byte[10];

   DatagramSocket s = new DatagramSocket(p);
   DatagramPacket d = new DatagramPacket(t,t.length);

   s.receive(d);

   String str = new String(d.getData(), 0, d.getLength());
   System.out.println(d.getAddress()+"/"+d.getPort()+"/"+str);

  }
  catch (Exception e) {
   e.printStackTrace();
  }
}
```

➡️ DatagramSocket : permet de créer un socket associé à un port. Équivalent en C de *socket()* et bind*()*

➡️ DatagramPacket : un packet prêt à l'envoi, créé à partir d'un tableau d'octets. *r* est la taille des données qu'on lit du fichier.

➡️ receive : lit sur le socket UDP et stocke les données dans le DatagramPacket

➡️ getData : renvoie un tableau d'octets depuis le DatagramPacket

➡️ getLength : retourne la taille des données reçues

➡️ getAddress et getPort : renvoient les adresse et port du client pour pouvoir envoyer une réponse

1.4.4 - Exemple complet

```java
public class Client {
  public static void main (String[] str) {
    try {
      Socket csock = new Socket("localhost",9999);
      ObjectOutputStream oos = new ObjectOutputStream (
        csock.getOutputStream());
      oos.writeObject(new Person("Dan","Hagi",53));
      csock.close();
    }
    catch (Exception e) {
      System.out.println("An error has occurred ...");
    }
}
```

```java
public class Server {
  public static void main (String[] str) {
    try {
      ServerSocket ss;
      int port = 9999;
      ss = new ServerSocket(port);
      System.out.println("Server ready ...");
      while (true) {
        Slave sl = new Slave(ss.accept());
        sl.start();
      }
    }
    catch (Exception e) {
      System.out.println("An error has occurred ...");
    }
  }
}
```

```java
public class Slave extends Thread {
  Socket ssock;

  public Slave(Socket s) {
    this.ssock = s;
  }

  public void run() {
    try {
      ObjectInputStream ois = new ObjectInputStream(
        ssock.getInputStream());
      Person v = (Person)ois.readObject();
      System.out.println("Received person: "+ v.toString());
      ssock.close();
    }
    catch (Exception e) {
      System.out.println("An error has occurred ...");
    }
  }
}
```

2 - RPC et Java-RMI

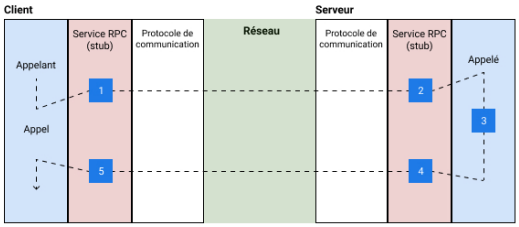
2.1 - Contexte

RPC = "Remote Procedure Call". Objectif = appels locaux et distants avec la même syntaxe

RMI = "Remote Method Invocation" = API qui implémente RPC en Java

2.2 - RPC

2.2.1 - Fonctionnement général



➡️ En bleu : segments de code avec le client qui appelle un service, et le serveur qui fournit le service

➡️ En rouge : segments de code créés par RPC : le stub client et le stub serveur

👉 Le stub client donne l'illusion que le service fourni par le serveur est local. Pour créer cette illusion, le client implémente les même procédures que le serveur

1️⃣ Le stub **collecte** les paramètres, **les assemble** dans un message et **arme un timer**
Un **identifiant** est généré pour l'appel RPC

2️⃣ Le protocole de transport **délivre** le message vers le service RPC (le stub serveur)
Le stub serveur **désassemble** les paramètres, l'id RPC est enregistré

3️⃣ **Le code est exécuté**

4️⃣ Les paramètres du résultat sont **assemblés** dans un message
Le stub serveur **arme un timer** et **transmet** le message au protocole de communication

5️⃣ Le protocole de transport **délivre** le message vers le service RPC (le stub client)
Le stub client **désassemble** les paramètres du résultat
Un **ACK** est envoyé au serveur, qui **désarme son timer**
Les **résultats sont transmis** à l'appelant

2.2.2 - Les rôles des stubs

Stub client

Sert d'interface avec le client

- Reçoit les appels locaux
- Les transforme en un appel distant
- Reçoit les résultats
- Retourne les résultats avec une procédure de retour normale

Stub serveur

Exécute sur le serveur

- Reçoit les appels sous forme de messages
- Exécute la procédure
- Reçoit les résultats de la procédure en local
- Transmets les résultats sous forme de message

2.3 - RMI

Objectif : rendre transparent la manipulation d'objets situés dans une autre JVM (souvent sur une autre machine)

2.3.1 - Service de nommage : RMI registry

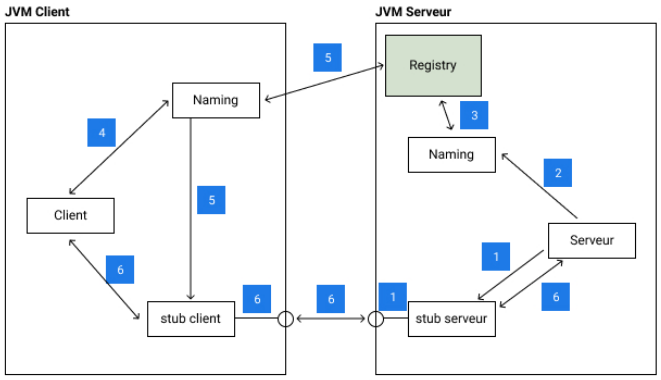
Registry = annuaire des services qui tournent sur le serveur distant

🚨 Contrainte : doit se situer sur la même machine que le serveur

➡️ Pour ajouter un service : *bind()*

➡️ Pour consulter un service : *lookup()*

2.3.2 - Fonctionnement général



1️⃣ Le serveur crée le stub serveur et le port est ouvert

2️⃣ Le serveur s'enregistre auprès du Naming (méthode *bind()* ou *rebind())*

3️⃣ Le Naming enregistre dans le registry le nom de l'objet serveur, son stub et les coordonnées de connexion (IP et port)

4️⃣ Le client fait appel à son Naming pour localiser le serveur (méthode *lookup()*)

5️⃣ Le naming récupère les coordonnées du serveur et crée le stub client

6️⃣ Le client appelle les méthodes du serveur. Ces requêtes passent par le stub client qui crée un port de communication côté client

2.3.3 - En pratique

```
import java.rmi.*;

public interface monServeur extends Remote {

  public int maMethode() throws RemoteException;

}
```

➡️ Un serveur doit avoir une interface qui hérite de la classe *Remote*.

➡️ Seules les méthodes décrites dans l'interface seront accessibles à distance.

➡️ Toutes les méthodes doivent propager une *RemoteException*.

➡️ Les types de retour peuvent être seulement des :

- Types primitifs (int, boolean, ..)
- Objets distants
- Objets sérialisables (voir plus bas)

🚨 En RMI, tous les arguments sont transmis par copie → un objet n'est pas modifié localement si il a été passé en paramètre d'une méthode distante

```
import java.rmi.server.*;
import java.rmi.*;

public class monServeurImpl extends UnicastRemoteObject implements monServeur {

  public monServeurImpl() throws RemoteException {
    System.out.println("Serveur créé");
  }

  public int maMethode() throws RemoteException {
    return 0;
  }
}
```

➡️ Les serveurs RMI doivent hériter de la classe *UnicastRemoteObject*.

➡️ Toutes les méthodes doivent propager une *RemoteException*, y compris le constructeur.

```
public void main(String[] args) {
  try {
    Naming.bind("//localhost/monService", new monServeurImpl());
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

Ce code doit être exécuté sur la machine serveur. Il permet de déclarer le serveur auprès du Naming (et donc du registry) pour pouvoir l'utiliser avec le client.

➡️ *bind()* permet de déclarer le serveur au naming

➡️ Par défaut, le port du registry est à 1099

```
import java.rmi.*;

public class Client {

  public static void main(String[] args) {

    try {

      monServeur s = (monServeur) Naming.lookup("//localhost/monService");
      System.out.println(s.maMethode());  // 0
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

➡️ La méthode *lookup()* renvoie un stub qui doit être converti en l'interface du serveur. ⚠️ Ne doit pas être convertie en serveur.

➡️ l'appel des méthodes distantes est maintenant aussi simple qu'en local

2.3.4 - Sérialisation

Les objets qui transitent doivent être sérialisables. Quasiment toutes les classes de base le sont, donc pas besoin de s'occuper de ça quand on renvoie un int par exemple. Si on renvoie un objet qu'on a créé, il faut le rendre sérialisable :

```java
import java.io.*;

public interface SFiche extends Serializable {

  public String getNom();
}
```

```java
import java.rmi.*;

public class SFicheImpl implements SFiche {

  private String nom;

  public SFicheImpl (String nom) {
    this.nom = nom;
  }

  @Override
  public String getNom() {
    return this.nom;
  }
}
```

3 - Web Services

Flemme de résumer, voir cours

4 - JRM

Flemme de résumer, voir cours