Les communications réseaux

Rappels : les protocoles d'Internet

Le **modèle TCP/IP** utilisé comporte 4 couches :

Couches	Protocoles
Application	HTTP, FTP, DNS, DHCP, SMTP, …
Transport	TCP, UDP, …
Internet	IP, …
Hôte-Réseau	Ethernet, …

TCP (**T**ransport **C**ontrol **P**rotocol) est

- **Orienté connexion** (une connexion doit être établie avant l'échange des données)
- **Fiable** (les paquets de données envoyés sont assurés d'arriver et dans le même ordre que celui dans lequel ils ont été émis)
- Assure un transfert de données par **« flot »** (un peu comme les pipes)

UDP (**U**ser **D**atagram **P**rotocol) est

- **Non orienté connexion** (pas d'établissement de connexion avant l'échange des données)
- **Non Fiable** (certaines données peuvent ne pas arriver à destination et l'ordre d'arrivée des paquets de données peut ne pas être respecté)
- Assure un transfert de données par **« paquets »** appelés datagrammes

Adresses IP, ports et sockets

- Une machine est identifiée logiquement sur le réseau à l'aide d'une **adresse IP** (codée sur 32 bits) → Exemple : 10.43.246.115
- Plusieurs processus peuvent utiliser la même adresse IP → pour diriger les paquets de données vers les bonnes applications, on utilise un **numéro de port** (codé sur 16 bits) différent par « application »

Un **numéro de port** est un nombre positif compris entre 0 et 65535 dont les valeurs de 0 à 1023 sont réservées par des protocoles bien connus :

- HTTP → port 80
- FTP → ports 20 et 21
- SSH → port 22
- Telnet → port 23
- SMTP → 25
- …

En **mode connecté (TCP)**,

Une connexion établie associe une adresse IP et un port → les deux ensembles constituent ce que l'on appelle une **socket**, ou un **point de connexion**

En **mode datagramme (UDP)**,

Les paquets de données sont accompagnés des adresses et numéros de port des destinataires

Adresses IP et Java

Java représente une adresse IP par la classe **InetAddress** du package java.net qui ne possède pas de constructeur mais les méthodes factory suivantes :

- **public static synchronized InetAddress getLocalHost()** → permet d'obtenir l'adresse IP de la machine locale
- **public static synchronized InetAdress getByName(String nom)** → permet d'obtenir l'adresse IP d'une machine dont on connaît le nom

Une fois l'objet **InetAddress** obtenu, on peut en connaître le contenu à l'aide des méthodes

- **public String getHostName()** → retourne le nom de la machine associée

- **public String getHostAddress()** → retourne l'adresse IP de la machine associée sous la forme d'une chaîne de caractères
- **public byte[] getAddress()** → retourne l'adresse IP de la machine associée sous la forme d'un tableau de 4 bytes exprimés dans l'ordre du réseau
- **public boolean isMulticastAddress()** → retourne vrai s'il s'agit d'une adresse IP de type D

Exemple d'utilisation de la classe InetAddress

Dans l'exemple qui suit, le programme affiche l'adresse IP de la machine sur laquelle il est lancé (« **moon** » d'adresse IP **192.168.228.167**) et celle du site www.google.be

Le code du programme de test (fichier **TestInetAddress.java**) est

```
import java.net.*;

public class TestInetAddress
{
    public static void main(String args[]) throws UnknownHostException
    {
        InetAddress ipLocale, ipGoogle;

        ipLocale = InetAddress.getLocalHost();
        System.out.println("IP locale (brut) : " + ipLocale);
        System.out.println("Nom machine locale : " + ipLocale.getHostName());
        System.out.println("Adresse IP machine locale : " +
ipLocale.getHostAddress());
        System.out.println("Nom de la classe : " + ipLocale.getClass().getName());

        ipGoogle = InetAddress.getByName("www.google.be");
        System.out.println("IP Google (brut) : " + ipGoogle);
        System.out.println("Nom machine Google : " + ipGoogle.getHostName());
        System.out.println("Adresse IP machine Google : " +
ipGoogle.getHostAddress());
    }
}
```

dont un exemple d'exécution fournit

```
# java TestInetAddress
IP locale (brut) : moon/192.168.228.167
Nom machine locale : moon
Adresse IP machine locale : 192.168.228.167
Nom de la classe : java.net.Inet4Address
IP Google (brut) : www.google.be/172.217.22.131
Nom machine Google : www.google.be
Adresse IP machine Google : 172.217.22.131
#
```

Remarque

La classe **InetAddress** possède 2 classes filles :

- **Inet4Address** → adresse IP V4
- **Inet6Address** → adresse IP V6

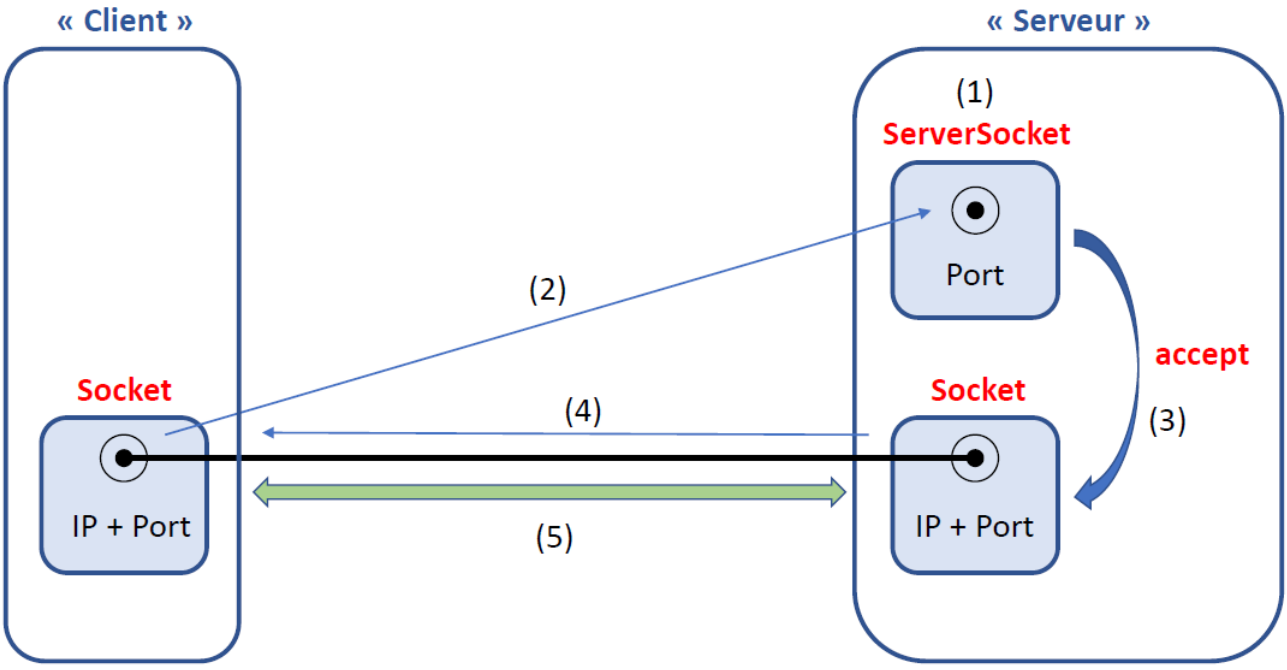
L'exemple ci-dessus montre que l'a obtenu une adresse IP de type 4.

Communication en mode connecté (TCP)

Pour ce type de communication, Java utilise les classes

- **ServerSocket** : qui permet à un processus « **serveur** » de se mettre en attente d'une connexion et, une fois la connexion établie, de fournir la socket (instance de la classe **Socket**) qui va permettre le transfert de données avec le « **client** » connecté
- **Socket** : qui permet à un processus « **client** » de se connecter à un processus « **serveur** » et, une fois la connexion établie, le transfert de données avec le « **serveur** »

Etablissement d'une connexion



Plus précisément,

1. Le serveur instancie un objet **ServerSocket** en précisant le port d'écoute et se met en attente d'une connexion en appelant la méthode bloquante **accept()**
2. Le client instancie un objet **Socket** en lui précisant l'adresse IP et le port d'écoute du serveur → le constructeur est bloquant jusqu'à ce que la connexion soit établie
3. Lorsque la demande de connexion est détectée au niveau du serveur, l'appel de la méthode **accept()** retourne une instance de la classe **Socket** → cette instance correspond à l'adresse IP et au port d'écoute du serveur
4. Le client est avisé que la connexion est établie → l'appel du constructeur de la classe **Socket** se débloque → cette instance contient l'adresse IP du client ainsi qu'un port libre qui lui a été attribué (sauf s'il le précise ; voir plus loin)
5. La connexion est établie entre le client et le serveur → le transfert de données peut commencer

La classe **ServerSocket** possède les constructeurs

- **public ServerSocket(int port) throws IOException** → crée une socket d'écoute en précisant le port sur lequel on veut attendre → le nombre total de connexions simultanées est limité à 50
- **public ServerSocket(int port, int nbCnx) throws IOException** → idem que le précédent sauf que l'on peut fixer le nombre maximum de connexions simultanées
- **public ServerSocket(int port, int nbCnx, InetAddress adrLocal) throws IOException** → idem que le précédent sauf que l'on peut préciser à quelle adresse IP de la machine « serveur » on souhaite lier (« binder ») la socket

Une fois instancié (**constructeur non bloquant**), un objet **ServerSocket** possède les méthodes

- **public Socket accept() throws IOException** → **bloquant** jusqu'à ce qu'un client réalise une demande de connexion → retourne un objet **Socket** représentant le point de communication établi avec le client
- **public synchronized void setSoTimeout(int timeout)** → permet de préciser un délai d'attente de connexion → si au bout de timeout millisecondes aucune connexion n'a été établie, la méthode **accept()** lance une exception du type **SocketTimeoutException**

Il est possible d'obtenir des renseignements sur la **ServerSocket** instanciée via les méthodes

- **public InetAddress getInetAddress()** → retourne l'adresse IP locale du serveur
- **public int getLocalPort()** → retourne le port local du serveur

La classe **Socket** possède les constructeurs **bloquants** :

- **public Socket(String machine, int port)** → précise le nom (adresse IP sous forme d'une chaîne de caractères, nom de la machine, …) et le port de la machine « serveur » sur laquelle on veut se connecter
- **public Socket(InetAddress adrDist, int port)** → précise l'adresse IP du serveur sur lequel on souhaite se connecter par l'intermédiaire d'un objet InetAddress
- **public Socket(String machineDist, int portDist, InetAddress adrLocal, int portLocal)** → permet en plus de préciser l'adresse IP et le port local du client
- **public Socket(InetAddress adrDist, int portDist, InetAddress adrLocal, int portLocal)** → idem que le précédent mais l'adresse IP de la machine locale client est spécifié par un objet InetAddress

Une fois la connexion établie, l'appel du constructeur se débloque et on peut obtenir des informations sur l'objet Socket à l'aide des méthodes :

- **public InetAddress getInetAddress()** → retourne l'adresse IP de la machine distante sur laquelle la socket est connectée
- **public InetAddress getLocalAddress()** → retourne l'adresse IP de la machine locale à laquelle est attaché la socket
- **public int getPort()** → retourne le port distant auquel la socket est connectée
- **public int getLocalPort()** → retourne le port local auquel la socket est attachée

Remarquez que ces 4 méthodes sont aussi bien utilisables par le client et que par le serveur.

Une connexion peut enfin être fermée par l'appel de la méthode **close()**.

Exemple d'établissement d'une connexion TCP

Dans l'exemple qui suit,

- Un programme « **client** », lancé sur une machine d'adresse IP **192.168.228.1** va se connecter à un programme « **Serveur** » lancé sur une machine d'adresse IP **192.168.228.167** en attente sur le port **50000**
- Une fois la connexion établie, client et serveur afficheront les informations de leurs sockets
- Le programme « **serveur** » va également se mettre en attente d'une seconde connexion en mettant en place un **time out** de 3 secondes

Voici le code du « **client** » (fichier **ClientTCP.java**) :

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

public class ClientTCP
{
    public static void main(String args[]) throws IOException
    {
        Socket csocket;

        // Création de la socket et connexion sur le serveur
        csocket = new Socket("192.168.228.167",50000);
        System.out.println("Connexion établie.");

        // Caractéristiques de la socket
        System.out.println("--- Socket ---");
        System.out.println("Adresse IP locale   : " +
csocket.getLocalAddress().getHostAddress());
        System.out.println("Port local          : " + csocket.getLocalPort());
        System.out.println("Adresse IP distante : " +
csocket.getInetAddress().getHostAddress());
        System.out.println("Port distant        : " + csocket.getPort());

        csocket.close();
    }
}
```

tandis que le code du « serveur » (fichier **ServeurTCP.java**) est

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

public class ServeurTCP
{
    public static void main(String args[]) throws IOException
    {
        ServerSocket ssocket;
        Socket csocket;

        // Création de la socket
        ssocket = new ServerSocket(50000);

        // Attente d'une connexion
        System.out.println("Attente d'une 1ere connexion...");
        csocket = ssocket.accept();
        System.out.println("Connexion établie.");

        // Caractéristiques des sockets
        System.out.println("--- ServerSocket ---");
        System.out.println("Adresse IP locale   : " +
ssocket.getInetAddress().getHostAddress());
        System.out.println("Port local          : " + ssocket.getLocalPort());

        System.out.println("--- Socket ---");
        System.out.println("Adresse IP locale   : " +
csocket.getLocalAddress().getHostAddress());
        System.out.println("Port local          : " + csocket.getLocalPort());
        System.out.println("Adresse IP distante : " +
csocket.getInetAddress().getHostAddress());
        System.out.println("Port distant        : " + csocket.getPort());

        // Fermeture de la connexion
        ssocket.close();
        csocket.close();

        // Attente d'une connexion avec Time Out
        ssocket = new ServerSocket(50000);
        ssocket.setSoTimeout(3000);
        System.out.println("Attente d'une 2eme  connexion...");
        try
        {
            csocket = ssocket.accept();
            System.out.println("Connexion établie.");
            ssocket.close();
            csocket.close();
        }
        catch(SocketTimeoutException e)
        {
            System.out.println("Time Out : " + e.getMessage());
        }
    }
}
```

Un exemple d'exécution du client est

```
#java ClientTCP
Connexion établie.
--- Socket ---
Adresse IP locale    : 192.168.228.1
Port local           : 52390
Adresse IP distante : 192.168.228.167
Port distant         : 50000
#
```

tandis que l'exécution du serveur (lancé avant !) est

```
# java ServeurTCP
Attente d'une 1ere connexion...
Connexion établie.
--- ServerSocket ---
Adresse IP locale    : 0.0.0.0
Port local           : 50000
--- Socket ---
Adresse IP locale    : 192.168.228.167
Port local           : 50000
Adresse IP distante : 192.168.228.1
Port distant         : 52390
Attente d'une 2eme  connexion...
Time Out : Accept timed out
#
```

On observe que

- Une fois la connexion établie, le port **52390** a été attribué à la **socket cliente**
- Aucune demande de connexion « client » n'a été faite lors de la 2ème attente de connexion du serveur → le **time out** a été généré → exception **SocketTimeoutException** lancée

Transfert de données

Une fois la connexion établie, les objets **Socket** permettent au client et au serveur de communiquer en bidirectionnel. Pour cela, on récupère les flux d'entrée/sortie via les méthodes :

- **public InputStream getInputStream()** throws IOException
- **public OutputStream getOutputStream()** throws IOException

Il suffit ensuite de monter par-dessus un flux de plus haut niveau comme

- **DataInputStream** et **DataOutputStream** → types de base
- **ObjectInputStream** et **ObjectOutputStream** → objets (sérialisation)
- **InputStreamReader** et **OutputStreamReader** → pour du texte
- …

Exemple de transmission de données de type « data » en TCP

Dans l'exemple qui suit,

- Un processus **client** va se connecter sur un processus **serveur** lancé sur une machine dont l'adresse IP est **192.168.228.167** en attente sur le port **50000**
- Processus client et serveur vont s'envoyer des « **types de base** » (int, String, double) en utilisant les flux **DataOutputStream** et **DataInputStream**
- Le **client** va récupérer en ligne de commande un **nom**, un **age** et un **poids** qui seront envoyés au serveur par 3 envois distincts
- Le **serveur** lit les données et envoie une réponse au client sous forme de 3 envois

Le code du **client** (fichier **ClientTCP.java**) est

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

public class ClientTCP
{
    public static void main(String argv[]) throws IOException
    {
        Socket csocket;

        // Création de la socket et connexion sur le serveur
        csocket = new Socket("192.168.228.167",50000);
        System.out.println("Connexion établie.");

        // Création des flux
        DataOutputStream dos = new DataOutputStream(csocket.getOutputStream());
        DataInputStream dis = new DataInputStream(csocket.getInputStream());

        // Préparation des données à envoyer
        String nom = argv[0];
        int age = Integer.parseInt(argv[1]);
        double poids = Double.parseDouble(argv[2]);
```

```
        // Envoi des données
        dos.writeUTF(nom);
        dos.writeInt(age);
        dos.writeDouble(poids);
        System.out.println("Requête envoyée.");

        // Lecture de la réponse
        String texte = dis.readUTF();
        int age2 = dis.readInt();
        Double poids2 = dis.readDouble();
        System.out.println("Réponse reçue.");

        // Fermeture de la connexion
        csocket.close();

        System.out.println(texte);
        System.out.println("L'année prochaine vous aurez " + age2 + " ans.");
        System.out.println("En perdant 10% de votre poids, vous pèserez " + poids2 +
" kg.");
    }
}
```

Le code du **serveur** (fichier **ServeurTCP.java**) est

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

public class ServeurTCP
{
    public static void main(String args[]) throws IOException
    {
        ServerSocket ssocket;
        Socket csocket;

        // Création de la socket d'écoute
        ssocket = new ServerSocket(50000);

        // Attente d'une connexion
        System.out.println("Attente d'une connexion...");
        csocket = ssocket.accept();
        System.out.println("Connexion établie.");

        // Création des flux
        DataOutputStream dos = new DataOutputStream(csocket.getOutputStream());
        DataInputStream dis = new DataInputStream(csocket.getInputStream());

        // Lecture des données
        String nom = dis.readUTF();
        int age = dis.readInt();
        double poids = dis.readDouble();
        System.out.println("Requête reçue.");

        // Préparation des réponses
```

```
        System.out.println("nom = " + nom);
        System.out.println("age = " + age);
        System.out.println("poids = " + poids);

        nom = "Bonjour Mr. " + nom + " !!!";
        age++;
        poids = 0.9F * poids;

        // Envois des réponses
        dos.writeUTF(nom);
        dos.writeInt(age);
        dos.writeDouble(poids);
        System.out.println("Réponse envoyée.");

        // Fermeture de la connexion
        ssocket.close();
        csocket.close();
    }
}
```

Un exemple d'exécution du client est

```
# java ClientTCP Wagner 48 78.35
Connexion établie.
Requête envoyée.
Réponse reçue.
Bonjour Mr. Wagner !!!
L'année prochaine vous aurez 49 ans.
En perdant 10% de votre poids, vous pèserez 70.51499813199042 kg.
#
```

tandis qu'un exemple d'exécution du serveur (lancé avant) est

```
# java ServeurTCP
Attente d'une connexion...
Connexion établie.
Requête reçue.
nom = Wagner
age = 48
poids = 78.35
Réponse envoyée.
#
```

Exemple de transmission de données de type « objet » en TCP (sérialisation)

Nous reprenons exactement le même exemple que précédemment à la différence que

- les trois données **nom**, **age** et **poids** sont encapsulées dans un objet **requete** instance de la classe **Donnee**
- Processus client et serveur s'échangent des objets en utilisant les flux **ObjectOutputStream** et **ObjectInputStream**
- Le serveur répond en envoyant un objet **reponse** instance de la même classe **Donnee**

Le code de la classe **Donnee** (fichier **Donnee.java**) est

```
import java.io.Serializable;

public class Donnee implements Serializable
{
    private String nom;
    private int age;
    private double poids;

    public Donnee(String n,int a,double p)
    {
        nom = n;
        age = a;
        poids = p;
    }

    public String getNom() {
        return nom;
    }

    public int getAge() {
        return age;
    }

    public double getPoids() {
        return poids;
    }
}
```

A noter que la classe **Donnee** doit implémenter l'interface **Serializable**.

Le code du **client** (fichier **Client TCP.java**) est

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

public class ClientTCP
{
    public static void main(String argv[]) throws IOException, ClassNotFoundException
    {
        Socket csocket;

        // Création de la socket et connexion sur le serveur
        csocket = new Socket("192.168.228.167",50000);
        System.out.println("Connexion établie.");

        // Création des flux
        ObjectOutputStream oos = new ObjectOutputStream(csocket.getOutputStream());
        ObjectInputStream ois = new ObjectInputStream(csocket.getInputStream());

        // Préparation des données à envoyer
        String nom = argv[0];
        int age = Integer.parseInt(argv[1]);
        double poids = Double.parseDouble(argv[2]);

        Donnee requete = new Donnee(nom,age,poids);

        // Envoi de la requête
        oos.writeObject(requete);
        System.out.println("Requête envoyée.");

        // Lecture de la réponse
        Donnee reponse = (Donnee) ois.readObject();
        System.out.println("Réponse reçue.");

        // Fermeture de la connexion
        csocket.close();

        System.out.println(reponse.getNom());
        System.out.println("L'année prochaine vous aurez " + reponse.getAge() + "
ans.");
        System.out.println("En perdant 10% de votre poids, vous pèserez " +
reponse.getPoids() + " kg.");
    }
}
```

tandis que le code du serveur (fichier ServeurTcp.java) est

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

public class ServeurTCP
{
    public static void main(String args[]) throws IOException, ClassNotFoundException
```

```
    {
        ServerSocket ssocket;
        Socket csocket;

        // Création de la socket d'écoute
        ssocket = new ServerSocket(50000);

        // Attente d'une connexion
        System.out.println("Attente d'une connexion...");
        csocket = ssocket.accept();
        System.out.println("Connexion établie.");

        // Création des flux
        ObjectOutputStream oos = new ObjectOutputStream(csocket.getOutputStream());
        ObjectInputStream ois = new ObjectInputStream(csocket.getInputStream());

        // Lecture de la requête
        Donnee requete = (Donnee) ois.readObject();
        System.out.println("Requête reçue.");

        // Préparation de la réponse
        System.out.println("nom = " + requete.getNom());
        System.out.println("age = " + requete.getAge());
        System.out.println("poids = " + requete.getPoids());

        String texte = "Bonjour Mr. " + requete.getNom() + " !!!";
        int age = requete.getAge() + 1;
        double poids = 0.9F * requete.getPoids();

        Donnee reponse = new Donnee(texte,age,poids);

        // Envoi de la réponse
        oos.writeObject(reponse);
        System.out.println("Réponse envoyée.");

        // Fermeture de la connexion
        ssocket.close();
        csocket.close();
    }
}
```

Un exemple d'exécution du client est

```
# java ClientTCP Wagner 48 78.35
Connexion établie.
Requête envoyée.
Réponse reçue.
Bonjour Mr. Wagner !!!
L'année prochaine vous aurez 49 ans.
En perdant 10% de votre poids, vous pèserez 70.51499813199042 kg.
#
```

tandis qu'un exemple d'exécution du serveur (lancé avant) est

```
# java ServeurTCP
Attente d'une connexion...
Connexion établie.
Requête reçue.
nom = Wagner
age = 48
poids = 78.35
Réponse envoyée.
#
```

Exemple de transmission de données de type « texte » en TCP

Dans l'exemple qui suit,

- Un processus **client** va se connecter sur un processus **serveur** lancé sur une machine dont l'adresse IP est **192.168.228.167** en attente sur le port **50000**
- Processus **client** et **serveur** vont d'échanger des données sous formes de chaines de caractères
- 3 couches de flux seront utilisées :
  - **OutputStream** et **InputStream** obtenus par les sockets → bas niveau (bytes)
  - **OutputSteamWriter** et **InputStreamReader** montés par-dessus les flux précédents → transformation de caractères selon le charset
  - **BufferedWriter** et **BufferedReader** montés par-dessus les flux précédents → concentration de caractères en lignes

Le code du **client** (fichier **ClientTCP.java**) est

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

public class ClientTCP
{
    public static void main(String argv[]) throws IOException
    {
        Socket csocket;

        // Création de la socket et connexion sur le serveur
        csocket = new Socket("192.168.228.167",50000);
        System.out.println("Connexion établie.");

        // Création des flux
```

```
        OutputStreamWriter osr = new OutputStreamWriter(csocket.getOutputStream());
        InputStreamReader isr = new InputStreamReader(csocket.getInputStream());

        BufferedWriter bw = new BufferedWriter(osr);
        BufferedReader br = new BufferedReader(isr);

        // Echanges de textes
        bw.write("Bonjour !");
        bw.newLine();
        bw.write("Comment allez-vous ?");
        bw.newLine();
        bw.flush();

        String reponse;
        reponse = br.readLine();
        System.out.println("Reçu : --" + reponse + "--");
        reponse = br.readLine();
        System.out.println("Reçu : --" + reponse + "--");

        // Fermeture de la connexion
        csocket.close();
    }
}
```

Le code du **serveur** (fichier **ServeurTCP.java**) est

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

public class ServeurTCP
{
    public static void main(String args[]) throws IOException
    {
        ServerSocket ssocket;
        Socket csocket;

        // Création de la socket d'écoute
        ssocket = new ServerSocket(50000);

        // Attente d'une connexion
        System.out.println("Attente d'une connexion...");
        csocket = ssocket.accept();
        System.out.println("Connexion établie.");

        // Création des flux
        OutputStreamWriter osr = new OutputStreamWriter(csocket.getOutputStream());
        InputStreamReader isr = new InputStreamReader(csocket.getInputStream());

        BufferedWriter bw = new BufferedWriter(osr);
        BufferedReader br = new BufferedReader(isr);

        // Echanges de textes
        String requete;
        requete= br.readLine();
```

```
        System.out.println("Reçu : --" + requete + "--");
        requete= br.readLine();
        System.out.println("Reçu : --" + requete + "--");

        bw.write("Je vais bien merci.");
        bw.newLine();
        bw.write("Vous aussi ?");
        bw.newLine();
        bw.flush();

        // Fermeture de la connexion
        ssocket.close();
        csocket.close();
    }
}
```

On remarquera que c'est au moment de l'appel de **flush()** que le buffer est effectivement vidé et que les données transitent sur le réseau.

Un exemple d'exécution du client est

```
# java ClientTCP
Connexion établie.
Reçu : --Je vais bien merci.--
Reçu : --Vous aussi ?--
#
```

Tandis qu'un exemple d'exécution du serveur (lancé avant) est

```
# java ServeurTCP
Attente d'une connexion...
Connexion établie.
Reçu : --Bonjour !--
Reçu : --Comment allez-vous ?--
#
```

Exemple de communication « Java/C » au niveau du « byte » en TCP

Lorsque client et serveur ne sont pas programmés dans le même langage, il est nécessaire de communiquer au plus bas niveau possible → envoi et réception de bytes

Dans l'exemple qui suit,

- Un **client** écrit en **C** va se connecter sur un **serveur** écrit en **Java** et lancé sur une machine dont l'adresse IP est **192.168.228.167** en attente sur le port **50000**

- **Client** et **serveur** s'envoie des **« trames » de bytes** → afin de savoir, lors de la lecture, byte par byte, si on arrive à la fin de la trame, on a jouté **2 caractères de « fin de transmission »** à la fin de la trame → ceux-ci ont été choisis comme « **$%** »

Le code du **client C** (fichier **ClientTCP.cpp**) est

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define IP   "192.168.228.167"
#define PORT 50000

int main()
{
  int    socketClient;
  struct sockaddr_in adresseSocket;

  // Creation de la socket client
  if ((socketClient = socket(AF_INET,SOCK_STREAM,0)) == -1)
  {
    perror("Erreur de socket");
    exit(1);
  }

  // Preparation de la structure sockaddr_in
  memset(&adresseSocket,0,sizeof(struct sockaddr_in));
  adresseSocket.sin_family = AF_INET;
  adresseSocket.sin_port = htons(PORT);
  adresseSocket.sin_addr.s_addr = inet_addr(IP);

  // Connexion
  if (connect(socketClient,(struct sockaddr *)&adresseSocket,sizeof(struct
sockaddr_in)) == -1)
  {
    perror("Erreur de connect");
    exit(1);
  }

  printf("Connexion etablie.\n");

  // Envoi de la requete
  char requete[80];
  sprintf(requete,"wagner#48#J'ai gagne 3.5$ !");
  char trame[100];
  sprintf(trame,"%s%s",requete,"$%");
  printf("Trame envoyee : --%s--\n",trame);
  printf("Nombre bytes envoyes : %d\n",strlen(trame));
```

```
  if (send(socketClient,trame,strlen(trame),0) != strlen(trame))
  {
      perror("Erreur de send");
      exit(1);
  }

  printf("Requete envoyee.\n");

  // Lecture de la reponse
  char buffer[100], b1, b2;
  int  i = 0;
  bool EOT = false;
  while(!EOT) // boucle de lecture byte par byte
  {
    if (recv(socketClient,&b1,1,0) != 1) { perror("Erreur de recv"); exit(1); }
    printf("b1 = --%c--\n",b1);
    if (b1 == '$')
    {
      if (recv(socketClient,&b2,1,0) != 1) { perror("Erreur de recv"); exit(1); }
       printf("b2 = --%c--\n",b2);
      if (b2 == '%') EOT = true;
      else
      {
              buffer[i] = b1; i++;
              buffer[i] = b2; i++;
      }
    }
    else
    {
      buffer[i] = b1; i++;
    }
  }

  buffer[i] = '\0';
  printf("Recu : --%s--\n",buffer);

  // Fermeture de la connexion
  close(socketClient);
  exit(0);
}
```

tandis que le code du **serveur Java** (fichier **Serveur TCP.java**) est

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

public class ServeurTCP
{
    public static void main(String args[]) throws IOException
    {
        ServerSocket ssocket;
        Socket csocket;
```

```
        // Création de la socket d'écoute
        ssocket = new ServerSocket(50000);

        // Attente d'une connexion
        System.out.println("Attente d'une connexion...");
        csocket = ssocket.accept();
        System.out.println("Connexion établie.");

        // Création des flux
        DataOutputStream dos = new DataOutputStream(new
BufferedOutputStream(csocket.getOutputStream()));
        DataInputStream dis = new DataInputStream(new
BufferedInputStream(csocket.getInputStream()));

        // Lecture de la requête
        StringBuffer buffer = new StringBuffer();
        boolean EOT = false;
        while(!EOT) // boucle de lecture byte par byte
        {
            byte b1 = dis.readByte();
            System.out.println("b1 : --" + (char)b1 + "--");
            if (b1 == (byte)'$')
            {
                byte b2 = dis.readByte();
                System.out.println("b2 : --" + (char)b2 + "--");
                if (b2 == (byte)'%') EOT = true;
                else
                {
                    buffer.append((char)b1);
                    buffer.append((char)b2);
                }
            }
            else buffer.append((char)b1);
        }

        String requete = buffer.toString();
        System.out.println("Reçu : --" + requete + "--");

        // Envoi de la réponse
        String reponse = "Bravo $$$ a vous !";
        String trame = reponse + "$%";
        dos.write(trame.getBytes());
        dos.flush();
        System.out.println("Réponse envoyée.");

        // Fermeture de la connexion
        ssocket.close();
        csocket.close();
    }
}
```

Un exemple d'exécution du **client C** est

```
# ClientTCP
Connexion etablie.
Trame envoyee : --wagner#48#J'ai gagne 3.5$ !$%--
Nombre bytes envoyes : 29
Requete envoyee.
b1 = --B--
b1 = --r--
b1 = --a--
b1 = --v--
b1 = --o--
b1 = -- --
b1 = --$--
b2 = --$--
b1 = --$--
b2 = -- --
b1 = --a--
b1 = -- --
b1 = --v--
b1 = --o--
b1 = --u--
b1 = --s--
b1 = -- --
b1 = --!--
b1 = --$--
b2 = --%--
Recu : --Bravo $$$ a vous !—
#
```

tandis qu'un exemple d'exécution du **serveur Java** (lancé avant) est

```
# java ServeurTCP
Attente d'une connexion...
Connexion établie.
b1 : --w--
b1 : --a--
b1 : --g--
b1 : --n--
b1 : --e--
b1 : --r--
b1 : --#--
b1 : --4--
b1 : --8--
b1 : --#--
b1 : --J--
b1 : --'--
b1 : --a--
b1 : --i--
b1 : -- --
b1 : --g--
b1 : --a--
b1 : --g--
```

```
b1 : --n--
b1 : --e--
b1 : -- --
b1 : --3--
b1 : --.--
b1 : --5--
b1 : --$--
b2 : -- --
b1 : --!--
b1 : --$--
b2 : --%--
Reçu : --wagner#48#J'ai gagne 3.5$ !--
Réponse envoyée.
#
```

Remarque

Au lieu de coller à la fin de la trame un ou plusieurs caractères de « fin de transmission », on pourrait imaginer

1. Envoyer tout d'abord le nombre **N** de bytes à envoyer
2. Envoyer les **N bytes**

Ainsi lors de la lecture,

1. On lit tout d'abord un nombre correspondant au nombre **N** de bytes à lire
2. On lit les **N bytes** qui ont été envoyés

Le nombre N devra dans ce cas être écrit sous forme de « caractères » (ou bytes donc ; comme une « **entête de trame** ») et non format « int ».

Exemple de serveur TCP multi-threads générique

Contexte et objectifs

Le but ici est de créer un serveur

- **multi-threads** basé sur les modèles « Producteur/Consommateur » à la demande et en pool
- **de connexions**, c'est-à-dire qu'une fois un qu'un client s'est connecté sur le serveur, c'est toujours le même thread qui s'occupe de lui et cela jusqu'à la fin de sa connexion (ce type de serveur est à comparer aux « serveurs de requêtes » dans lequel une requête est attribuée à un thread → les requêtes d'un client connecté peuvent être traitées par des threads différents → à chaque requête, une nouvelle connexion pourrait être établie)
- **générique** dans le sens où les threads ne savent absolument pas pour quel protocole ils travaillent → le traitement du protocole est délégué à des interfaces et des classes spécifiques indépendantes des classes du serveur lui-même

La gestion du protocole : des interfaces

Le protocole n'étant pas connu du serveur générique, il sera représenté par des interfaces.

Nous avons l'interface **Requete** (fichier **Requete.java**) :

```
package ServeurGeneriqueTCP;

import java.io.*;

public interface Requete extends Serializable { }
```

et l'interface **Reponse** (fichier **Reponse.java**) :

```
package ServeurGeneriqueTCP;

import java.io.*;

public interface Reponse extends Serializable { }
```

qui devront toutes deux être implémentées par les futures classes du protocole proprement dit. Elles héritent toutes deux de l'interface **Serializable** car les requêtes et les réponses transiteront sur le réseau sous forme d'objets sérialisés.

Le protocole est représenté par l'interface **Protocole** (fichier **Protocole.java**) :

```
package ServeurGeneriqueTCP;

import java.net.Socket;

public interface Protocole
{
    String  getNom();
    Reponse TraiteRequete(Requete requete, Socket socket) throws
FinConnexionException;
}
```

qui comporte

- une méthode retournant le nom du protocole
- une méthode qui assure le traitement des requêtes → c'est elle qui va permettre de rendre le serveur **générique** (la socket correspond au client connecté et peut être utilisée par le protocole mais ce n'est pas le protocole qui envoie la réponse)

Le traitement des requêtes étant générique, le serveur ne sait pas quand il doit fermer la connexion avec le client. Voilà pourquoi la méthode **TraiteRequete** est susceptible de lancer l'exception **FinConnexionException** (fichier **FinConnexionException.java**) :

```
package ServeurGeneriqueTCP;

public class FinConnexionException extends Exception
{
    private Reponse reponse;

    public FinConnexionException(Reponse reponse)
    {
        super("Fin de Connexion décidée par protocole");
        this.reponse = reponse;
    }

    public Reponse getReponse()
    {
        return reponse;
    }
}
```

Celle-ci sera lancée lorsque le protocole décidera que la connexion doit se fermer après le traitement de la requête → si une dernière réponse doit être envoyée au client avant fermeture, celle-ci sera récupérée par le serveur générique dans l'exception catchée.

Le Thread Client

Quel que soit le modèle de serveur choisi (à la demande ou en pool), un thread (lancé dans le processus serveur) va être chargé de s'occuper entièrement de la connexion de ce client → on appelle ce thread le **Thread client** (fichier **ThreadClient.java**) :

```
package ServeurGeneriqueTCP;

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

public abstract class ThreadClient extends Thread
{
    protected Protocole protocole;
    protected Socket csocket;
    protected Logger logger;
    private   int numero;

    private static int numCourant = 1;

    public ThreadClient(Protocole protocole, Socket csocket, Logger logger) throws
IOException
    {
        super("TH Client " + numCourant + " (protocole=" + protocole.getNom() + ")");
        this.protocole = protocole;
        this.csocket = csocket;
        this.logger = logger;
        this.numero = numCourant++;
    }

    public ThreadClient(Protocole protocole, ThreadGroup groupe, Logger logger)
throws IOException
    {
        super(groupe,"TH Client " + numCourant + " (protocole=" + protocole.getNom()
+ ")");
        this.protocole = protocole;
        this.csocket = null;
        this.logger = logger;
        this.numero = numCourant++;
    }

    @Override
    public void run()
    {
        try
        {
```

```
            ObjectOutputStream oos = null;
            ObjectInputStream  ois = null;

            try
            {
                ois = new ObjectInputStream(csocket.getInputStream());
                oos = new ObjectOutputStream(csocket.getOutputStream());

                while (true)
                {
                    Requete requete = (Requete) ois.readObject();
                    Reponse reponse = protocole.TraiteRequete(requete,csocket);
                    oos.writeObject(reponse);
                }
            }
            catch (FinConnexionException ex)
            {
                logger.Trace("Fin connexion demandée par protocole");
                if (oos != null && ex.getReponse() != null)
oos.writeObject(ex.getReponse());
            }
        }
        catch (IOException ex) { logger.Trace("Erreur I/O"); }
        catch (ClassNotFoundException ex) { logger.Trace("Erreur requete invalide");
}
        finally
        {
            try { csocket.close(); }
            catch (IOException ex) { logger.Trace("Erreur fermeture socket"); }
        }
    }
}
```

On constate que

- la classe **ThreadClient** est abstraite → une classe ThreadClient spécifique à chaque modèle sera héritée de cette classe mais la méthode **run()** de **traitement de la connexion** sera la même quelle que soit la classe héritée
- on attribuera à chaque thread un numéro qui permettra de lui générer un nom précis
- deux constructeurs sont présents : le premier sera utilisé pour le modèle « à la demande » et le second par le modèle « en pool » (voir plus loin)
- la **socket** correspond au client déjà connecté sur le serveur
- On observe bien la boucle « infinie » de gestion des requêtes → celle-ci ne prendra fin que lorsque la méthode **TraiteRequete** de l'objet protocole lancera l'exception **FinConnexionException** → de plus, si l'exception contient une réponse, celle-ci est envoyée au client

- Dans tous les cas, la socket est fermée avant la fin de la méthode **run()** → **fin de la connexion**

L'interface **Logger** (fichier **Logger.java**)

```
package ServeurGeneriqueTCP;

public interface Logger
{
    void Trace(String message);
}
```

permettra simplement de faire des traces (des « logs ») de messages sur une sortie dédiée : un fichier, en console, dans une interface graphique, …

Le Thread Serveur

Au sein du processus serveur, un thread sera dédié à l'acception des demandes de connexion par les clients et ces « connexions établies » (les sockets de service) seront transmises aux threads clients → on appelle ce thread le **Thread Serveur** (fichier **ThreadServeur.java**) :

```
package ServeurGeneriqueTCP;

import java.io.IOException;
import java.net.ServerSocket;

public abstract class ThreadServeur extends Thread
{
    protected int port;
    protected Protocole protocole;
    protected Logger logger;

    protected ServerSocket ssocket;

    public ThreadServeur(int port, Protocole protocole, Logger logger) throws
IOException
    {
        super("TH Serveur (port=" + port + ",protocole=" + protocole.getNom() + ")");
        this.port = port;
        this.protocole = protocole;
        this.logger = logger;

        ssocket = new ServerSocket(port);
    }
}
```

On constate que

- La classe **ThreadServeur** est abstraite → une classe ThreadServeur spécifique à chaque modèle sera héritée de cette classe et disposera de sa propre méthode run()
- Le **port** reçu en paramètre permet de créer la **socket d'écoute**

Le modèle « à la demande »

Dans ce modèle, à chaque nouvelle connexion établie, le thread serveur va créer un thread client et lui transmettre la socket d'écoute obtenue lors de l'**accept()**.

Le code du **thread Client** pour ce modèle (fichier **ThreadClientDemande.java**) est

```
package ServeurGeneriqueTCP;

import java.io.IOException;
import java.net.Socket;

public class ThreadClientDemande extends ThreadClient
{
    public ThreadClientDemande(Protocole protocole, Socket csocket, Logger logger)
throws IOException
    {
        super(protocole, csocket, logger);
    }

    @Override
    public void run()
    {
        logger.Trace("TH Client (Demande) démarre...");
        super.run();
        logger.Trace("TH Client (Demande) se termine.");
    }
}
```

On constate que la vie d'un thread client dans ce modèle est d'exécuter la méthode **run()** de sa classe mère, c'est-à-dire **traiter la connexion dans son entièreté** avant de **se terminer**.

Le code du **thread Serveur** dans ce modèle (fichier **ThreadServeurDemande.java**) est

```
package ServeurGeneriqueTCP;

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

public class ThreadServeurDemande extends ThreadServeur
{
    public ThreadServeurDemande(int port, Protocole protocole, Logger logger) throws
IOException
    {
        super(port, protocole, logger);
    }

    @Override
    public void run()
    {
        logger.Trace("Démarrage du TH Serveur (Demande)...");
        while(!this.isInterrupted())
        {
            Socket csocket;
            try
            {
                ssocket.setSoTimeout(2000);
                csocket = ssocket.accept();
                logger.Trace("Connexion acceptée, création TH Client");
                Thread th = new ThreadClientDemande(protocole,csocket,logger);
                th.start();
            }
            catch (SocketTimeoutException ex)
            {
                // Pour vérifier si le thread a été interrompu
            }
            catch (IOException ex)
            {
                logger.Trace("Erreur I/O");
            }
        }
        logger.Trace("TH Serveur (Demande) interrompu.");
        try { ssocket.close(); }
        catch (IOException ex) { logger.Trace("Erreur I/O"); }
    }
}
```

On constate que

- la socket d'écoute a été configurée avec un **time out** afin de vérifier toutes les 2 secondes si le thread Serveur a reçu une demande d'interruption.
- à chaque nouvel **accept(),** le thread Serveur crée et lance un nouveau thread client en lui passant en paramètre la socket de service obtenue.

Le modèle « en pool »

Dans ce modèle, le thread serveur doit créer un pool de threads clients qui vont tous se mettre en attente de connexions (plus précisément de socket de clients connectés).

Etant donné que le nombre de threads clients est limité, il se peut que certaines **connexions** soient **mises en attente** et doivent être stockées dans une « **file d'attente** » dont le code (fichier **FileAttente.java**) est

```
package ServeurGeneriqueTCP;

import java.net.Socket;
import java.util.LinkedList;

public class FileAttente
{
    private LinkedList<Socket> fileAttente;

    public FileAttente()
    {
        fileAttente = new LinkedList<>();
    }

    public synchronized void addConnexion(Socket socket)
    {
        fileAttente.addLast(socket);
        notify();
    }

    public synchronized Socket getConnexion() throws InterruptedException
    {
        while(fileAttente.isEmpty()) wait();
        return fileAttente.remove();
    }
}
```

On constate que la classe **FileAttente** est

- en réalité un conteneur de sockets clients connectés
- un **moniteur** avec 2 méthodes **synchronized** d'ajout et de retrait de sockets

Le code du **thread Client** pour ce modèle (fichier **ThreadClientPool.java**) est

```
package ServeurGeneriqueTCP;

import java.io.IOException;

public class ThreadClientPool extends ThreadClient
{
    private FileAttente connexionsEnAttente;

    public ThreadClientPool(Protocole protocole, FileAttente file, ThreadGroup
groupe, Logger logger) throws IOException
    {
        super(protocole, groupe, logger);
        connexionsEnAttente = file;
    }

    @Override
    public void run()
    {
        logger.Trace("TH Client (Pool) démarre...");
        boolean interrompu = false;
        while(!interrompu)
        {
            try
            {
                logger.Trace("Attente d'une connexion...");
                csocket = connexionsEnAttente.getConnexion();
                logger.Trace("Connexion prise en charge.");
                super.run();
            }
            catch (InterruptedException ex)
            {
                logger.Trace("Demande d'interruption...");
                interrompu = true;
            }
        }
        logger.Trace("TH Client (Pool) se termine.");
    }
}
```

On constate que

- le thread client tourne indéfiniment dans une boucle infinie jusqu'au moment où il recevra une demande d'interruption
- à chaque tour de boucle, le thread client **attend et récupère une socket de service** dans la **file d'attente** et **traite la connexion** en appelant la méthode **run()** de sa classe mère

- tous les threads clients d'un **même pool** feront partie d'un même **groupe de threads** créé par le thread serveur. Ainsi, lors d'une demande d'interruption, le thread serveur pourra les interrompre tous en même temps.

Le code du **thread Serveur** dans ce modèle (fichier **ThreadServeurPool.java**) est

```
package ServeurGeneriqueTCP;

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

public class ThreadServeurPool extends ThreadServeur
{
    private FileAttente connexionsEnAttente;
    private ThreadGroup pool;
    private int taillePool;

    public ThreadServeurPool(int port, Protocole protocole, int taillePool, Logger
logger) throws IOException
    {
        super(port, protocole, logger);

        connexionsEnAttente = new FileAttente();
        pool = new ThreadGroup("POOL");
        this.taillePool = taillePool;
    }

    @Override
    public void run()
    {
        logger.Trace("Démarrage du TH Serveur (Pool)...");
        // Création du pool de threads
        try
        {
            for (int i=0 ; i<taillePool ; i++)
            new ThreadClientPool(protocole,connexionsEnAttente,pool,logger).start();
        }
        catch (IOException ex)
        {
            logger.Trace("Erreur I/O lors de la création du pool de threads");
            return;
        }

        // Attente des connexions
        while(!this.isInterrupted())
        {
            Socket csocket;
            try
            {
                ssocket.setSoTimeout(2000);
                csocket = ssocket.accept();
                logger.Trace("Connexion acceptée, mise en file d'attente.");
```

```
                connexionsEnAttente.addConnexion(csocket);
            }
            catch (SocketTimeoutException ex)
            {
                // Pour vérifier si le thread a été interrompu
            }
            catch (IOException ex)
            {
                logger.Trace("Erreur I/O");
            }
        }
        logger.Trace("TH Serveur (Pool) interrompu.");
        pool.interrupt();
    }
}
```

On constate que

- Le thread serveur crée un **groupe de threads** appelé « POOL » (choix arbitraire ici) et crée son pool de threads en leur passant en paramètre la **file d'attente** des connexions acceptées et le **groupe de threads** auquel les threads clients doivent se rattacher
- A chaque **accept()**, le thread serveur ajoute la socket de service obtenue dans la file d'attente et se remet en attente d'acception d'une nouvelle connexion → la méthode **addConnexion** de la file d'attente va réveiller un des threads du pool en attente de connexions
- La socket d'écoute a été configurée avec un **time out** afin que le thread serveur vérifie toutes les 2 secondes s'il a reçu une demande d'interruption → si c'est le cas, il envoie une demande équivalente à tous les threads du pool avant de se terminer

Toutes les classes et interface du serveur générique sont à présents terminés et présents dans le package ServeurGeneriqueTCP que l'on pourrait embarquer dans un « jar librairie » utilisable dans un autre projet.

Exemple d'utilisation du serveur générique : le protocole LILOC

En guise d'exemple, nous mettons en place un protocole très simple appelée **LILOC** (« **L**og **I**n **L**og **O**ut **C**alcul ») à 3 requêtes/réponses, permettant à un client de se (dé)logger sur un serveur réalisant des opérations algébriques simples ('+', '-', 'x', '/') sur des entiers.

La requête de « Log In » est modélisée par la classe **RequeteLOGIN** (fichier **RequeteLOGIN.java**) :

```
package ProtocoleLILOC;

import ServeurGeneriqueTCP.*;

public class RequeteLOGIN implements Requete
{
    private String login;
    private String password;

    public RequeteLOGIN(String l,String p) {
        login = l;
        password = p;
    }

    public String getLogin() {
        return login;
    }

    public String getPassword() {
        return password;
    }
}
```

et dont la réponse est la classe **ReponseLOGIN** (fichier **ReponseLOGIN.java**) :

```
package ProtocoleLILOC;

import ServeurGeneriqueTCP.*;

public class ReponseLOGIN implements Reponse
{
    private boolean valide;

    ReponseLOGIN(boolean v) {
        valide = v;
    }

    public boolean isValide() {
        return valide;
    }
}
```

On constate que

- La classe **RequeteLOGIN** implémente l'interface **Requete** du serveur générique et comporte simplement un login et un mot de passe
- La classe **ReponseLOGIN** implémente l'interface **Reponse** du serveur générique et comporte le résultat de la requête à savoir un booléen à true si le couple login/mot de passe est correct ou à false sinon

La requête de « Log Out » est modélisée par la classe **RequeteLOGOUT** (fichier **RequeteLOGOUT.java**) :

```
package ProtocoleLILOC;

import ServeurGeneriqueTCP.*;

public class RequeteLOGOUT implements Requete
{
    private String login;

    public RequeteLOGOUT(String l) {
        login = l;
    }

    public String getLogin() {
        return login;
    }
}
```

Cette requête comporte simplement le login de l'utilisateur qui veut se délogger et ne nécessite pas de réponse de la part du serveur.

La requête de « Calcul » est modélisée par la classe **RequeteCALCUL** (fichier **RequeteCALCUL.java**) :

```
package ProtocoleLILOC;

import ServeurGeneriqueTCP.*;

public class RequeteCALCUL implements Requete
{
    private char operation;
    private int operande1;
    private int operande2;

    public RequeteCALCUL(char op,int val1,int val2) {
        operation = op;
        operande1 = val1;
```

```
        operande2 = val2;
    }

    public char getOperation() {
        return operation;
    }

    public int getOperande1() {
        return operande1;
    }

    public int getOperande2() {
        return operande2;
    }
}
```

et dont la réponse est la classe **ReponseCALCUL** (fichier **ReponseCALCUL.java**) :

```
package ProtocoleLILOC;

import ServeurGeneriqueTCP.*;

public class ReponseCALCUL implements Reponse
{
    private int resultat;

    public ReponseCALCUL(int r) {
        resultat = r;
    }

    public int getResultat() {
        return resultat;
    }
}
```

On constate que

- La requête présente 3 variables membres : l'opération représentée sous forme d'un caractère pouvant prendre les valeurs '+', '-', 'x', '/', et 2 opérandes entières
- La réponse comporte simplement le résultat du calcul

Le protocole proprement dit est représenté par la classe **LILOC** (fichier **LILOC.java**) :

```
package ProtocoleLILOC;

import ServeurGeneriqueTCP.*;
import java.net.Socket;
import java.util.HashMap;
```

```
public class LILOC implements Protocole
{
    private HashMap<String,String> passwords;
    private HashMap<String,Socket> clientsConnectes;

    private Logger logger;

    public LILOC(Logger log)
    {
        passwords = new HashMap<>();
        passwords.put("wagner","abcd");
        passwords.put("charlet","1234");
        passwords.put("calmant","azerty");

        logger = log;

        clientsConnectes = new HashMap<>();
    }

    @Override
    public String getNom()
    {
        return "LILOC";
    }

    @Override
    public synchronized Reponse TraiteRequete(Requete requete, Socket socket) throws
FinConnexionException
    {
        if (requete instanceof RequeteLOGIN) return TraiteRequeteLOGIN((RequeteLOGIN)
requete, socket);
        if (requete instanceof RequeteCALCUL) return
TraiteRequeteCALCUL((RequeteCALCUL) requete);
        if (requete instanceof RequeteLOGOUT) TraiteRequeteLOGOUT((RequeteLOGOUT)
requete);
        return null;
    }

    private synchronized ReponseLOGIN TraiteRequeteLOGIN(RequeteLOGIN requete, Socket
socket) throws FinConnexionException
    {
        logger.Trace("RequeteLOGIN reçue de " + requete.getLogin());
        String password = passwords.get(requete.getLogin());
        if (password != null)
            if (password.equals(requete.getPassword()))
            {
                String ipPortClient = socket.getInetAddress().getHostAddress() + "/"
+ socket.getPort();
                logger.Trace(requete.getLogin() + " correctement loggé de " +
ipPortClient);
                clientsConnectes.put(requete.getLogin(),socket);
                return new ReponseLOGIN(true);
            }
        logger.Trace(requete.getLogin() + " --> erreur de login");
        throw new FinConnexionException(new ReponseLOGIN(false));
```

```
    }

    private synchronized void TraiteRequeteLOGOUT(RequeteLOGOUT requete) throws
FinConnexionException
    {
        logger.Trace("RequeteLOGOUT reçue de " + requete.getLogin());
        clientsConnectes.remove(requete.getLogin());
        logger.Trace(requete.getLogin() + " correctement déloggé");
        throw new FinConnexionException(null);
    }

    private synchronized ReponseCALCUL TraiteRequeteCALCUL(RequeteCALCUL requete)
    {
        logger.Trace("RequeteCALCUL reçue");
        switch (requete.getOperation())
        {
            case '+' : return new ReponseCALCUL(requete.getOperande1() +
requete.getOperande2());
            case '-' : return new ReponseCALCUL(requete.getOperande1() -
requete.getOperande2());
            case '*' : return new ReponseCALCUL(requete.getOperande1() *
requete.getOperande2());
            case '/' : return new ReponseCALCUL(requete.getOperande1() /
requete.getOperande2());
        }
        return null;
    }
}
```

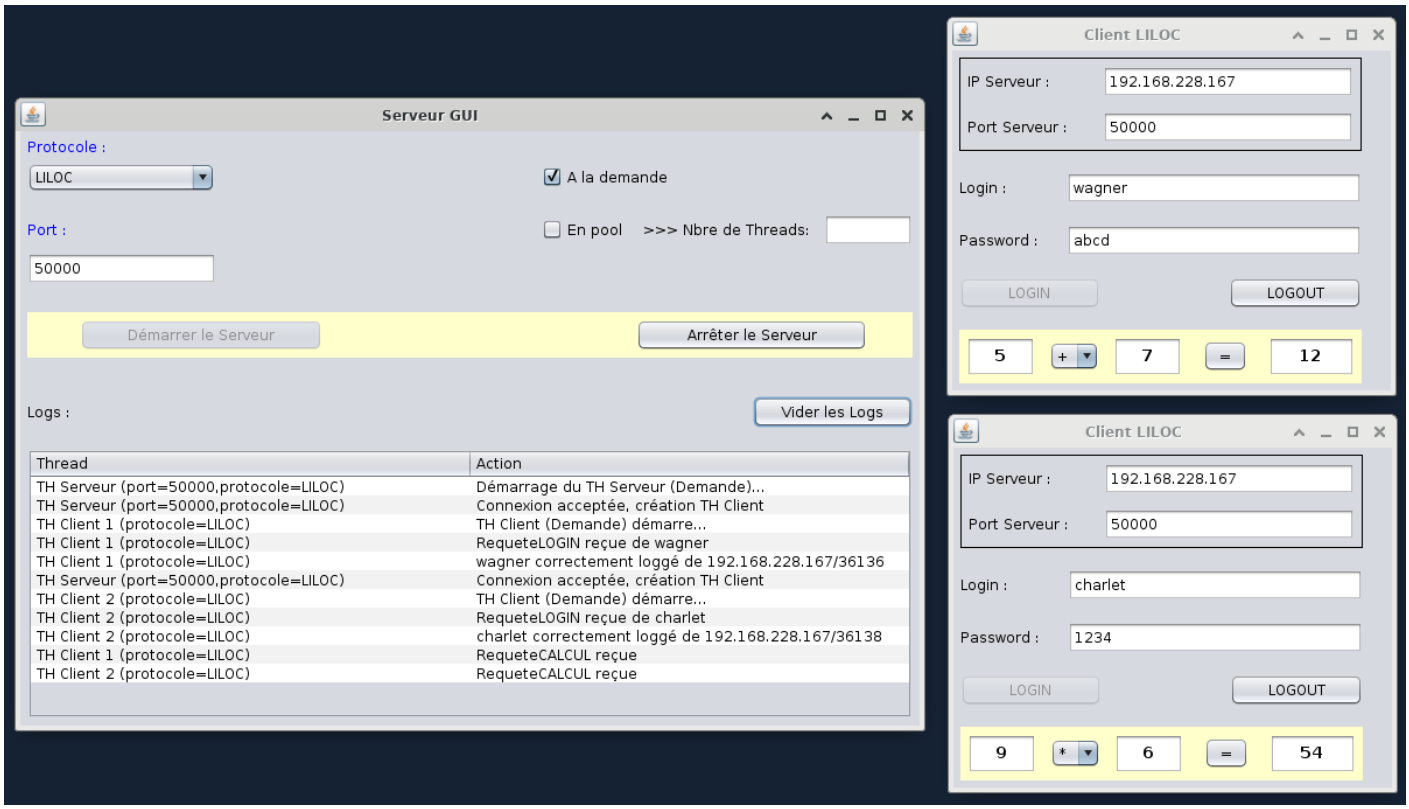
On constate que

- La classe **LILOC** implémente l'interface **Protocole** du serveur générique et implémente donc la méthode **TraiteRequete** qui appelle la méthode adaptée à chaque type de requête reçue → le type de requête reçue est détectée à l'aide de l'opérateur instanceof
- La classe LILOC contient 2 conteneurs de type **HashMap**, un contenant les couples **login/mot de passe** connus (et donc acceptés) et un second permettant de stocker les clients connectés selon les couples **login/socket** → ce dernier pourrait être utilisé afin de « monitorer » le serveur (savoir qui est connecté et interagir avec eux)
- La méthode **TraiteRequeteLOGIN** vérifie si le login et le mot de passe reçus sont corrects. Si c'est le cas, la socket du client est stockée dans la HashMap et un objet ReponseLOGIN contenant true est retourné. Si le couple login/mot de passe n'est pas correct, la connexion doit être interrompue et une exception **FinConnexionException** est lancée → celle-ci contient de plus une réponse à envoyer au client.

- La méthode **TraiteRequeteLOGOUT** retire simplement le client de la HashMap des clients connectés et met fin à la connexion avec le client en lançant une exception **FinConnexionException** qui ne contient aucune réponse à envoyer au client.
- La méthode **TraiteRequeteCALCUL** crée et retourne un objet ReponseCALCUL contenant le résultat du calcul demandé par la requête reçue.

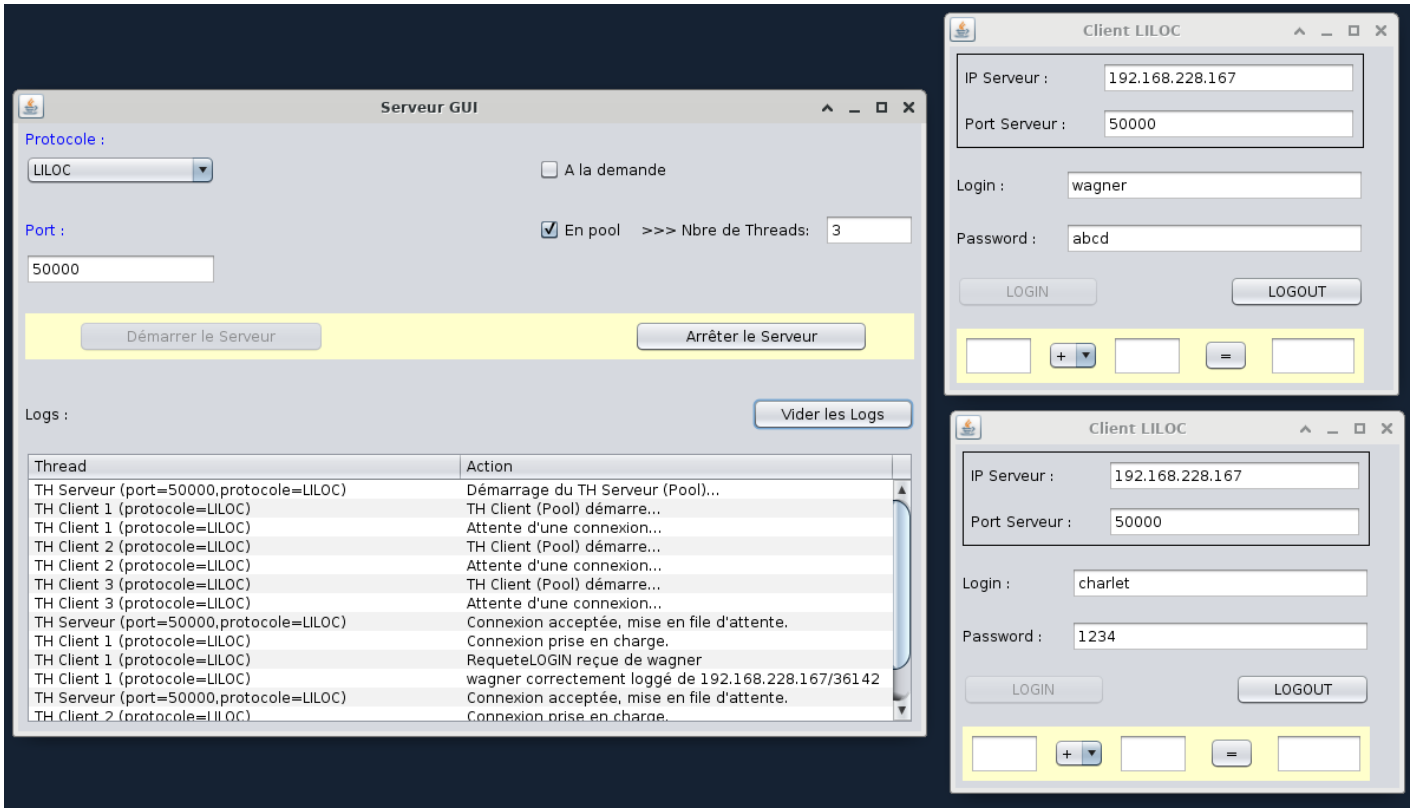
Le protocole est à présent complétement défini, il reste maintenant à instancier un objet de la classe LILOC et de le fournir à une instance de notre serveur générique.

Dans cet exemple, serveur et client vont être embarqués dans des fenêtres graphiques Swing. Voici une capture d'exécution dans le cas du modèle « **à la demande** » :



Dans la console du serveur (le « logger »), on voit la création d'un thread client à chaque nouvelle connexion.

Voici une capture correspondant au modèle « **en pool** » :



Ici, on voit clairement la création du pool de threads (ici 3) lors du lancement du serveur. Chaque nouvelle connexion est alors prise en charge par un des threads du pool.

Le code de la classe correspondant au **serveur** (fichier **ServeurLILOC.java**) est

```
import ProtocoleLILOC.*;
import ServeurGeneriqueTCP.*;
import java.io.IOException;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

public class ServeurLILOC extends javax.swing.JFrame implements Logger
{
    ThreadServeur threadServeur;

    public ServeurLILOC()
    {
        initComponents();

        threadServeur = null;
    }

    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {
        …
    }// </editor-fold>
```

```
    private void jButtonStartActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            Protocole protocole = null;
            if (((String)jComboBoxProtocole.getSelectedItem()).equals("LILOC"))
                protocole = new LILOC(this);

            int port = Integer.parseInt(jTextFieldPort.getText());

            if (jCheckBoxDemande.isSelected())
                threadServeur = new ThreadServeurDemande(port,protocole,this);

            if (jCheckBoxPool.isSelected())
            {
              int taillePool = Integer.parseInt(jTextFieldTaillePool.getText());
              threadServeur = new ThreadServeurPool(port,protocole,taillePool,this);
            }

            videLogs();
            threadServeur.start();

            jButtonStart.setEnabled(false);
            jButtonStop.setEnabled(true);
        }
        catch (NumberFormatException ex)
        {
            JOptionPane.showMessageDialog(this,"Erreur de Port et/ou taille Pool
!","Erreur...",JOptionPane.ERROR_MESSAGE);
        }
        catch (IOException ex)
        {
            JOptionPane.showMessageDialog(this,"Erreur I/O
!","Erreur...",JOptionPane.ERROR_MESSAGE);
        }
    }

    private void jButtonStopActionPerformed(java.awt.event.ActionEvent evt) {
        threadServeur.interrupt();

        jButtonStart.setEnabled(true);
        jButtonStop.setEnabled(false);
    }

    private void jButtonViderLogsActionPerformed(java.awt.event.ActionEvent evt) {
        videLogs();
    }

    public static void main(String args[]) {
        …

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new ServeurLILOC().setVisible(true);
            }
```

```
        });
    }

    // Variables declaration - do not modify
    private javax.swing.ButtonGroup buttonGroup1;
    private javax.swing.JButton jButtonStart;
    …
    // End of variables declaration

    @Override
    public void Trace(String message)
    {
        DefaultTableModel modele = (DefaultTableModel) jTableLogs.getModel();
        Vector<String> ligne = new Vector<>();
        ligne.add(Thread.currentThread().getName());
        ligne.add(message);
        modele.insertRow(modele.getRowCount(),ligne);
    }

    private void videLogs()
    {
        DefaultTableModel modele = (DefaultTableModel) jTableLogs.getModel();
        modele.setRowCount(0);
    }
}
```

On remarque que

- Le thread Serveur est instanciée et lancé au moment du clic sur le bouton « Démarrer le serveur ».
- La **JFrame** s'enregistre elle-même en tant que **Logger** pour le serveur générique et le protocole. Elle implémente donc la méthode **Trace** qui affiche les logs dans une **JTable** : à gauche le nom du thread, à droite le message du thread en question.
- Un clic sur le bouton « Arrêter Serveur » interrompt le thread Serveur qui, dans le cas du modèle en pool, arrêtera lui-même les threads du pool qui sont en attente sur le moniteur.
- La **JFrame** elle-même ne sait pour quel protocole elle va travailler → elle instancie un objet « protocole » et le passe en paramètre au serveur générique → on pourrait imaginer qu'il y ait plusieurs protocoles disponibles dans la combobox de la fenêtre et même de lancer plusieurs threads serveurs sur des ports différents et des protocoles différents.

Le code de la classe correspondant au **client** (fichier **ClientLILOC.java**) est

```
import ProtocoleLILOC.*;
import java.io.*;
import java.net.Socket;
import javax.swing.JOptionPane;

public class ClientLILOC extends javax.swing.JFrame
{
    private Socket socket;

    private String login;

    private ObjectOutputStream oos;
    private ObjectInputStream ois;

    public ClientLILOC() {
        initComponents();

        oos = null;
        ois = null;
    }

    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {
        …
    }// </editor-fold>

    private void jButtonLoginActionPerformed(java.awt.event.ActionEvent evt) {
        String ipServeur = jTextFieldIP.getText();
        int portServeur = Integer.parseInt(jTextFieldPort.getText());
        String login = jTextFieldLogin.getText();
        String password = jTextFieldPassword.getText();

        try
        {
            socket = new Socket(ipServeur,portServeur);
            RequeteLOGIN requete = new RequeteLOGIN(login,password);
            oos = new ObjectOutputStream(socket.getOutputStream());
            ois = new ObjectInputStream(socket.getInputStream());
            oos.writeObject(requete);
            ReponseLOGIN reponse = (ReponseLOGIN) ois.readObject();
            if (reponse.isValide())
            {
                jButtonLogin.setEnabled(false);
                jButtonLogout.setEnabled(true);
                jButtonCalcul.setEnabled(true);
                this.login = login;
            }
            else
            {
                JOptionPane.showMessageDialog(this,"Erreur de login
!","Erreur...",JOptionPane.ERROR_MESSAGE);
                socket.close();
```

```
            }
        }
        catch (IOException | ClassNotFoundException ex)
        {
            JOptionPane.showMessageDialog(this,"Problème de connexion
!","Erreur...",JOptionPane.ERROR_MESSAGE);
        }
    }

    private void jButtonLogoutActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            RequeteLOGOUT requete = new RequeteLOGOUT(login);
            oos.writeObject(requete);
            oos.close();
            ois.close();
            socket.close();
            jButtonLogin.setEnabled(true);
            jButtonLogout.setEnabled(false);
            jButtonCalcul.setEnabled(false);
        }
        catch (IOException ex)
        {
            JOptionPane.showMessageDialog(this,"Problème de connexion
!","Erreur...",JOptionPane.ERROR_MESSAGE);
        }
    }

    private void jButtonCalculActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            char operation = ((String)jComboBox1.getSelectedItem()).charAt(0);
            int op1 = Integer.parseInt(jTextFieldOp1.getText());
            int op2 = Integer.parseInt(jTextFieldOp2.getText());

            RequeteCALCUL requete = new RequeteCALCUL(operation,op1,op2);
            oos.writeObject(requete);
            ReponseCALCUL reponse = (ReponseCALCUL) ois.readObject();
            jTextFieldResultat.setText(String.valueOf(reponse.getResultat()));
        }
        catch (IOException | ClassNotFoundException ex)
        {
            JOptionPane.showMessageDialog(this,"Problème de connexion
!","Erreur...",JOptionPane.ERROR_MESSAGE);
        }
    }

    public static void main(String args[]) {
        …
        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new ClientLILOC().setVisible(true);
            }
        });
    }
```

```
    // Variables declaration - do not modify
    private javax.swing.JButton jButtonCalcul;
    …
    // End of variables declaration
}
```

On constate que

- La **socket** (et donc la connexion) est créée au moment du clic sur le bouton « LOGIN ». C'est cette socket qui est utilisée tout le long jusqu'au moment du clic sur le bouton « LOGOUT » → la connexion reste donc établie tout du long
- La classe **ClientLILOC** doit elle-même gérer les flux d'entrée et de sortie → rien n'est prévu à ce niveau dans le serveur générique

Remarque importante

Lorsque le serveur est une classe graphique, il est nécessaire que **l'attente sur la SocketServer** soir réalisée par un thread dédié à cet effet (comme c'est le cas dans l'exemple ci-dessus → cette attente est à la charge du thread Serveur). Si la socket était créée et mise en attente sur un accept dans une méthode de la **JFrame** du serveur, c'est **thread « dispatcher »** qui serait bloqué sur l'attente de l'accept → toutes les actions (clics, touches de clavier, …) sur la fenêtre seraient alors bloquées tant que l'accept de la socket ne serait pas réalisé.