# Les threads dans la programmation de socket en Java

## TP 4

Master 2 STIC

Module : Les applications réparties

Département Informatique

Université de Guelma

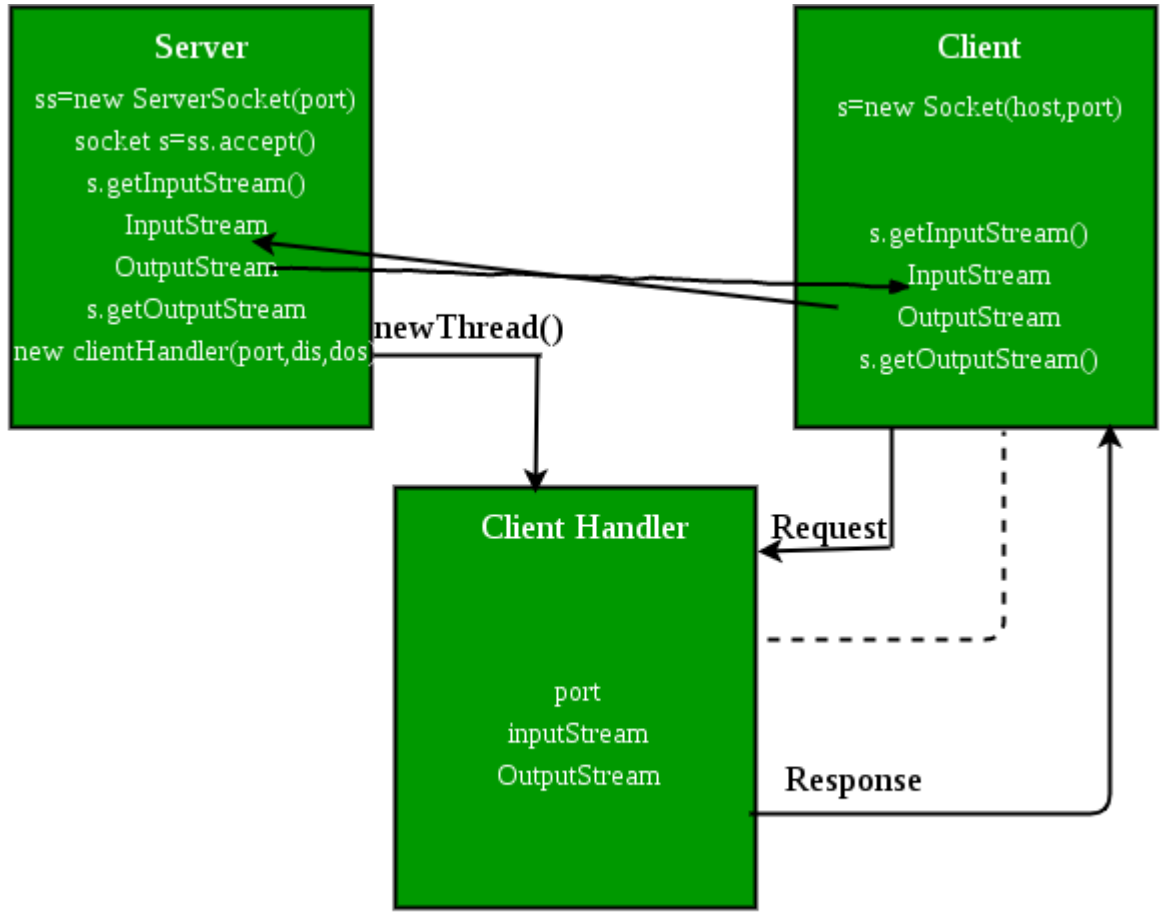
2021/2021

Dr. M. A. FERRAG (ferrag.mohamedamine@univ-guelma.dz)

# Pourquoi utiliser des threads dans la programmation réseau?

- La raison est simple, nous ne voulons pas qu’un seul client se connecte au serveur à un moment donné, mais plusieurs clients simultanément.
- Nous voulons que notre architecture prenne en charge plusieurs clients en même temps.
- Pour cette raison, nous devons utiliser des threads côté serveur afin que chaque fois qu'une demande client arrive, un thread distinct puisse être attribué pour gérer chaque demande.

- Nous allons créer deux fichiers java, Server.java et Client.java.
- Le fichier serveur contient deux classes à savoir Server (classe publique pour la création de serveur) et ClientHandler (pour gérer tout client utilisant le multithreading).
- Le fichier client ne contient qu'une seule classe publique Client (pour créer un client).



# Classe de serveur

- Classe de serveur: Les étapes impliquées côté serveur sont similaires à l'article Programmation de socket en Java avec une légère modification pour créer l'objet thread après avoir obtenu les flux et le numéro de port.

1- Etablissement de la connexion: l'objet socket serveur est initialisé et à l'intérieur d'une boucle while, un objet socket accepte en permanence la connexion entrante.

2- Obtention des Streams: L'objet inputstream et l'objet outputstream sont extraits de l'objet socket des requêtes en cours.

3-Création d'un objet gestionnaire: Après avoir obtenu les flux et le numéro de port, un nouvel objet clientHandler est créé avec ces paramètres.

4- Appel de la méthode start (): La méthode start () est invoquée sur cet objet thread nouvellement créé.

# Classe ClientHandler

- Classe ClientHandler: Comme nous utiliserons des threads séparés pour chaque requête, comprenons le fonctionnement et l'implémentation de la classe ClientHandler étendant les Threads. Un objet de cette classe sera instancié à chaque fois qu'une requête arrive.

1- Tout d'abord, cette classe étend Thread afin que ses objets assume toutes les propriétés de Threads.

2- Deuxièmement, le constructeur de cette classe prend trois paramètres, qui peuvent identifier de manière unique toute requête entrante, c'est-à-dire un Socket, un DataInputStream à lire et un DataOutputStream à écrire. Chaque fois que nous recevons une demande du client, le serveur extrait son numéro de port, l'objet DataInputStream et l'objet DataOutputStream et crée un nouvel objet thread de cette classe et appelle la méthode start () sur celui-ci.

- Remarque: chaque requête aura toujours un triplet de socket, de flux d'entrée et de flux de sortie. Cela garantit que chaque objet de cette classe écrit sur un flux spécifique plutôt que sur plusieurs flux.

3- Dans la méthode run () de cette classe, elle effectue trois opérations: demander à l'utilisateur de spécifier si l'heure ou la date est nécessaire, lire la réponse à partir de l'objet de flux d'entrée et en conséquence écrire la sortie sur l'objet de flux de sortie.

```java
// Java implementation of  Server side
// It contains two classes : Server and ClientHandler
// Save file as Server.java

import java.io.*;
import java.text.*;
import java.util.*;
import java.net.*;

// Server class
public class Server
{
    public static void main(String[] args) throws IOException
    {
        // server is listening on port 5056
        ServerSocket ss = new ServerSocket(5056);

        // running infinite loop for getting
        // client request
        while (true)
        {
            Socket s = null;
            try
            {
                // socket object to receive incoming client requests
                s = ss.accept();
                System.out.println("A new client is connected : " + s);
                // obtaining input and out streams
                DataInputStream dis = new DataInputStream(s.getInputStream());
                DataOutputStream dos = new DataOutputStream(s.getOutputStream());

                System.out.println("Assigning new thread for this client");

                // create a new thread object
                Thread t = new ClientHandler(s, dis, dos);

                // Invoking the start() method
                t.start();
            }
            catch (Exception e){
                s.close();
                e.printStackTrace();
            }
        }
    }
}
```

```java
// ClientHandler class
class ClientHandler extends Thread
{
    DateFormat fordate = new SimpleDateFormat("yyyy/MM/dd");
    DateFormat fortime = new SimpleDateFormat("hh:mm:ss");
    final DataInputStream dis;
    final DataOutputStream dos;
    final Socket s;


    // Constructor
    public ClientHandler(Socket s, DataInputStream dis, DataOutputStream dos)
    {
        this.s = s;
        this.dis = dis;
        this.dos = dos;
    }

    @Override
    public void run()
    {
        String received;
        String toreturn;
        while (true)
        {
            try {

                // Ask user what he wants
                dos.writeUTF("What do you want?[Date | Time]..\n"+
                            "Type Exit to terminate connection.");

                // receive the answer from client
                received = dis.readUTF();

                if(received.equals("Exit"))
                {
                    System.out.println("Client " + this.s + " sends exit...");
                    System.out.println("Closing this connection.");
                    this.s.close();
                    System.out.println("Connection closed");
                    break;
                }
```

```
// creating Date object
                Date date = new Date();

                // write on output stream based on the
                // answer from the client
                switch (received) {

                    case "Date" :
                        toreturn = fordate.format(date);
                        dos.writeUTF(toreturn);
                        break;

                    case "Time" :
                        toreturn = fortime.format(date);
                        dos.writeUTF(toreturn);
                        break;

                    default:
                        dos.writeUTF("Invalid input");
                        break;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        try
        {
            // closing resources
            this.dis.close();
            this.dos.close();

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

# Programmation côté client (Client.java)

- La programmation côté client est similaire à celle du programme de programmation de socket général avec les étapes suivantes:

- Établir une connexion de socket
- la communication

```java
// Save file as Client.java

import java.io.*;
import java.net.*;
import java.util.Scanner;

// Client class
public class Client
{
    public static void main(String[] args) throws IOException
    {
        try
        {
            Scanner scn = new Scanner(System.in);

            // getting localhost ip
            InetAddress ip = InetAddress.getByName("localhost");

            // establish the connection with server port 5056
            Socket s = new Socket(ip, 5056);

            // obtaining input and out streams
            DataInputStream dis = new DataInputStream(s.getInputStream());
            DataOutputStream dos = new DataOutputStream(s.getOutputStream());

            // the following loop performs the exchange of
            // information between client and client handler
            while (true)
            {
                System.out.println(dis.readUTF());
                String tosend = scn.nextLine();
                dos.writeUTF(tosend);

                // If client sends exit,close this connection
                // and then break from the while loop
                if(tosend.equals("Exit"))
                {
                    System.out.println("Closing this connection : " + s);
                    s.close();
                    System.out.println("Connection closed");
                    break;
                }

                // printing date or time as requested by client
                String received = dis.readUTF();
                System.out.println(received);
            }

            // closing resources
            scn.close();
            dis.close();
            dos.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
```

# Comment ces programmes fonctionnent-ils ensemble?

1- Lorsqu'un client, disons client1, envoie une demande de connexion au serveur, le serveur attribue un nouveau thread pour gérer cette demande. Le thread nouvellement attribué a accès aux flux pour communiquer avec le client.

2- Après avoir assigné le nouveau thread, le serveur via sa boucle while, entre à nouveau dans l'état d'acceptation.

3- Lorsqu'une deuxième requête arrive alors que la première est toujours en cours, le serveur accepte cette requête et attribue à nouveau un nouveau thread pour la traiter. De cette manière, plusieurs demandes peuvent être traitées même lorsque certaines demandes sont en cours de traitement.

# Comment tester le programme ci-dessus sur votre système?

- Enregistrez les deux programmes dans le même package ou n'importe où. Ensuite, exécutez d'abord Server.java suivi de Client.java.
- Vous pouvez soit copier le programme client dans deux trois fichiers distincts et les exécuter individuellement, soit si vous avez un IDE comme eclipse, exécuter plusieurs instances à partir du même programme.