Création d'un système de communication Client-Serveur basé sur les Sockets

PROGRAMMATION RESEAU EN JAVA

Eya ZAOUI

Rapport de Projet Téléinformatique | RT 2–1

INSAT 2020

Table des matières

Introduction

La notion de sockets a été introduite dans les distributions de Berkeley (un système UNIX) dans les années 80. On parle parfois de sockets BSD (Berkeley Software Distribution).

Dans le contexte des logiciels, les sockets sont des « connecteurs réseau » ou des « interfaces de connexion » . C'est un mécanisme universel de bas niveau : il permet l'utilisation directe du transport réseau (construit sur TCP ou UDP), depuis tout langage de programmation. Sur cette couche logicielle, le développeur requiert des services en appelant des fonctions, et elle masque le travail nécessaire de gestion du réseau, pris en charge par le système.

Nous allons utiliser les Sockets Java pour établir une connexion entre un programme client et un programme serveur. Le serveur gère une base de données SQL.
Nous voulons utiliser l'application cliente pour envoyer un ID d'un étudiant au serveur et récupérer ses données de la BD.

Etude théorique

La communication Client-Serveur est un modèle de communication où une application cliente envoie une requête à un serveur, qui traite cette requête selon un service précis, puis envoie une réponse au client.

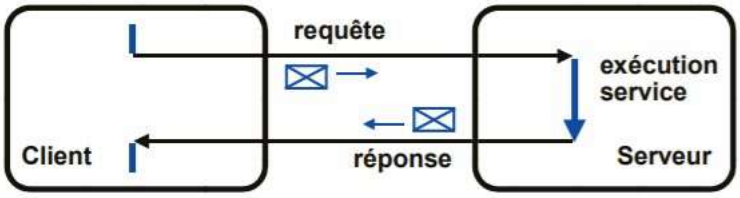
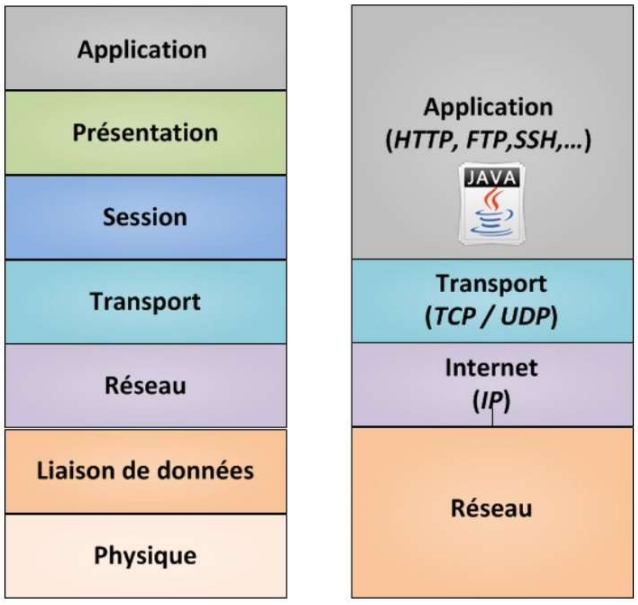


Figure 1- Modèle Client-Serveur

Cette communication peut se faire via les Sockets.
Les sockets forment un mécanisme permettant la communication inter-processus (IPC) sur une même machine ou la communication à travers un réseau TCP/IP.
Les sockets définissent l'interface entre la couche application et la couche transport du modèle TCP/IP.



La couche de Transport requiert la définition au moins d'un port et d'un mode de communication.
Les sockets étant Full-Duplex (flux entrée-sortie) , il suffit de déterminer un port sur lequel le client et le serveur vont communiquer. Celui-ci ne doit pas se confondre avec d'autre ports, comme les ports connus ou enregistrés.

Il existe 2 modes de communications :

- Mode connecté : Protocole TCP
  Ce mode garantie l'ordre, la fiabilité et le contrôle de flux de données. Une connexion durable est établie entre les deux processus. La communication se fait par : Ouverture d'une liaison, suite d'échanges séquentielle, fermeture de la liaison. Si un paquet est perdu, sa transmission est relancée. Le serveur préserve son état entre deux requêtes (toujours connecté).

- Mode déconnecté : Protocole UDP
  Ce mode n'a pas de garantie particulière. Les requêtes successives sont indépendantes. Il n'y aucune liaison établie et il n'y a pas de préservation de l'état entre les requêtes (Le serveur peut être déconnecté). Donc, pour recevoir des réponses, le client doit indiquer son adresse à chaque requête. En fait, il émet des datagrammes : paquets de données chacune comprenant son adresse et sa requête. Puisque le serveur ne connait pas si un client essaie de lui communiquer, si un paquet est perdu, il ne peut pas être récupéré.

■ Points communs : Le client a l'initiative de la communication ; le serveur doit être à l'écoute. Le client doit également connaître l'adresse IP et le numéro de port du serveur. Le serveur peut servir plusieurs clients (1 thread unique ou 1 thread par client).

Dans ce projet, nous traitons la programmation des Sockets en Java. Nous adoptons la communication Client-Serveur en mode connecté (TCP). Par ailleurs, nous ne considérons qu'un seul client. Il n'y aura pas de multi-threading pour plusieurs clients.
Le mode de gestion de données choisie est itératif : le serveur traite les requêtes les unes après les autres. Le client ouvre une connexion avec le serveur avant de pouvoir envoyer des requêtes, puis ferme la connexion à la fin de la suite d'opérations. Ceci assure le traitement des pannes et le maintien de la logique d'ordre des opérations.

Description du travail

I. Conception

Notre système se compose par un programme serveur et une base de données SQL qui définit le service fournit par notre serveur. Au cotée client, nous créons deux classes : une classe Client qui établit effectivement la connexion avec le serveur et une classe ClientUI qui instancie un Client et crée une interface graphique (User interface).

Nous utilisons les outils fournis par les bibliothèques Java pour construire le système.

❖ Sockets en Java :

- La classe ServerSocket : socket côté serveur. Ce socket attend les connexions et crée un socket service client pour répondre aux requêtes du client.

- La classe Socket : sockets ordinaires, pour les échanges. Elles fournissent des classes InputStream et OutputStream pour échanger de l'information en entrée-sortie.

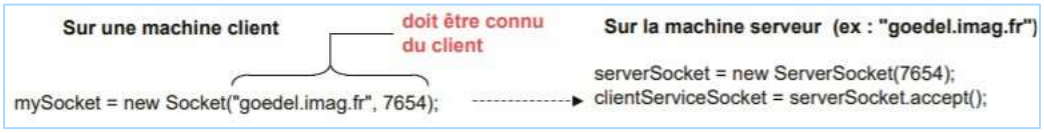


Figure 2- Socket client et Socket serveur

❖ Structure de données :

- La classe Etudiant : cette classe Java est l'image de la table Etudiant de la base de données. Le serveur utilise les objets de cette classe pour manipuler les résultats des requêtes SQL.

Etudiant
– id : int – nom : String – dateDeNaissance : String – nationalite : String – email : String – adresse : String

- La classe DatabaseManager : c'est la classe avec laquelle on a créé et géré la base de données SQLite pendant le développement. Sa méthode main() n'est pas nécessaire pour le déploiement, elle est commentée.

- La classe Server : C'est le programme serveur qui ouvre un server socket et écoute sur son port en attendant une connexion d'un client sur le même port. Elle instancie une connexion avec la base de données à l'aide du DatabaseManager.

- La classe Client : C'est le programme client qui établit une communication avec le serveur. Néanmoins, cette classe ne contient pas une méthode main(), elle est plutôt instanciée par ClientUI.

- La classe ClientUI : La fonction de cette classe est de créer une interface graphique pour l'utilisateur (User Interface). Elle instancie un objet Client qui lance la communication avec le serveur.

- La base de données « etudiants.db » : elle contient une seule table Etudiant, avec 3 entrées. Elle a été générée par la DatabaseManager.

Id	Nom	DateDeNaissance	Nationalite	Email	Adresse
1800234	Madeline Dubois	2000-06-06	Française	M.Dubois@hotmail.fr	9 rue Palestine Tunis
1800467	Eya Zaoui	1999-07-20	Tunisienne	zaouieya@gmail.com	14 rue 10469 Ariana
1800568	Ahmed Samir	1998-03-14	Tunisienne	ahmeds@yahoo.fr	2 rue Les Jasmins Manzah 1

Figure 3- La table Etudiant, (depuis phpMyAdmin)

II. Réalisation

1. Outils et langages utilisés

Le langage de développement est **Java** , JDK version 13. L'environnement utilisé est l'IDE **Eclipse.**
La base de données est développée en **SQLite** par la classe Java DatabaseManager. L'ancienne version du projet avait une BD développée sur MySQL Server.
Pour assurer une utilisation plus aisée, le projet était mis à jour en SQLite, puisqu'on n'a plus besoin d'importer la BD.

2. Présentation du système

Le déroulement du système se fait comme suit :

- Pour mettre notre serveur en marche, on exécute ServerApp.jar en ligne de commande CMD en tapant : **java -jar ServerApp.jar**. Si tout va bien, on reçoit ce résultat :

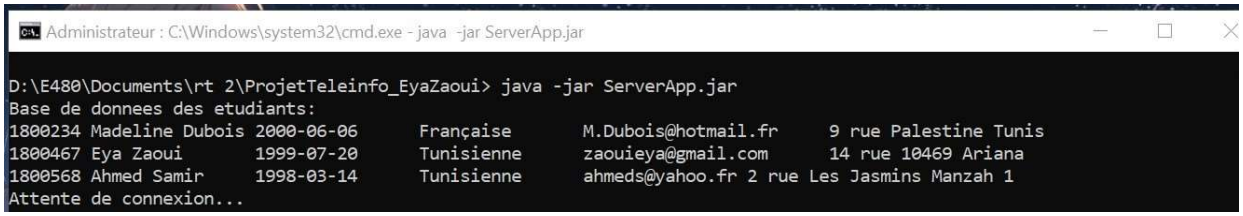


Figure 4- ServerApp.jar en marche

En fait, notre programme serveur tente en premier lieu de créer un Server Socket sur un port déclaré au début.
Si aucun problème ne survient -par exemple, port en cours d'utilisation par un autre programme- , le serveur tente d'accéder à la Base de Données(BD). Cet accès est défini dans la méthode connect() de l'instance du DatabaseManager. Cette méthode retourne

un objet de la classe Connection du paquet java.sql. On lui instancie une connexion par le DriverManager, qui prend en argument un URL décrivant quel type de JavaDataBaseConnector à utiliser (ici, c'est `jdbc:sqlite`) et le nom de la BD (`etudiants.db`). Avec ces données, le DriverManager accède à la BD. Notre objet Connection peut maintenant exécuter des requêtes SQL vers notre BD. On crée une requête pour récupérer toutes les entrées de la table Etudiant dans la méthode selectAll(), pour visualiser le contenue de la BD.
Le serveur reste sur écoute du socket en attente d'une connexion d'un client.

- On exécute ClientApp.jar pour lancer le programme client. D'abord, nous aurons cette interface graphique, développée avec les paquets Swing et AWT :

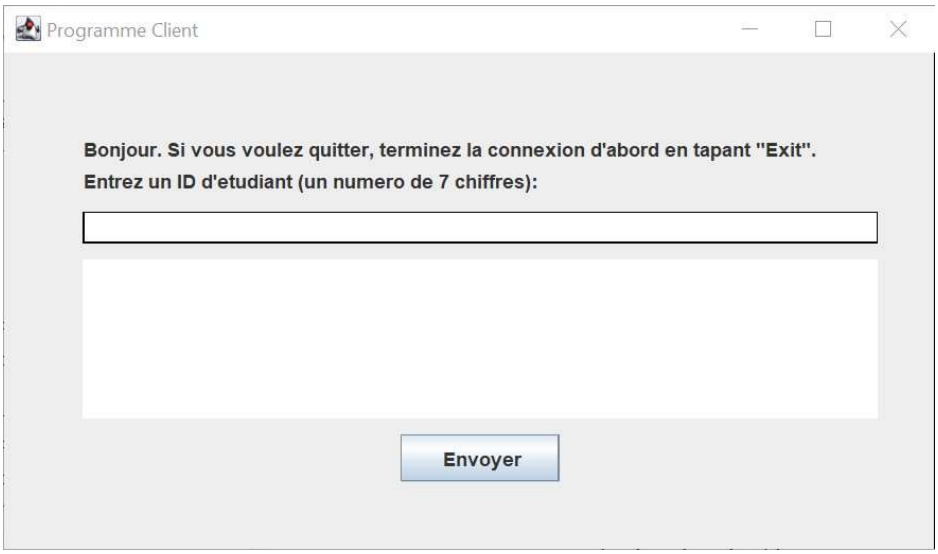


Figure 5- Démarrage de ClientApp.jar

Ensuite, ce programme instancie un objet Client, qui établit une connexion avec notre serveur. Dans le constructeur du Client on trouve :

```
//get the localhost IP address
        InetAddress serverAddress = InetAddress.getLocalHost();

 //establish socket connection to server
        socket = new Socket(serverAddress.getHostName(), port);
```

On trouve aussi l'instanciation des flux d'Entrée/Sortie (Input/Output), pour assurer l'échange de données dans le socket.

Notre serveur se rend compte de la connexion du client et instancie de sa part les flux I/O. On aura dans la console du serveur ce ligne s'ajouter :

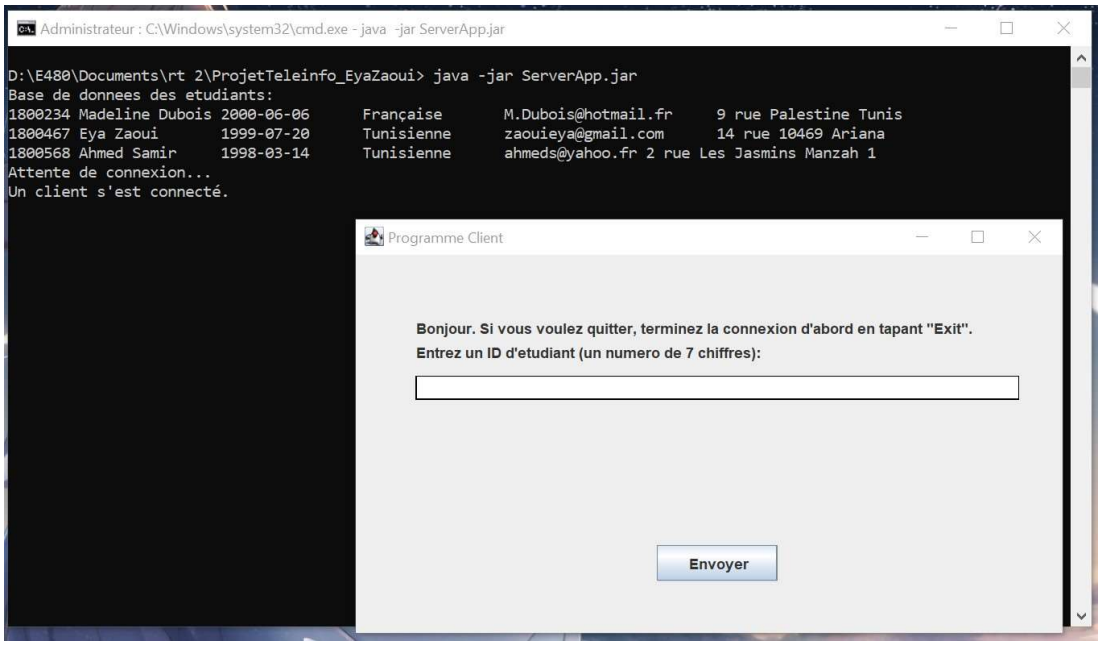


Figure 6- Server.java : un client détecté

- Le serveur entre en écoute sur socket en boucle, jusqu'à la réception de « exit » :

```
//keeps listening until receiving 'exit' call or program terminates
message = brSocket.readLine();
while (!message.equalsIgnoreCase("exit"))
```

Cela assure la possibilité d'échanger plusieurs requêtes entre le client et le serveur dans une même session, puisque le serveur ne s'éteint que lorsque le client demande de terminer la connexion : serveur itératif en mode connectée.
D'autre part, il faudrait vérifier que le client ne quitte son programme qu'après avoir entré « exit » pour terminer la session, pour que le serveur ne reste pas en marche indéfiniment. Pour cela, l'icône de fermeture du programme ClientApp n'est activé que si le client entre « exit ».
Le message « exit » entré engendre :

- La signalisation au serveur de quitter sa boucle.
- La fermeture des ressources (sockets et flux I/O) de chaque côté client-serveur.
- L'activation de l'icône de fermeture du programme de ClientApp.
- La désactivation du bouton « envoyer ».

La fin de connexion entre le client et le serveur est définit dans la méthode endConnection() du Client.

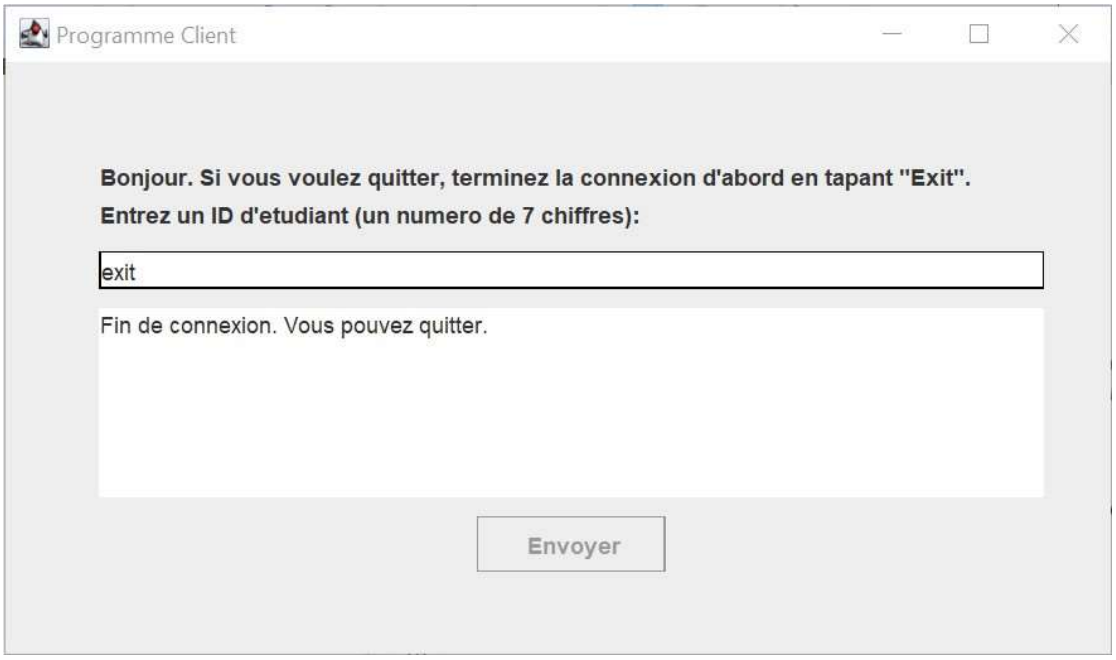


Figure 7- ClientUI: Fin de connexion

Evidemment, si l'utilisateur exécute ClientUI avant de mettre en marche le serveur, il sera signalé dans l'interface, le bouton « envoyer » sera désactivé, et l'icône de fermeture sera accessible.

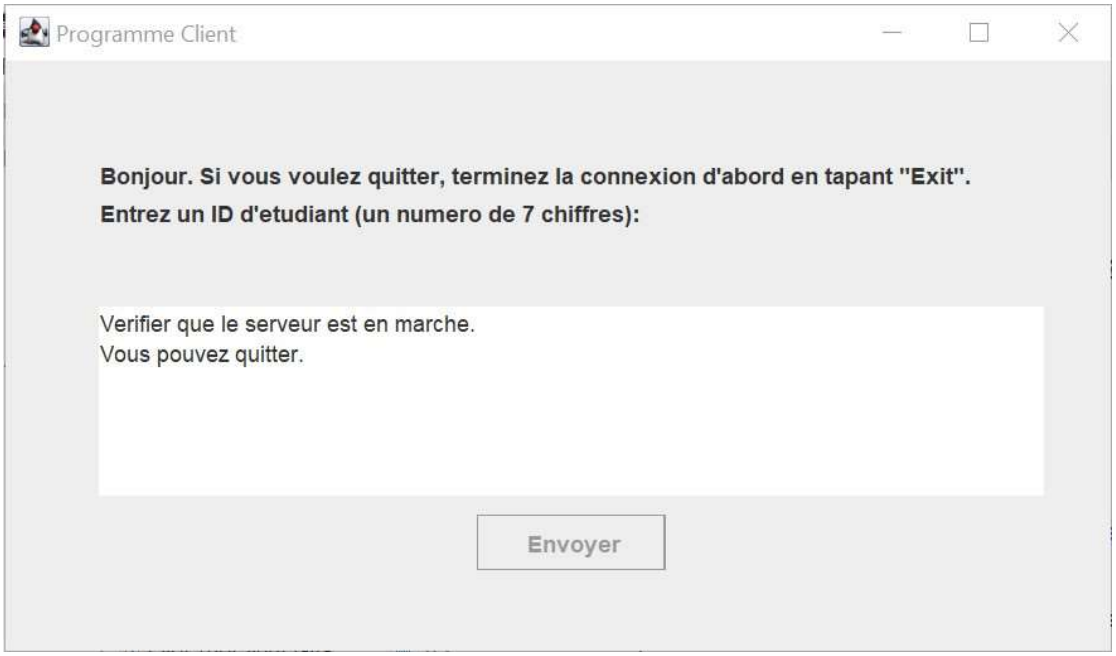


Figure 8- Client UI: connexion refusée, serveur arrêté

- La connexion entre le client et le serveur est établie. Il suffit d'entrer un ID d'étudiant valide pour voir le résultat de la BD :

Figure 10- Résultat côté client

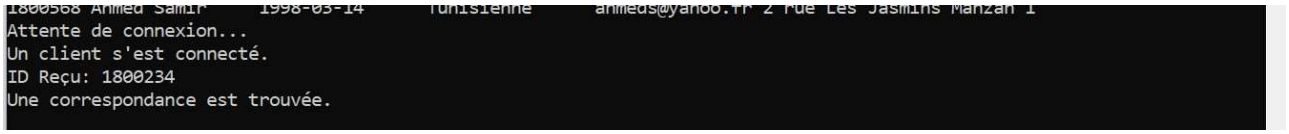


Figure 9- Résultat côté serveur

- La validité du format de l'entrée de données est traitée dans la méthode actionPerformed(ActionEvent) de la classe ClientUI :

```
//defining send Button action:
@Override
public void actionPerformed(ActionEvent e) {

//Handle input format:
	int id=0;
	String request = tf.getText();
	String resultStr = "";

	if (!request.equalsIgnoreCase("exit")) {
		if (request == null || request.length() != 7) {
			txt.setText("Requete invalide.");
		} else {
			try {
				id = Integer.parseInt(request);
			} catch (NumberFormatException ex) {
				txt.setText("Requete invalide.");
				return;
			}
			//If valid, write to socket
```

On assure que la donnée entrée est : soit « exit » (Pour signaler au serveur de terminer la connexion), soit un nombre de 7 chiffres. Sinon, la donnée n'est pas envoyée dans le socket et l'utilisateur est signalé que sa requête est invalide.

Quand l'utilisateur entre un nombre de 7 chiffres, celui-ci est envoyé dans le socket vers le serveur. Le serveur recherche un IDetudiant correspondant à ce nombre dans la BD. S'il trouve une entité correspondante, il instancie avec elle un objet Etudiant et l'envoie dans le socket -au client- avec tous ses attributs et ses valeurs. Sinon, il envoie qu'aucun résultat n'est trouvé pour cet IDetudiant :

```
// look for student by ID number
System.out.println("ID Reçu: " + message);
sql = "select * from etudiant where id = "+message;
mserver.statement = mserver.con.prepareStatement(sql);
mserver.result = mserver.statement.executeQuery();

if(mserver.result.next()) {

	System.out.println("Une correspondance est trouvée.");
	et = new Etudiant(mserver.result.getInt(1),mserver.result.getString(2),
		mserver.result.getDate(3).toString(),mserver.result.getString(4),
		mserver.result.getString(5),mserver.result.getString(6));
	response = et.toString();

} else {
	response = "Pas de resultat.";
	System.out.println(response);
	}

pwSocket.println(response);
pwSocket.flush();
```

Conclusion

Nous avons créé un programme client et un programme serveur en Java qui communique en mode TCP via les Sockets. Le serveur reste en écoute sur le port du socket pour une connexion client. Le client établit une connexion avec le serveur, envoi ses requêtes, reçoit des réponses du serveur puis ferme la connexion.
A la réception d'une requête, le serveur accède à une base de données SQL pour récupérer les informations correspondantes à un ID étudiant fourni par le client.

Nous avons utilisé les bibliothèques java.net pour l'implémentation des sockets, java.sql pour la communication avec la BD et Swing et AWT pour l'interface graphique.

Bibliographie

[**S. Krakowiak, 2003-2004**] *Programmation client-serveur sockets – RPC*, Université Joseph Fourier, France. Lien : http://lig-membres.imag.fr/krakowia/.
Plusieurs tutoriels sur https://www.javatpoint.com/java-tutorial.

Annexe

I. Modes d'emploi

- Assurez-vous de lancer le programme serveur en premier. Pour cela, nous recommandons de le lancer via la ligne de commande CMD par:

```
java -jar ServerApp.jar
```

Ceci vous permet de voir des commentaires du coté serveur en CMD, comme le contenu de la BD. Sinon, le serveur se met en marche par simple clic sur le fichier ServerApp.jar, vous n'aurez aucun signal.

- Cliquez sur ClientApp.jar pour lancer le programme client. Vous aurez un message sur l'interface si le serveur n'est pas en marche, sinon tout va bien.

II. Listing du code source

1. Client.Java :

```
public class Client {
    private static int port = 9876; //same port as server program.

    public PrintWriter pwSocket = null;
    public BufferedReader brSocket = null;
    private Socket socket = null;

    public Client() throws IOException{
       //get the localhost IP address
       InetAddress serverAddress = InetAddress.getLocalHost();

     //establish socket connection to server
       socket = new Socket(serverAddress.getHostName(), port);


      //instantiating streams
       pwSocket = new PrintWriter(socket.getOutputStream());
       brSocket = new BufferedReader(new InputStreamReader(socket
                                                    .getInputStream()));
    }

   public void endConnection() throws IOException {
       //write to socket "Exit"
       pwSocket.println("Exit");
       pwSocket.flush();
       try {
              Thread.sleep(1000); //ensuring server reception
       } catch (InterruptedException e) {
              System.out.println(e);
              }

     //close resources
              brSocket.close();
              pwSocket.close();
              socket.close();
   }


}
```

2. ClientUI.Java :

```
public class ClientUI extends JFrame implements ActionListener {

//UI Preparation
private static final long serialVersionUID = 1L;

JButton sendBtn;
JLabel l1,l2;
JTextArea txt;
JTextField tf;
Client client;

ClientUI(String s){
	super(s);
	setLayout(null);
	setVisible(true);
	setSize(600,350);
	this.setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
	sendBtn = new JButton("Envoyer");
	sendBtn.setBounds(250,240,100,30);
	sendBtn.addActionListener(this);
	add(sendBtn);
	l1 = new JLabel("Bonjour. Si vous voulez quitter, terminez la connexion
d'abord en tapant \"Exit\". ");
	l1.setBounds(50, 50,500,20 );
	l2= new JLabel("Entrez un ID d'etudiant (un numero de 7 chiffres):");
	l2.setBounds(50, 70, 500, 20);
	add(l1); add(l2);
	txt = new JTextArea();
	txt.setBounds(50, 130, 500, 100);
	txt.setWrapStyleWord(true);
	txt.setLineWrap(true);
	txt.setEditable(false);
	add(txt);
	tf=new JTextField();
	tf.setBounds(50,100,500,20);
	tf.setBorder(BorderFactory.createLineBorder(Color.black));
	add(tf);

//creating a Client and connecting to Server
	try {
		client = new Client();
	} catch (IOException e) {
		txt.setText("Verifier que le serveur est en marche.\nVous pouvez
quitter.");
		System.out.println(e);
		sendBtn.setEnabled(false);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
			}
		}

public static void main(String[] args) {

		new ClientUI("Programme Client"); //launching program

		}
```

```
//defining send Button action:
@Override
public void actionPerformed(ActionEvent e) {

//Handle input format:
	int id=0;
	String request = tf.getText();
	String resultStr = "";
	if (!request.equalsIgnoreCase("exit")) {

		if (request == null || request.length() != 7) {

			resultStr = "Requete invalide.";

		} else {

			try {
				id = Integer.parseInt(request);
			} catch (NumberFormatException ex) {
				txt.setText("Requete invalide.");
				return;
					}

			//If valid, write to socket :
			client.pwSocket.println(request);
			client.pwSocket.flush();

			//Read the server response :
			try {
				resultStr = client.brSocket.readLine();
			} catch (IOException e1) {
				System.out.println(e1);
				txt.setText("Une erreur est survenue.");
				return;
					}
			}

	txt.setText("Resultat: "+ resultStr);

		} else {
			//Closing connection:
			txt.setText("Fin de connexion. Vous pouvez quitter.");
			try {
				client.endConnection();

			} catch (IOException e1) {
				System.out.println(e1);
				txt.setText("Une erreur est survenue.");
			    }

	setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	sendBtn.setEnabled(false);

			}
		}
}
```

3. Etudiant.java :

```
public class Etudiant {

	private final int id;
	private final String nom;
	private final String dateDeNaissance;
	private final String nationalite;
	private final String email;
	private final String adresse;

	public Etudiant(int id, String nom, String dateDeNaissance, String
nationalite, String email, String adresse) {
		this.id = id;
		this.nom = nom;
		this.dateDeNaissance = dateDeNaissance;
		this.nationalite = nationalite;
		this.email = email;
		this.adresse = adresse;
	}

	public int getId() {
		return id;
	}
	public String getNom() {
		return nom;
	}
	public String getAge() {
		return dateDeNaissance;
	}
	public String getNationalite() {
		return nationalite;
	}
	public String getEmail() {
		return email;
	}
	public String getAdresse() {
		return adresse;
	}

	@Override
	public String toString() {
		return "Etudiant [id=" + id + ", nom=" + nom + ", date de naissance="
+ dateDeNaissance + ", nationalite=" + nationalite + ", email=" + email + ",
adresse=" + adresse+"]";
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Etudiant other = (Etudiant) obj;
		if (id != other.id)
			return false;
		return true;
	}
}
```

4. DatabaseManager.java

```
public class DatabaseManager {
/*
* Cette methode main() a servi pendant le developpement pour la creation de la
BD SQLite.
* Elle n'est pas necessaire pour le deploiement.

public static void main(String[] args) {

	DatabaseManager db = new DatabaseManager();
	db.createDB();
	db.createNewTable();
	db.insert();
	db.selectAll(); }   */

// SQLite connection string
	public static String url = "jdbc:sqlite:etudiants.db";

	private void createDB() {
		Connection conn = null;
		try {
		     conn = DriverManager.getConnection(url);
		    if (conn != null) {
		        DatabaseMetaData meta = conn.getMetaData();
		        System.out.println("A new database has been created.");
		    }
		} catch (SQLException e) {
		    System.out.println(e.getMessage());
		}
	    }


	private void createNewTable() {
		String sql = "CREATE TABLE IF NOT EXISTS etudiant ("
			+ "Id integer PRIMARY KEY NOT NULL,"
			+ "Nom text NOT NULL,"
			+ "DateDeNaissance text NOT NULL,"
			+ " Nationalite text NOT NULL,"
			+ " Email text NOT NULL,"
			+ " Adresse text NOT NULL"
			+ ");";

		try (Connection conn = DriverManager.getConnection(url);
			Statement stmt = conn.createStatement()) {
		    // create a new table
		    stmt.execute(sql);
		} catch (SQLException e) {
		    System.out.println(e.getMessage());
		}
	    }
```

```
private void insert() {
    String sql = "INSERT INTO `etudiant` VALUES" +
"(1800234, 'Madeline Dubois', '2000-06-06', 'Française', 'M.Dubois@hotmail.fr',
 '9 rue Palestine Tunis')," +
"(1800467, 'Eya Zaoui', '1999-07-20', 'Tunisienne', 'zaouieya@gmail.com',
 '14 rue 10469 Ariana')," +
"(1800568, 'Ahmed Samir', '1998-03-14', 'Tunisienne', 'ahmeds@yahoo.fr',
 '2 rue Les Jasmins Manzah 1');";
     try (Connection conn = DriverManager.getConnection(url);
          PreparedStatement pstmt = conn.prepareStatement(sql)) {
                pstmt.executeUpdate();
            } catch (SQLException e) {
                System.out.println(e.getMessage());
            }
          }

public Connection connect() throws SQLException {

              return  DriverManager.getConnection(url);

          }

public void selectAll() throws SQLException{
      String sql = "SELECT * FROM etudiant";

              Connection conn = DriverManager.getConnection(url);
              Statement stmt  = conn.createStatement();
              ResultSet rs    = stmt.executeQuery(sql);

// loop through the result set
   while (rs.next()) {
        System.out.println(rs.getInt("Id")+ "\t" +  rs.getString("Nom")+"\t" +
                           rs.getString(3) + "\t" + rs.getString(4) + "\t" +
                           rs.getString(5) + "\t" + rs.getString(6));
                }

          }


}
```

5. Server.java :

```
public class Server  {

    //static ServerSocket variable
    private static ServerSocket serverSock;
    //socket server port on which it will listen
    private static int port = 9876;

    private static Connection con;
    private static PreparedStatement statement;
    private static ResultSet result;
    private static DatabaseManager dbManager = new DatabaseManager();

    public static void main(String args[]) throws IOException, SQLException {
      String message = "";
      String response ="";
      String sql = "";
        PrintWriter pwSocket = null;
        BufferedReader brSocket = null;
        Etudiant et = null ;

        //create the server socket and client socket objects
        serverSock = new ServerSocket(port);
        Socket clientSocket = null;

        //initiating database connection
        System.out.println("Base de donnees des etudiants:");
        try {
                    con = dbManager.connect();
                    dbManager.selectAll();

             }  catch (SQLException e1) {
                 System.out.println("Echec de connexion a la base de donnee.");
                 System.out.println(e1);
                 return;
             }

//waiting for client connection
       System.out.println("Attente de connexion...");
        clientSocket = serverSock.accept();
        System.out.println("Un client s'est connecté.");

//instantiating streams
try {
  pwSocket = new PrintWriter(clientSocket.getOutputStream());
  brSocket = new BufferedReader(newInputStreamReader(clientSocket
                                                    .getInputStream()));
      } catch (IOException e) {
                System.out.println(e);
                return;
      }

  //keeps listening until receiving 'exit' call or program terminates
      message = brSocket.readLine();
       while (!message.equalsIgnoreCase("exit")) {

     // look for student by ID number
      System.out.println("ID Reçu: " + message);
      sql = "select * from etudiant where id = "+message;
      statement = con.prepareStatement(sql);
```

```
result = statement.executeQuery();

if(result.next()) {

	System.out.println("Une correspondance est trouvée.");
		et = new Etudiant(result.getInt(1), result.getString(2),
	result.getString(3), result.getString(4), result.getString(5),
	result.getString(6));
		response = et.toString();

	} else {
		 response = "Pas de resultat.";
		System.out.println(response);
		 }

		pwSocket.println(response);
		pwSocket.flush();

		message = brSocket.readLine();
			}

//close the connection
	System.out.println("Arrêt du serveur.");
	brSocket.close();
	 pwSocket.close();
	 clientSocket.close();
	 serverSock.close();
    }


}
```