Programmation Réseau en Java

Objectif du module

Comprendre et mettre en œuvre la communication entre systèmes en Java.

Concrètement, cela veut dire :

- Créer des programmes capables d'envoyer/recevoir des messages.
- Comprendre les **protocoles de communication** (TCP, UDP, HTTP, WebSocket).
- Implémenter des **clients et serveurs réseau**.

1. Sockets TCP / UDP

Qu'est-ce qu'une socket ?

Une **socket** est un **point de communication** entre deux machines reliées par un réseau.
C'est comme une "prise réseau" entre un **client** et un **serveur**.

TCP vs UDP

Protocole	Type	Fiable ?	Exemple d'usage
TCP	Orienté connexion	Oui (garantie d'ordre et livraison)	HTTP, FTP, SSH
UDP	Sans connexion	Non (plus rapide mais non garanti)	Jeux, streaming, VoIP

Exemple 1 : Communication TCP (client/serveur)

Serveur TCP

```
import java.io.*;

import java.net.*;


public class ServeurTCP {

   public static void main(String[] args) throws IOException {

      ServerSocket server = new ServerSocket(5000);

      System.out.println("Serveur en attente de connexion...");

      Socket socket = server.accept();

      System.out.println("Client connecté !");


      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

```
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        String message = in.readLine();
        System.out.println("Message reçu : " + message);
        out.println("Message bien reçu !");

        socket.close();
        server.close();
    }
}
```

Client TCP

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

public class ClientTCP {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 5000);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        out.println("Bonjour serveur !");
        System.out.println("Réponse du serveur : " + in.readLine());

        socket.close();
    }
}
```

Explication :

- Le serveur attend (accept()) qu'un client se connecte.
- Le client ouvre une connexion (new Socket("localhost", 5000)).
- Les deux échangent des messages via des flux (InputStream, OutputStream).

Exemple 2 : Communication UDP

Serveur UDP

```
import java.net.*;


public class ServeurUDP {
   public static void main(String[] args) throws Exception {
      DatagramSocket socket = new DatagramSocket(4000);
      byte[] buffer = new byte[1024];
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);


      System.out.println("Serveur UDP en attente...");
      socket.receive(packet);


      String message = new String(packet.getData(), 0, packet.getLength());
      System.out.println("Message reçu : " + message);
      socket.close();
   }
}
```

Client UDP

```
import java.net.*;


public class ClientUDP {
   public static void main(String[] args) throws Exception {
      DatagramSocket socket = new DatagramSocket();
      byte[] data = "Salut serveur UDP".getBytes();
      InetAddress address = InetAddress.getByName("localhost");


      DatagramPacket packet = new DatagramPacket(data, data.length, address, 4000);
      socket.send(packet);
      socket.close();
   }
```

}

Différence clé :

- UDP n'établit **pas de connexion** : les paquets sont envoyés directement.
- Pas de garantie de réception → plus rapide mais moins sûr.

2. HTTP Client / Server avec Java 11+

Java 11 a introduit une API moderne : **java.net.http.HttpClient**, qui permet de consommer facilement des **API REST**.

Exemple : Appel d'une API REST avec HttpClient

```
import java.net.http.*;

import java.net.*;

import java.io.*;


public class HttpClientExample {

   public static void main(String[] args) throws Exception {

      HttpClient client = HttpClient.newHttpClient();

      HttpRequest request = HttpRequest.newBuilder()

            .uri(URI.create("https://jsonplaceholder.typicode.com/todos/1"))

            .build();


      HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

      System.out.println("Code HTTP: " + response.statusCode());

      System.out.println("Corps: " + response.body());

   }

}
```

Explication :

- HttpClient permet de créer un client HTTP asynchrone.
- HttpRequest définit l'URL et les en-têtes.
- HttpResponse contient le code et le contenu renvoyés.

On peut facilement faire des **POST/PUT/DELETE** pour interagir avec un serveur REST (par exemple un backend Spring Boot).

Exemple : Petit serveur HTTP avec com.sun.net.httpserver.HttpServer

```
import com.sun.net.httpserver.*;
```

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

public class SimpleHttpServer {
   public static void main(String[] args) throws IOException {
      HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
      server.createContext("/hello", exchange -> {
         String response = "Bonjour depuis le serveur HTTP Java!";
         exchange.sendResponseHeaders(200, response.length());
         OutputStream os = exchange.getResponseBody();
         os.write(response.getBytes());
         os.close();
      });
      server.start();
      System.out.println("Serveur HTTP démarré sur le port 8080...");
   }
}
```

Ici, le serveur répond à toute requête sur http://localhost:8080/hello.

3. WebSocket (Java EE / Jakarta EE)

Objectif :

Les WebSockets permettent une **communication bidirectionnelle en temps réel**.
Contrairement au HTTP classique (requête → réponse), ici :

- Le **serveur peut envoyer des messages spontanément**.
- Les deux côtés restent connectés (comme un chat ou un tableau de bord live).

Exemple : WebSocket avec Jakarta EE

Serveur WebSocket

```
import jakarta.websocket.*;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint("/chat")
```

```
public class ChatEndpoint {

    @OnOpen
    public void onOpen(Session session) {
        System.out.println("Nouvelle connexion : " + session.getId());
    }

    @OnMessage
    public void onMessage(String message, Session session) throws Exception {
        System.out.println("Message reçu : " + message);
        session.getBasicRemote().sendText("Echo: " + message);
    }

    @OnClose
    public void onClose(Session session) {
        System.out.println("Connexion fermée : " + session.getId());
    }
}
```

Ce serveur écoute sur /chat et renvoie le même message reçu.

👩‍💻 Client WebSocket (JavaScript par exemple)

```
<script>
const socket = new WebSocket("ws://localhost:8080/chat");

socket.onopen = () => {
  console.log("Connecté !");
  socket.send("Bonjour serveur WebSocket !");
};

socket.onmessage = e => {
  console.log("Message du serveur :", e.data);
```

};

</script>

Comparaison entre technologies

Technologie	Type	Sens de communication	Usage typique
TCP Socket	Bas niveau	Bidirectionnelle	Chat, transfert de fichiers
UDP Socket	Bas niveau, rapide	Unidirectionnelle	Jeux, streaming
HTTP Client/Server	Requête-Réponse	Client → Serveur	API REST, Web
WebSocket	Temps réel	Bidirectionnelle	Chat, notifications live

En résumé

Sujet	Ce qu'il faut maîtriser
TCP/UDP	Connexion client/serveur, flux d'E/S, ports
HTTP Client	Requêtes GET/POST avec HttpClient
HTTP Server	Réponses simples avec HttpServer
WebSocket	Communication bidirectionnelle en temps réel