SUITE TP ARCHITECTURE DISTRIBUEE

Exercice 1 : Communication Client-Serveur avec Sockets

Prérequis :

- IntelliJ IDEA installé.
- JDK configuré.
- Application Java classique sans nécessiter de serveur web

Étapes pour configurer dans IntelliJ IDEA :

1. Créer un nouveau projet Java :
    - Lancez IntelliJ IDEA.
    - Créez un nouveau projet Java (File -> New -> Project -> Java).
    - Sélectionnez la version de JDK (Java 8 ou plus récent).
2. Ajouter les classes Server et Client :
    - Créez une nouvelle classe Server (clic droit sur src -> New -> Java Class).
    - Collez le code du serveur.

```
import java.io.*;
import java.net.*;
public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Serveur en attente de connexion...");
            Socket socket = serverSocket.accept();
            System.out.println("Client connecté");
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            String message = input.readLine();
            System.out.println("Message reçu du client : " + message);
            output.println("Message bien reçu !");
            socket.close();
```

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

Explication du code :

- ServerSocket : Ce serveur écoute sur le port 5000 et attend les connexions des clients.
- accept() : Cette méthode bloque l'exécution jusqu'à ce qu'un client se connecte. Une fois qu'une connexion est acceptée, un objet Socket est créé pour la communication avec le client.
- BufferedReader et PrintWriter : Ils permettent respectivement de lire et d'écrire des messages via le flux d'entrée et de sortie du Socket.
- socket.close() : Une fois la communication terminée, le serveur ferme la connexion avec le client.

- Répétez pour la classe Client.

```
import java.io.*;
import java.net.*;
public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5000)) {
            // Création des flux de sortie et d'entrée
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            // Envoyer un message au serveur
            output.println("Bonjour Serveur !");
                    // Lire la réponse du serveur
            String response = input.readLine();
            System.out.println("Réponse du serveur : " + response);
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Explication du code client:

- Le client se connecte au serveur sur localhost (même machine) et le port 5000 avec new Socket("localhost", 5000).
- Le client envoie un message "Bonjour Serveur !" au serveur en utilisant un PrintWriter.
- Il lit la réponse du serveur à l'aide d'un BufferedReader.

3. Exécuter les programmes :
    - Exécutez d'abord la classe Server (clic droit sur le fichier -> Run).
    - Ensuite, exécutez la classe Client (clic droit sur le fichier -> Run).
    - Vous devriez voir la communication se produire entre le client et le serveur.

Exercice 2 : Introduction à Java RMI

Prérequis :

- IntelliJ IDEA installé.
- JDK configuré.
- Le projet doit être configuré pour utiliser Java RMI .

Étapes pour configurer dans IntelliJ IDEA :

1. Créer un nouveau projet Java avec RMI :
2. Configurer le registre RMI (RMI Registry) :

Avant d'exécuter l'application, vous devez vous assurer que le registre RMI est démarré. Pour cela, ouvrez une fenêtre de terminal et tapez rmiregistry (assurez-vous que le JDK est bien installé et configuré pour RMI).

Vous devriez voir un message indiquant que le registre est en cours d'exécution. Le registre RMI permet au client de se connecter au serveur RMI via un nom spécifique (ici, CalculatriceService).

Configuration du registre RMI (RMI Registry) sous Windows

SUITE TP ARCHITECTURE DISTRIBUEE

RMI (Remote Method Invocation) est une technologie Java permettant l'exécution de méthodes sur des objets distants. Pour que cela fonctionne, il est nécessaire de configurer et démarrer un registre RMI.

1. Vérifier l'installation de Java

Avant de commencer, assurez-vous que Java est bien installé sur votre machine en exécutant la commande :

java -version

Si Java est installé, vous verrez une sortie avec la version de Java. Si ce n'est pas le cas, téléchargez et installez le JDK depuis le site officiel d'Oracle ou adoptium.net.

2. Démarrer le registre RMI

Le registre RMI fonctionne sur le port **1099** par défaut. Pour le démarrer, ouvrez un terminal (CMD) et exécutez :

start rmiregistry

Cela lance rmiregistry en arrière-plan. Si vous obtenez une erreur indiquant que la commande rmiregistry n'est pas reconnue, cela signifie que **le chemin du JDK n'est pas configuré dans les variables d'environnement**.

3. Ajouter le chemin de Java dans les variables d'environnement

Si la compilation javac *.java ne fonctionne pas, il est possible que le compilateur Java ne soit pas trouvé. Voici comment ajouter le chemin du JDK à la variable d'environnement PATH sous Windows :

a) Trouver le chemin du JDK

- Allez dans le répertoire où Java est installé (souvent dans C:\Program Files\Java).
- Trouvez le dossier correspondant à votre version du JDK (par exemple, jdk-17).
- Copiez le chemin du dossier bin, qui ressemble à ceci :
- C:\Program Files\Java\jdk-17\bin

b) Ajouter le chemin dans les variables d'environnement

- Appuyez sur **Win + R**, tapez sysdm.cpl, et appuyez sur **Entrée**.
- Allez dans l'onglet **Avancé** et cliquez sur **Variables d'environnement**.
- Dans la section **Variables système**, recherchez la variable **Path**, sélectionnez-la, puis cliquez sur **Modifier**.

- Cliquez sur **Nouveau** et ajoutez le chemin copié (C:\Program Files\Java\jdk-17\bin).
- Cliquez sur **OK** pour valider les modifications.

c) Vérifier la configuration

Fermez et rouvrez le terminal, puis exécutez :

javac -version

java -version

Si les versions s'affichent correctement, la configuration est réussie.

4. Vérifier si le port 1099 est utilisé

Après avoir lancé rmiregistry, vous pouvez vérifier si le port **1099** est bien ouvert avec la commande suivante :

netstat -an | findstr "1099"

Si un processus écoute sur ce port, la commande affichera une ligne comme :

TCP 0.0.0.0:1099 0.0.0.0:0 LISTENING

Si rien n'apparaît, vérifiez si rmiregistry a bien démarré ou si un autre processus utilise déjà ce port.

➢ **Créer un nouveau projet Java dans IntelliJ IDEA**

a. Ouvrir IntelliJ IDEA :

- Lancez **IntelliJ IDEA**.
- Sélectionnez **New Project**.

b. Configurer le projet Java :

- Sélectionnez **Java** dans la liste des types de projets.
- Assurez-vous que le **JDK** est bien configuré (Java 8 ou supérieur).
- Cliquez sur **Next**.

c. Nommer le projet :

- Donnez un nom à votre projet, par exemple, **RMIExample**.
- Choisissez un emplacement où vous souhaitez enregistrer le projet et cliquez sur **Finish**.

SUITE TP ARCHITECTURE DISTRIBUEE

- **Créer l'interface distante (Calculatrice.java)**
  a. Dans le répertoire src, faites un clic droit et sélectionnez **New -> Java Class** pour créer une classe **Calculatrice**.

  b. Ajoutez le code suivant pour définir une **interface distante** qui sera utilisée par le client et le serveur :

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Calculatrice extends Remote {

    int addition(int a, int b) throws RemoteException;

}
```

- L'interface Calculatrice étend Remote pour spécifier qu'elle sera utilisée à distance via RMI.
- La méthode addition est déclarée avec throws RemoteException pour indiquer qu'elle peut lever des exceptions liées à RMI.

- **Créer l'implémentation du serveur (CalculatriceImpl.java)**

  a. Créez une nouvelle classe **CalculatriceImpl** pour implémenter l'interface **Calculatrice**.

  b. Le code de la classe CalculatriceImpl ressemble à ceci :

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculatriceImpl extends UnicastRemoteObject implements Calculatrice {
    protected CalculatriceImpl() throws RemoteException {
        super();
    }
    @Override
    public int addition(int a, int b) throws RemoteException {
        return a + b;
    }
}
```

- **UnicastRemoteObject** : Cette classe permet à l'objet de communiquer via RMI. L'appel à super() initialise l'objet pour qu'il puisse être exporté.
- La méthode addition est implémentée pour additionner deux entiers et renvoyer le résultat.

➢ **Créer le serveur RMI (ServerRMI.java)**

a. Créez une nouvelle classe **ServerRMI** dans laquelle vous lancerez le serveur RMI.

b. Le code de la classe ServerRMI ressemble à ceci :

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
public class ServerRMI {
    public static void main(String[] args) {
        try {
            // Créer le registre RMI sur le port 1099
            LocateRegistry.createRegistry(1099);
            System.out.println("Registre RMI créé.");
            // Créer une instance de l'implémentation de Calculatrice
            CalculatriceImpl calculatrice = new CalculatriceImpl();
            // Lier l'implémentation au registre sous le nom "CalculatriceService"
            Naming.rebind("rmi://localhost/CalculatriceService", calculatrice);
            System.out.println("Serveur RMI prêt...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- **LocateRegistry.createRegistry(1099)** : Crée le registre RMI sur le port **1099**, ce qui permet aux clients de se connecter à ce serveur.
- **Naming.rebind()** : Enregistre l'objet calculatrice dans le registre sous le nom "CalculatriceService", ce qui le rend accessible à partir de ce nom.

SUITE TP ARCHITECTURE DISTRIBUEE

➢ **Créer le client RMI (ClientRMI.java)**

a. Créez une nouvelle classe **ClientRMI** pour faire la demande à notre serveur.

b. Le code du client ClientRMI ressemble à ceci :

```
import java.rmi.Naming;

public class ClientRMI {
    public static void main(String[] args) {
        try {
            // Obtenir le stub du serveur via le registre RMI
            Calculatrice calculatrice = (Calculatrice) Naming.lookup("rmi://localhost/CalculatriceService");

            // Appeler la méthode addition sur le serveur
            int resultat = calculatrice.addition(5, 3);
            System.out.println("Résultat de l'addition : " + resultat);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- **Naming.lookup()** : Permet au client de localiser l'objet distant via le registre RMI.
- Le client appelle la méthode **addition** et affiche le résultat.

➢ **Exécuter le projet RMI**

a. **Démarrer le registre RMI** :

- Avant d'exécuter le serveur, vous devez démarrer le **registre RMI** dans une fenêtre de terminal.
- Ouvrez une fenêtre de terminal et tapez rmiregistry (assurez-vous que le JDK est installé et accessible dans votre PATH).
- Laissez cette fenêtre ouverte, car elle est nécessaire pour la communication RMI entre le client et le serveur.

b. **Exécuter le serveur RMI** :

- Dans IntelliJ IDEA, faites un clic droit sur **ServerRMI.java** et sélectionnez **Run 'ServerRMI.main()'**.
- Vous devriez voir le message : "Serveur RMI prêt...".

c. **Exécuter le client RMI** :

- Faites un clic droit sur **ClientRMI.java** et sélectionnez **Run 'ClientRMI.main()'**.
- Le client se connectera au serveur, appellera la méthode **addition(5, 3)**, et le résultat sera affiché dans la console du client : "Résultat de l'addition : 8".

Exercice 3 : Amélioration avec Multithreading et Sécurité

Prérequis :

- Utilisation de Tomcat pour l'aspect serveur web ou services web (pour le cas où vous voudriez faire un service REST ou SOAP avec Tomcat).

1. Multithreading dans le serveur Socket (Server avec plusieurs clients) :

Pour gérer plusieurs clients simultanément, vous pouvez modifier le code du serveur pour utiliser des threads. Voici comment faire dans IntelliJ IDEA :

1. Ajouter la gestion du multithreading dans le serveur :

Modifiez le serveur pour qu'il crée un nouveau thread pour chaque client qui se connecte.

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


public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Serveur en attente de connexion...");
            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("Client connecté");
                // Nouveau thread pour chaque client
                new Thread(new ClientHandler(socket)).start();
```

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

class ClientHandler implements Runnable {
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

            String message = input.readLine();
            System.out.println("Message reçu : " + message);
            output.println("Message bien reçu !");
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

}

2. Exécuter dans IntelliJ IDEA :

Exécutez le serveur et plusieurs clients (ils seront gérés en parallèle).

2. Sécuriser la communication avec SSL/TLS (RMI ou Socket) :

a. Configurer SSL pour Socket :

- Si vous utilisez des sockets, vous pouvez activer SSL pour sécuriser les échanges en utilisant SSLSocket et SSLServerSocket.
- Cela nécessite un keystore et un certificat SSL valide pour le chiffrement des échanges.

b. Configurer SSL pour RMI :

- Pour sécuriser RMI, vous devez configurer un serveur RMI utilisant SSL. Cela nécessite un keystore Java et l'utilisation de la bibliothèque JSSE (Java Secure Socket Extension).

Exercice 4 : Utilisation de Tomcat pour une application web

Si vous souhaitez déployer une application web dans un environnement distribué en utilisant Tomcat, voici comment procéder :

Étapes pour configurer un projet Java Web dans IntelliJ IDEA avec Tomcat :

1. Configurer un serveur Tomcat dans IntelliJ IDEA :
   - Allez dans File -> Settings -> Build, Execution, Deployment -> Application Servers.
   - Ajoutez votre installation Tomcat en choisissant le répertoire d'installation de Tomcat sur votre machine.
2. Créer un projet Web :
   - Créez un projet de type Java EE (Dynamic Web Project).
   - Dans le répertoire WEB-INF, créez les servlets nécessaires pour la gestion des requêtes HTTP.
   - Développez des servlets pour les interactions avec les clients.
3. Déployer sur Tomcat :
   - Une fois votre projet créé, configurez un artefact pour déployer l'application sur Tomcat.
   - Allez dans Run -> Edit Configurations et sélectionnez votre serveur Tomcat.
   - Ajoutez le projet comme artefact à déployer.

- Cliquez sur Run pour démarrer le serveur Tomcat avec votre application.

Exemple : Servlet Simple

```
@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().write("Bonjour, ceci est un servlet!");
    }
}
```