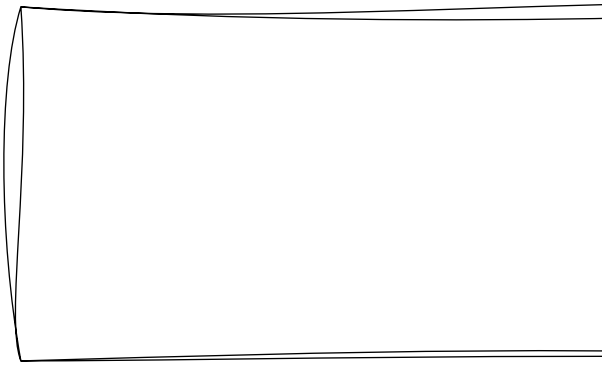
École d'Ingénierie Digitale
et d'Intelligence Artificielle



APPLICATION REPARTIE

RAPPORT DU PROJET DE FIN DE
FIN DE SEMESTRE

DZIME AUBIANG JOHAN KALEB

GESTION D'UN COMMERCE AVEC RPC

INTRODUCTION :

La gestion d'un commerce à l'aide de RPC (Remote Procedure Call) implique l'utilisation de plusieurs fichiers pour mettre en place un système distribué efficace. Les cinq fichiers clés impliqués dans ce processus
sont `StoreClient.java`, `StoreServer.java`, `StoreService.java`, `StoreServiceImpl.java` et `Product.java`. Chacun de ces fichiers joue un rôle spécifique dans l'architecture globale du système de gestion du commerce.

Dans ce contexte, `StoreClient.java` représente l'application cliente qui interagit avec le serveur, `StoreServer.java` est responsable de la gestion des requêtes et des communications avec les clients, `StoreService.java` définit les interfaces pour les services du magasin, `StoreServiceImpl.java` implémente ces services et `Product.java` modélise les produits vendus par le commerce.

Cette approche basée sur RPC permet une communication transparente entre les différentes parties du système, offrant ainsi une solution robuste pour la gestion des opérations de commerce. Chaque fichier joue un rôle crucial dans le bon fonctionnement et la cohérence de l'ensemble du système, assurant une expérience utilisateur fluide et une gestion efficace des activités commerciales.

Dans la suite notre exposé nous vous détaillerons ces diiférents fichiers qui font notre commerce.

I – STORESERVICE

StoreService.java

```java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface StoreService extends Remote {
    List<Product> getProducts() throws RemoteException;
    String buyProduct(String productName) throws RemoteException;
}
```

- L'interface `StoreService` étend `Remote`, ce qui indique qu'elle peut être utilisée pour l'invocation de méthodes à distance.
- Elle déclare deux méthodes :
    - `getProducts()` : Une méthode qui retourne une liste d'objets `Product`. Elle lance une `RemoteException` en cas d'erreur de communication.
    - `buyProduct(String productName)` : Une méthode qui prend le nom d'un produit en paramètre et retourne une `String`. Elle lance une `RemoteException` en cas d'erreur de communication.

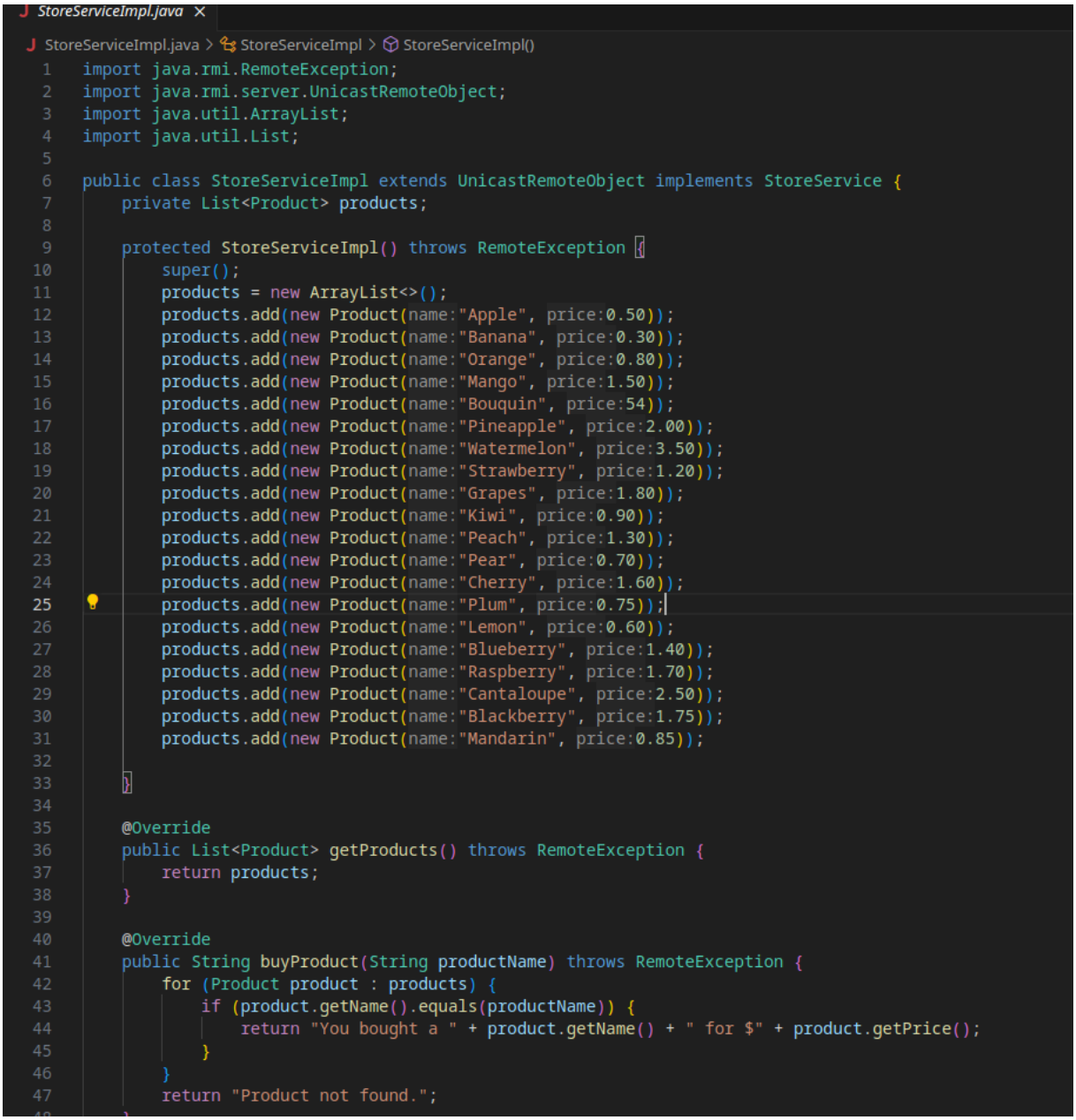
Cette interface définit le contrat pour les services qui peuvent être accessibles à distance. Les clients peuvent appeler ces méthodes sur un serveur distant implémentant cette interface pour récupérer une liste de produits ou acheter un produit spécifique.

II-STORESERVICEIMPL

La classe `StoreServiceImpl` implémente l'interface `StoreService` et étend `UnicastRemoteObject` pour supporter les appels distants.

- Elle contient une liste de produits initialisée avec plusieurs produits et leurs prix dans le constructeur.
- La méthode `getProducts()` permet de récupérer la liste de tous les produits disponibles.
- La méthode `buyProduct(String productName)` permet d'acheter un produit en vérifiant le nom du produit demandé et en renvoyant un message indiquant l'achat et le prix si le produit est trouvé, sinon un message indiquant que le produit n'a pas été trouvé.

Cette classe constitue l'implémentation du service de magasinage à distance et fournit la logique pour obtenir la liste des produits et acheter un produit spécifique.



III – STORESERVER

La classe `StoreServer` contient une méthode `main` qui est le point d'entrée du programme.

- Dans le bloc `try`, elle crée un registre RMI sur le port 1099 en utilisant `LocateRegistry.createRegistry(1099)`.
- Ensuite, elle crée une instance de `StoreServiceImpl` pour le service de magasinage.
- En utilisant `Naming.rebind("StoreService", storeService)`, elle lie ce service au registre RMI avec le nom "StoreService".
- En cas d'exception, l'erreur est affichée à l'aide de `e.printStackTrace()`.

Cette classe est essentielle pour démarrer le serveur RMI qui expose le service de magasinage à distance pour que les clients puissent y accéder.

```java
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class StoreServer {
    Run | Debug
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099);
            StoreService storeService = new StoreServiceImpl();
            Naming.rebind("StoreService", storeService);
            System.out.println("Store Service is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

IV-PRODUCTS

```java
import java.io.Serializable;

public class Product implements Serializable {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

La classe `Product` représente un produit avec deux attributs : le nom du produit (`name`) et son prix (`price`).

- Le constructeur de la classe permet de créer une instance de produit en spécifiant le nom et le prix.
- Les méthodes `getName()` et `getPrice()` permettent respectivement de récupérer le nom et le prix du produit.

En implémentant l'interface `Serializable`, les instances de la classe `Product` peuvent être sérialisées, c'est-à-dire converties en un format pouvant être facilement stocké ou transmis sous forme de flux d'octets, ce qui est utile lors de la communication à distance via RMI

V -STORECLIENT

La classe `StoreClient` étend `JFrame` pour créer une fenêtre GUI.

- La méthode `updateProductList()` est utilisée pour mettre à jour la liste des produits affichés dans l'interface en récupérant la liste des produits à partir du service distant.
- La méthode `buyProduct()` est appelée lorsqu'un utilisateur souhaite acheter un produit sélectionné dans la liste. Elle communique avec le service distant pour effectuer l'achat et affiche le message de confirmation dans la zone de sortie.
- Un bouton "Achat Product" est ajouté à l'interface pour permettre à l'utilisateur d'acheter un produit.
- Un bouton "Trier par Prix" est ajouté à l'interface pour permettre à l'utilisateur de trier les produits par prix.

La méthode `sortProductsByPrice()` :

    - Cette méthode trie la liste des produits par prix en utilisant `Collections.sort` avec un `Comparator` qui compare les prix des produits.
    - Une fois les produits triés, la méthode appelle `updateProductList()` pour mettre à jour l'affichage de la liste des produits dans l'interface.

2. Le bloc `main` :

    - La méthode `main` est l'entrée principale de l'application.
    - Elle utilise `SwingUtilities.invokeLater()` pour exécuter la création de l'interface graphique dans le thread de l'interface utilisateur (UI).
    - En créant une nouvelle instance de `StoreClient` et en la rendant visible, l'interface utilisateur du client de magasinage est lancée pour l'utilisateur.

Ces ajouts permettent à l'utilisateur de trier les produits par prix en cliquant sur le bouton "Trier par Prix" dans l'interface graphique. Une fois triés, la liste des produits est mise à jour pour refléter le nouvel ordre basé sur les prix.

StoreClient.java > ...

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.List;
import java.util.Collections;
import java.util.Comparator;

public class StoreClient extends JFrame {
    private StoreService storeService;
    private DefaultListModel<String> listModel;
    private JList<String> productList;
    private JTextArea outputArea;
    private List<Product> products;

    public StoreClient() {
        try {
            storeService = (StoreService) Naming.lookup("rmi://localhost/StoreService");
        } catch (Exception e) {
            e.printStackTrace();
        }

        setTitle("commerce de kaleb");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        listModel = new DefaultListModel<>();
        productList = new JList<>(listModel);
        JScrollPane scrollPane = new JScrollPane(productList);

        JButton buyButton = new JButton("Achat Product");
        buyButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                buyProduct();
            }
        });

        JButton sortButton = new JButton("Trier par Prix");
        sortButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            sortProductsByPrice();
                }
            });
```

0 0 Java: Ready

```
                remoteException.printStackTrace();
            }
        } else {
            outputArea.setText("Please select a product.");
        }
    }
    private void sortProductsByPrice() {
    Collections.sort(products, Comparator.comparing(Product::getPrice));
    updateProductList();
}


    Run | Debug
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new StoreClient().setVisible(true);
            }
        });
    }
}
```

```
        outputArea = new JTextArea(5, 20);
        outputArea.setEditable(false);
        JScrollPane outputScrollPane = new JScrollPane(outputArea);

        JPanel panel = new JPanel();
        panel.add(buyButton);
        panel.add(sortButton);

        add(scrollPane, BorderLayout.CENTER);
        add(panel, BorderLayout.SOUTH);
        add(outputScrollPane, BorderLayout.NORTH);

        updateProductList();
    }

    private void updateProductList() {
        try {
            products = storeService.getProducts();
            listModel.clear();
            List<Product> products = storeService.getProducts();
            for (Product product : products) {
                listModel.addElement(product.getName());
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    private void buyProduct() {
        String selectedProduct = productList.getSelectedValue();
        if (selectedProduct != null) {
            String productName = selectedProduct.split(" - $")[0];
            try {
                String message = storeService.buyProduct(productName);
                outputArea.setText(message);
                updateProductList();
            } catch (RemoteException remoteException) {
                remoteException.printStackTrace();
            }
        } else {
            outputArea.setText("Please select a product.");
        }
    }
    private void sortProductsByPrice() {
    Collections.sort(products, Comparator.comparing(Product::getPrice));
    updateProductList();
```

CONCLUSION :

En conclusion, la mise en œuvre d'un système de gestion de commerce utilisant RPC (Remote Procedure Call) à travers les fichiers `StoreClient.java`, `StoreServer.java`, `StoreService.java`, `StoreServiceImpl.java` et `Product.java` permet de créer une architecture distribuée efficace pour gérer les opérations commerciales.

Grâce à cette approche, l'interaction entre l'application cliente, le serveur, les services du magasin et les objets produits est facilitée, garantissant une communication fluide et fiable. Chaque composant du système joue un rôle essentiel dans la gestion des activités commerciales, de la manipulation des requêtes client à la fourniture des services du magasin et à la modélisation des produits vendus.

En combinant ces différents fichiers de manière cohérente, il est possible de mettre en place un environnement robuste pour la gestion des opérations commerciales, offrant une expérience utilisateur optimale et une gestion efficace des stocks, des ventes et des autres aspects d'un commerce. L'utilisation de RPC dans ce contexte permet d'améliorer la scalabilité, la performance et la fiabilité du système, contribuant ainsi à une gestion efficace et automatisée des activités commerciales.