1.2 Activité pratique : Service de conversion de devises

Objectif du TP

L'objectif de ce TP est de permettre aux étudiants de se familiariser avec la création d'un service gRPC en utilisant Protocol Buffers (Protobuf) pour la sérialisation des données. Le service implémentera un système de conversion de devises, offrant ainsi un exemple concret des communications gRPC. Le TP explore plusieurs modes de communication gRPC, notamment l'appel unitaire, le streaming côté serveur, le streaming côté client, et le streaming bidirectionnel. Enfin, la génération automatique des stubs Protobuf sera couverte pour assurer la bonne interaction entre le client et le serveur.

Ce TP est conçu pour simuler une situation réelle où les services gRPC sont largement utilisés pour développer des systèmes distribués, en particulier dans les environnements où la latence faible et la scalabilité sont des priorités (par exemple, dans les microservices et les systèmes cloud-native).

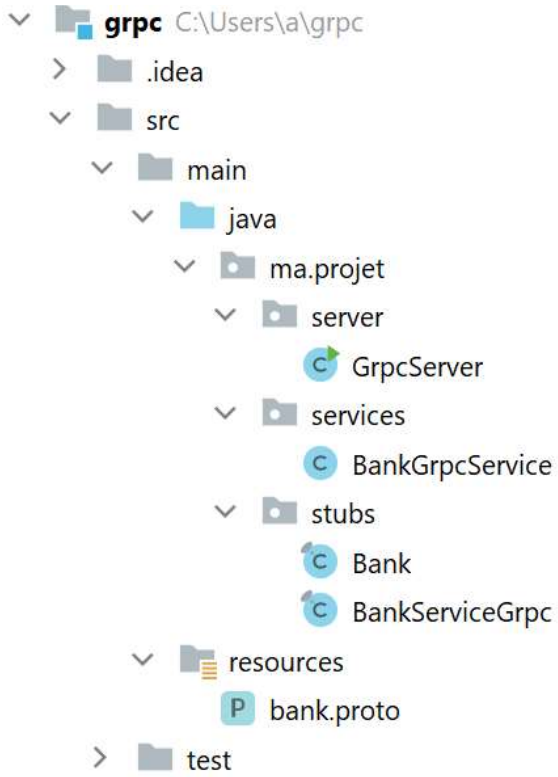

Figure 1.1: Structure de projet.

Prérequis

- Avoir Java installé sur la machine, car gRPC s'intègre parfaitement à Java via des stubs générés automatiquement.
- Utiliser Maven comme outil de build pour la gestion des dépendances et l'automatisation de tâches comme la génération de stubs Protobuf.
- Avoir des notions de base sur gRPC et Protocol Buffers, car ces technologies sont à la base de la communication interservices dans ce TP.

1. Dépendances Maven

Pour commencer, il est essentiel d'ajouter les bonnes dépendances dans le fichier `pom.xml`. Ces dépendances incluent les bibliothèques nécessaires pour utiliser gRPC et Protocol Buffers dans un projet Java. Voici un résumé des dépendances importantes :

- `protobuf-java` : Cette dépendance inclut les classes Java générées à partir de vos fichiers Protobuf (`.proto`) qui définissent les messages.

- `grpc-netty-shaded` : Fournit l'implémentation réseau pour gRPC, permettant de gérer les connexions clients-serveur.
- `grpc-protobuf` : Ce paquet génère automatiquement des stubs basés sur les messages Protobuf pour établir les communications entre client et serveur.
- `grpc-stub` : Contient les stubs nécessaires pour appeler les services gRPC.
- `javax.annotation-api` : Requis pour utiliser des annotations qui facilitent la génération de code.

```
<dependencies>
  <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.22.0</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.53.0</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>1.53.0</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>1.53.0</version>
  </dependency>
  <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
  </dependency>
</dependencies>
```

2. Configuration du Build Maven

Pour que Maven génère automatiquement les classes Java à partir des fichiers Protobuf (`.proto`), il est nécessaire de configurer un plugin Maven. Ce plugin, `protoc-jar-maven-plugin`, exécute le compilateur Protobuf lors de la phase de génération des sources.

Le rôle du compilateur Protobuf est de convertir les définitions de message et de service contenues dans les fichiers `.proto` en stubs Java. Ces stubs seront ensuite utilisés dans le code pour les appels gRPC.

```
<build>
  <plugins>
    <plugin>
      <groupId>com.github.os72</groupId>
      <artifactId>protoc-jar-maven-plugin</artifactId>
      <version>3.11.4</version>
      <executions>
        <execution>
```

```
            <phase>generate-sources</phase>
            <goals>
               <goal>run</goal>
            </goals>
            <configuration>
               <inputDirectories>
                  <include>src/main/resources</include>
               </inputDirectories>
               <outputTargets>
               <outputTarget>
                  <type>java</type>
               <outputDirectory>src/main/java</outputDirectory>
               </outputTarget>
               <outputTarget>
                  <type>grpc-java</type>
               <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.15.0</pluginArtifact>
                  <outputDirectory>src/main/java</outputDirectory>
               </outputTarget>
               </outputTargets>
            </configuration>
         </execution>
      </executions>
   </plugin>
</plugins>
</build>
```

3. Fichier Protobuf

Le fichier `.proto` est essentiel dans gRPC. Il définit les messages (les données à transmettre) ainsi que les services (les méthodes que les clients peuvent appeler). Ici, nous définissons un service de conversion de devises qui inclut plusieurs méthodes de communication.

Les méthodes de conversion utilisent différents modes de communication :

- `convert` : Une simple requête-réponse.
- `getCurrencyStream` : Envoie un flux de réponses depuis le serveur au client.
- `performStream` : Envoie un flux de requêtes du client au serveur.
- `fullCurrencyStream` : Gère un flux bidirectionnel de requêtes et de réponses.

```
syntax = "proto3";

option java_package="ma.projet.stubs";

service BankService {
   rpc convert (ConvertCurrencyRequest) returns (ConvertCurrencyResponse);
   rpc getCurrencyStream (ConvertCurrencyRequest) returns (stream
       ConvertCurrencyResponse);
   rpc performStream (stream ConvertCurrencyRequest) returns
       (ConvertCurrencyResponse);
   rpc fullCurrencyStream (stream ConvertCurrencyRequest) returns (stream
       ConvertCurrencyResponse);
}

message ConvertCurrencyRequest {
   string currencyFrom = 1;
   string currencyTo = 2;
```

```
    double amount = 3;
}

message ConvertCurrencyResponse {
    string currencyFrom = 1;
    string currencyTo = 2;
    double amount = 3;
    double result = 4;
}
```

4. Implémentation du Service gRPC

L'implémentation Java d'un service gRPC repose sur l'extension de la classe générée automatiquement `BankServiceGrpc.BankServiceImplBase`. Dans cet exemple, nous implémentons la méthode `convert`, qui prend en charge une conversion de devises simple.

```
package ma.projet.services;

import io.grpc.stub.StreamObserver;
import ma.projet.stubs.Bank;
import ma.projet.stubs.BankServiceGrpc;

public class BankGrpcService extends BankServiceGrpc.BankServiceImplBase {

    @Override
    public void convert(Bank.ConvertCurrencyRequest request,
    StreamObserver<Bank.ConvertCurrencyResponse> responseObserver) {

        // Extraction des paramètres de la requête
        String currencyFrom = request.getCurrencyFrom();
        String currencyTo = request.getCurrencyTo();
        double amount = request.getAmount();

        // Logique simple de conversion de devises (ici avec un taux fixe fictif)
        Bank.ConvertCurrencyResponse response =
            Bank.ConvertCurrencyResponse.newBuilder()
        .setCurrencyFrom(currencyFrom)
        .setCurrencyTo(currencyTo)
        .setAmount(amount)
        .setResult(amount * 11.4) // Le résultat de la conversion
        .build();

        // Envoi de la réponse au client
        responseObserver.onNext(response);

        // Marquer la fin de l'appel gRPC
        responseObserver.onCompleted();
    }
}
```

R Dans cet exemple, la logique de conversion est simple, avec un taux fixe (`amount * 11.4`). Dans une implémentation réelle, cette logique pourrait être plus complexe, impliquant des taux de change dynamiques récupérés depuis une API tierce.

5. Serveur gRPC

Le serveur gRPC est initialisé en spécifiant le port sur lequel il écoutera les requêtes et en enregistrant le service `BankGrpcService`.

```
package ma.projet.server;

import io.grpc.Server;
import io.grpc.ServerBuilder;
import ma.projet.services.BankGrpcService;

import java.io.IOException;

public class GrpcServer {
   public static void main(String[] args) throws IOException,
        InterruptedException {
      // Création et configuration du serveur gRPC
      Server server = ServerBuilder.forPort(5555)
      .addService(new BankGrpcService())
      .build();

      // Démarrage du serveur
      server.start();

      // Attendre que le serveur soit arrêté
      server.awaitTermination();
   }
}
```

R Ce serveur est configuré pour écouter sur le port 5555, mais ce numéro de port peut être modifié en fonction des besoins du projet. Le serveur reste en attente de requêtes clients, et lorsqu'il en reçoit, il invoque le service correspondant (dans ce cas, la conversion de devises).

6. Lancement du Serveur

Une fois le serveur configuré, il peut être démarré via la commande suivante :

```
mvn exec:java -Dexec.mainClass="ma.projet.server.GrpcServer"
```

R Le serveur gRPC est une application long-running, ce qui signifie qu'il restera en fonctionnement jusqu'à ce qu'il soit explicitement arrêté. Cela permet aux clients de se connecter et d'envoyer des requêtes en continu.

R Ce TP permet d'acquérir une bonne compréhension des fondamentaux de gRPC, y compris la définition des services et messages via Protobuf, l'implémentation des services en Java, et la configuration et exécution d'un serveur gRPC.

7. Test du Service gRPC avec BloomRPC

Étapes pour tester avec BloomRPC

- **Installer BloomRPC** : Télécharger et installer BloomRPC depuis BloomRPC Releases.
- **Charger le fichier** `bank.proto` :
  - Ouvrir BloomRPC.
  - Cliquer sur le bouton **"+"** pour ajouter un fichier `.proto`.

  - Sélectionner le fichier `bank.proto`.
- **Configurer le serveur gRPC** :
  - Dans le champ d'adresse du serveur, entrer `localhost:5555`.
  - S'assurer que la méthode de communication sélectionnée est `Unary Call`.
- **Tester la méthode** `convert` :
  - Sélectionner la méthode `convert` dans la liste des services à gauche.
  - Dans l'éditeur JSON à droite, entrer les valeurs suivantes :

```
{
   "currencyFrom": "Devise1",
   "currencyTo": "Devise2",
   "amount": 123
}
```

  - Cliquer sur l'icône "Play" pour exécuter l'appel.
- **Vérifier la réponse du serveur** :
  - Observer la réponse dans l'onglet de réponse à droite.
  - La réponse doit contenir les champs `currencyFrom`, `currencyTo`, `amount`, et `result`.
- **Référence à la figure** : Voir la figure ci-dessous pour un exemple de l'appel avec la méthode `convert` et la réponse affichée dans BloomRPC.

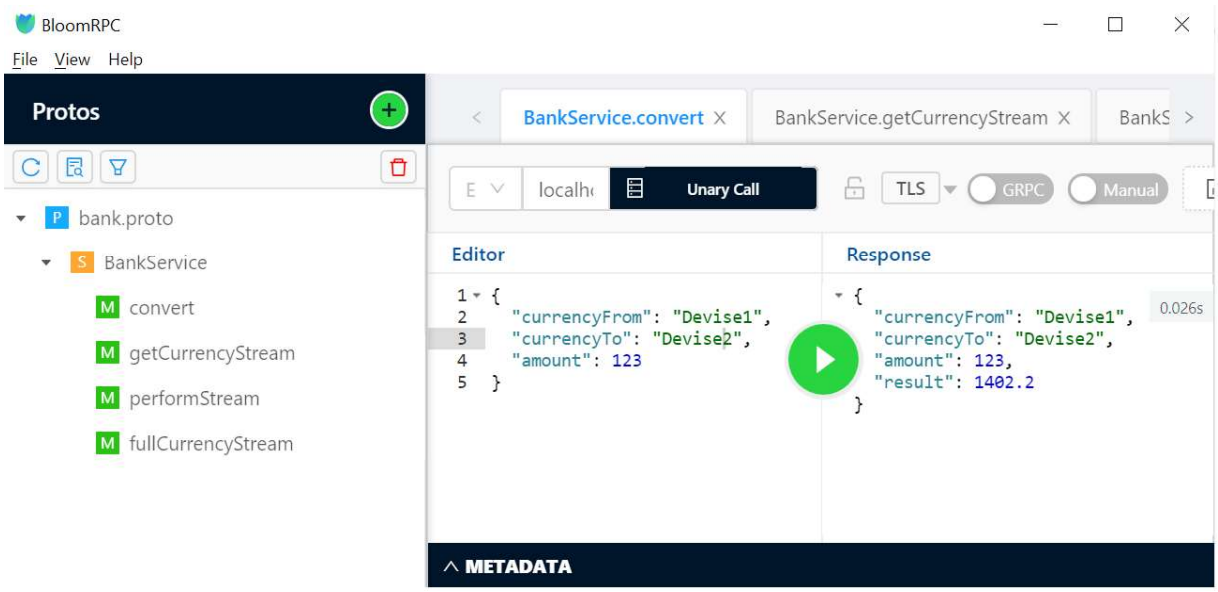

Figure 1.2: Test de la méthode `convert` avec BloomRPC

1.3 Activité pratique : Implémentation du Client gRPC

Le fichier `proto` et les dépendances `POM` utilisés pour l'implémentation du client sont les mêmes que ceux utilisés par le serveur. Voici le détail de l'implémentation des clients gRPC en mode synchrone et asynchrone.

Client Synchrone gRPC

Le client synchrone effectue une conversion de devise en appelant la méthode `convert` de manière bloquante.

```
public class Client1 {
```

```
    public static void main(String[] args) {
        // Creating a managed channel for gRPC communication
        ManagedChannel managedChannel = ManagedChannelBuilder.forAddress("localhost",
            5555)
        .usePlaintext() // This ensures that the connection is made in plaintext, not
            SSL
        .build();

        // Creating a stub to interact with the gRPC server using the blocking API
        BankServiceGrpc.BankServiceBlockingStub blockingStub =
            BankServiceGrpc.newBlockingStub(managedChannel);

        // Creating a request object for currency conversion
        Bank.ConvertCurrencyRequest request = Bank.ConvertCurrencyRequest.newBuilder()
        .setCurrencyFrom("MAD") // Setting the source currency as Moroccan Dirham
        .setCurrencyTo("USD") // Setting the target currency as US Dollars
        .setAmount(7600)      // Setting the amount to be converted
        .build();

        // Sending the request and receiving the response from the server
        Bank.ConvertCurrencyResponse currencyResponse = blockingStub.convert(request);

        // Printing the response to the console
        System.out.println(currencyResponse);
    }

}
```

R Le client utilise `ManagedChannelBuilder` pour créer un canal de communication avec le serveur gRPC en spécifiant l'adresse `localhost` et le port 5555. L'option `usePlaintext()` est utilisée pour désactiver le SSL, ce qui est commun dans les environnements de développement.

Le stub bloquant (`BankServiceGrpc.BankServiceBlockingStub`) est ensuite créé à partir de ce canal pour envoyer des requêtes synchrones au serveur. La méthode `convert` est appelée avec un objet `ConvertCurrencyRequest` spécifiant les devises source (`MAD`) et cible (`USD`), ainsi que le montant à convertir (7600 MAD).

Une fois la réponse reçue, le résultat est imprimé dans la console. La réponse contient des informations sur les devises et le montant converti.

Client Asynchrone gRPC

Le client asynchrone utilise un `StreamObserver` pour traiter les réponses reçues de manière non bloquante.

```
public class Client2 {

    public static void main(String[] args) throws IOException {
        // Creating a managed channel for gRPC communication
        ManagedChannel managedChannel = ManagedChannelBuilder.forAddress("localhost",
            5555)
        .usePlaintext() // Ensures that the connection is made in plaintext, without
            SSL
        .build();

        // Creating an asynchronous stub to interact with the gRPC server
```

```
        BankServiceGrpc.BankServiceStub asyncStub =
            BankServiceGrpc.newStub(managedChannel);

        // Creating a request object for currency conversion
        Bank.ConvertCurrencyRequest request = Bank.ConvertCurrencyRequest.newBuilder()
        .setCurrencyFrom("MAD") // Setting the source currency (Moroccan Dirham)
        .setCurrencyTo("USD") // Setting the target currency (US Dollars)
        .setAmount(7500)      // Setting the amount to be converted
        .build();

        // Sending the request asynchronously and receiving the response
        asyncStub.convert(request, new StreamObserver<Bank.ConvertCurrencyResponse>()
            {

           @Override
           public void onNext(Bank.ConvertCurrencyResponse convertCurrencyResponse) {
              // Handling the response when received
              System.out.println("**********************");
              System.out.println(convertCurrencyResponse);
              System.out.println("**********************");
           }

           @Override
           public void onError(Throwable throwable) {
              // Handling any error that occurs during communication
              System.out.println(throwable.getMessage());
           }

           @Override
           public void onCompleted() {
              // Handling the completion of the gRPC call
              System.out.println("END...");
           }
        });

        // Waiting for the server response before closing
        System.out.println(".....?");
        System.in.read();
    }

}
```

R Le client asynchrone fonctionne de manière similaire au client synchrone, mais utilise un `StreamObserver` pour gérer les réponses de manière non bloquante. Le stub asynchrone (`BankServiceGrpc.BankServiceStub`) est utilisé ici.

Une fois la requête envoyée, les méthodes du `StreamObserver` s'occupent de traiter la réponse reçue :

- `onNext()` : Affiche la réponse lorsque le serveur envoie une conversion de devise.
- `onError()` : Affiche un message d'erreur en cas de problème durant l'exécution.
- `onCompleted()` : Indique la fin du traitement.

Le programme attend la fin de la communication avant de se terminer grâce à `System.in.read()`.