

Rapport

Activité pratique : GRPC
webService d'une banque



MASTER Systèmes Distribués et Intelligence Artificielle

Réalisé par
El Aloui Oussama

Prof. Mohamed Youssefi

SOMMAIRE

I.	Objectifs :	6
II.	Partie 1 : Protocol Buffers	7
1.	: Implémentation :	7
2.	Exécution :	8
3.	Partie 2 : Serveur gRPC	9
1.	Implémentation :	9
4.	Partie 3 : Implémentation des services	10
1.	Unary Model :	10
1.1.	Coté serveur :	10
1.2.	Coté client :	11
1.3.	Exécution :	12
2.	Client Streaming Model	13
2.1	. Coté serveur :	13
2.2.	Coté client :	13

2.3. Exécution :	15
3. Client Streaming Model	16
3.1. Coté serveur :	16
3.2. Coté client :	16
3.3. Exécution :	18
4. Bi-directional Streaming Model :	19
4.1. Coté serveur :	19
4.2. Coté client :	19
4.3. Exécution :	21

Table de figures

Figure 1 : Protocole Buffer (définition des méthode)	7
Figure 2 : Protocole Buffer (définition des messages)	7
Figure 3 : Compilation de proto Buffer	8
Figure 4 : Stubs (Bank, BankServiceGrpc)	8
Figure 5 : Implémentation de serveur webservice gRPC.....	9
Figure 6 : Définition d'un service de type Unary Model.....	10
Figure 7: Classe Client en mode UNARY MODEL.....	11
Figure 8: Résultat d'exécution pour le cas UNARY MODEL.	12
Figure 9 : Définition d'un service de type Server Streaming Model.	13
Figure 10 : Définition de la classe Client en mode Server Streaming Model (1) 13	
Figure 11: Résultat d'exécution- pour le cas Serveur Streaming Model.....	15
Figure 12 : Définition d'un service de type Client Streaming Model.	16
Figure 13: Définition de la classe Client en mode Client Streaming Model (1)..	16
Figure 14 : Définition de la classe Client en mode Client Streaming Model (3). 17	
Figure 15: Définition de la classe Client en mode Client Streaming Model (2)..	17
Figure 16 : Résultat d'exécution pour le cas Client Streaming Model	18
Figure 17 : Définition de service en mode Bi-directional Streaming Model	19

Figure 18: Définition de la classe Client en mode Bi-directional Streaming Model (1).....	19
Figure 19 : Définition de la classe Client en mode Bi-directional Streaming Model (3)	20
Figure 20: Définition de la classe Client en mode Bi-directional Streaming Model (2).....	20
Figure 21: Résultat d'exécution pour le cas Bi-directional Streaming model	21

I. Objectifs :

Implémenter un webservice de type GRPC qui contient le service de convertir une monnaie d'une devise vers une autre devise en utilisant 4 modèles suivants:

- Unary Model
- Server Streaming Model
- Client Streaming Model
- BiDirectional Streaming Model

II. Partie 1 : Protocol Buffers

1. : Implémentation :

```
service BankService{
    //Type of communication
    rpc convert(ConvertCurrencyRequest) returns(ConvertCurrencyResponse); //type : unary model

    rpc convertStreamServer(ConvertCurrencyRequest) returns(stream ConvertCurrencyResponse); //type : Server Streaming model

    rpc convertStreamClient(stream ConvertCurrencyRequest) returns(ConvertCurrencyResponse); // type : Client Streaming model

    rpc convertStreamCltSrv(stream ConvertCurrencyRequest) returns(stream ConvertCurrencyResponse); // type : Bi-directional Streaming model

    rpc getListAccount(listAccountRequest) returns (listAccount);
}
```

Figure 1 : Protocole Buffer (définition des méthode)

```
// Définir le contenu des attributs
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;
}
```

Figure 2 : Protocole Buffer (définition des messages)

2. Exécution :

- On compile le fichier 'bank.proto' pour générer les stubs (des classes en java) en fonction les méthodes déclarées sur le fichier 'bank.proto'.

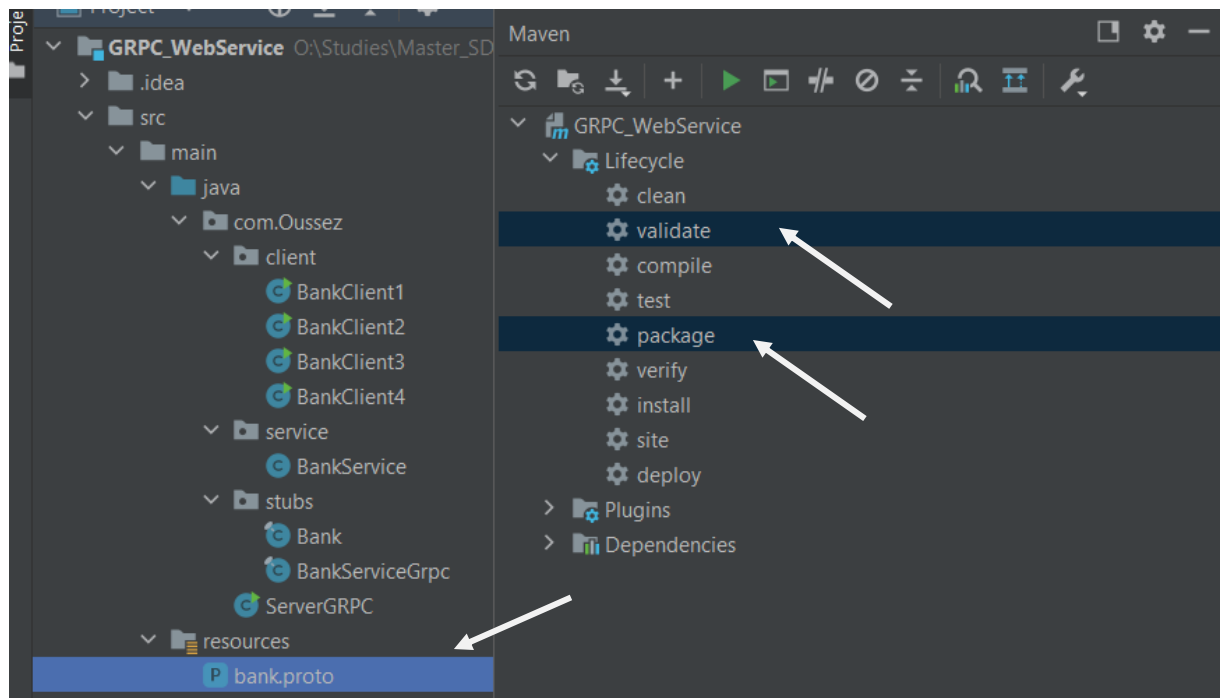


Figure 3 : Compilation de proto Buffer

- Voici les différents stubs générés :

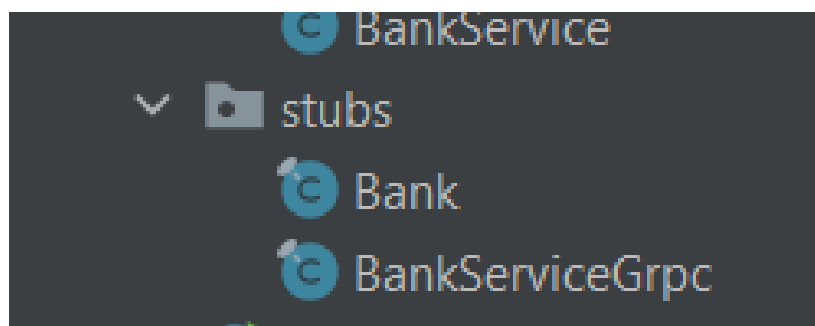


Figure 4 : Stubs (Bank, BankServiceGrpc)

3. Partie 2 : Serveur gRPC

1. Implémentation :

- Pour lancer un webservice, il faut implémenter et déployer son serveur.
- Dans cette partie, on doit définir une classe qui implémente le serveur webservice gRPC.

```
no usages
public class ServerGRPC {

    no usages
    public static void main(String[] args) throws IOException, InterruptedException {
        Server server = ServerBuilder.forPort(9999)
            .addService(new BankService()) //ClassXService is the class that defines the methods included in fileX.proto
            .build();

        server.start();
        System.out.println(">> SevrerGRPC is ON ...");
        server.awaitTermination(); //permet de laisser le thread figé après l'execution de cette instruction pour que
        //le serveur reste toujours actif !
    }
}
```

Figure 5 : Implémentation de serveur webservice gRPC.

- Pour démarrer le serveur, il suffit d'exécuter cette classe pour qu'il soit en état d'écoute des requêtes clientes.

4. Partie 3 : Implémentation des services

1. Unary Model :

1.1. Coté serveur :

- Dans cette partie, on définit les différentes méthodes déclarées sur le PROTO Buffer de webservice.

```
public void convert(Bank.ConvertCurrencyRequest request,
    StreamObserver<com.Oussez.stubs.Bank.ConvertCurrencyResponse> responseObserver) {

    //initialisation des attributs de message de réponse.
    double amount = request.getAmount();
    String currencyFrom = request.getCurrencyFrom();
    String currencyTo = request.getCurrencyTo();

    double result=amount*10.3;

    Bank.ConvertCurrencyResponse convertCurrencyResponse = Bank.ConvertCurrencyResponse.newBuilder()
        .setCurrencyFrom(currencyFrom)
        .setCurrencyTo(currencyTo)
        .setAmount(amount)
        .setResult(result)
        .build();//Creation de message de réponse

    responseObserver.onNext(convertCurrencyResponse); //envoyer la réponse au client
    responseObserver.onCompleted(); //notifier le client que le serveur à arreter la communication.
}
```

Figure 6 : Définition d'un service de type Unary Model

1.2. Coté client :

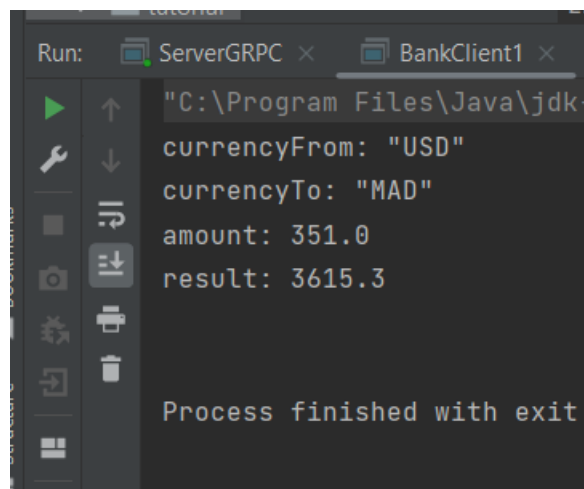
- Une nouvelle classe sera créée pour représenter un scénario d'échange des messages avec le serveur.

```
public class BankClient1 {  
    /**                                UNARY MODEL                                **/  
    no usages  
    public static void main(String[] args) {  
        //Déclarer un canal de communication  
        ManagedChannel managedChannel = ManagedChannelBuilder.forAddress( name: "localhost", port: 9999) ManagedChan  
            .usePlaintext() capture of ?  
            .build();  
  
        BankServiceGrpc.BankServiceBlockingStub blockingStub = BankServiceGrpc.newBlockingStub(managedChannel);  
        //Initialisation de message requete  
        Bank.ConvertCurrencyRequest currencyRequest = Bank.ConvertCurrencyRequest.newBuilder()  
            .setCurrencyTo("MAD")  
            .setCurrencyFrom("USD")  
            .setAmount(351)  
            .build();  
  
        Bank.ConvertCurrencyResponse currencyResponse = blockingStub.convert(currencyRequest); //faire appel à l  
        System.out.println(currencyResponse);  
    }  
}
```

Figure 7: Classe Client en mode UNARY MODEL

1.3. Exécution :

- On lance en premier lieu la classe qui présente le serveur, puis on lance la classe de client pour débiter leur communication.



```
Run: ServerGRPC x BankClient1 x
"C:\Program Files\Java\jdk-
currencyFrom: "USD"
currencyTo: "MAD"
amount: 351.0
result: 3615.3

Process finished with exit
```

Figure 8: Résultat d'exécution pour le cas UNARY MODEL.

2. Client Streaming Model

2.1. Coté serveur :

```
1 usage
2
public StreamObserver<Bank.ConvertCurrencyRequest> convertStreamCltSrv(StreamObserver<Bank.ConvertCurrencyResponse> responseObserver) {
    return new StreamObserver<Bank.ConvertCurrencyRequest>() {
        2 usages
        double somme;
        2 usages
        int total_request;
        @Override
        public void onNext(Bank.ConvertCurrencyRequest convertCurrencyRequest) {
            total_request++;
            somme += convertCurrencyRequest.getAmount();
            Bank.ConvertCurrencyResponse response = Bank.ConvertCurrencyResponse.newBuilder()
                .setResult(somme)
                .build();

            System.out.println(">> Client request ["+total_request+"]");
            responseObserver.onNext(response);
        }

        @Override
        public void onError(Throwable throwable) {

        }
    }
}
```

Figure 9 : Définition d'un service de type Server Streaming Model.

2.2. Coté client :

```
public static void main(String[] args) throws IOException {
    //Déclarer un canal de communication
    ManagedChannel managedChannel = ManagedChannelBuilder.forAddress( name: "localhost", port: 9999) ManagedChannelBuilder<capture of ?>
        .usePlaintext() capture of ?
        .build();

    BankServiceGrpc.BankServiceStub asyncStub = BankServiceGrpc.newStub(managedChannel); //communication asynchrone(non bloquante )
    Bank.ConvertCurrencyRequest currencyRequest = Bank.ConvertCurrencyRequest.newBuilder()
        .setCurrencyTo("MAD")
        .setCurrencyFrom("USD")
        .setAmount(351)
        .build();
}
```

Figure 10 : Définition de la classe Client en mode Server Streaming Model (1)

```

//Faire appeler le service de serveur en mode streaming
asyncStub.convertStreamServer(currencyRequest, new StreamObserver<Bank.ConvertCurrencyResponse>() {

    @Override
    public void onNext(Bank.ConvertCurrencyResponse convertCurrencyResponse) { //response sent from server
        System.out.println(">> [SERVER RESPONSE] :\n "+convertCurrencyResponse);
    }

    @Override
    public void onError(Throwable throwable) { System.out.println(">> [ERROR]: "+throwable.getMessage()); }

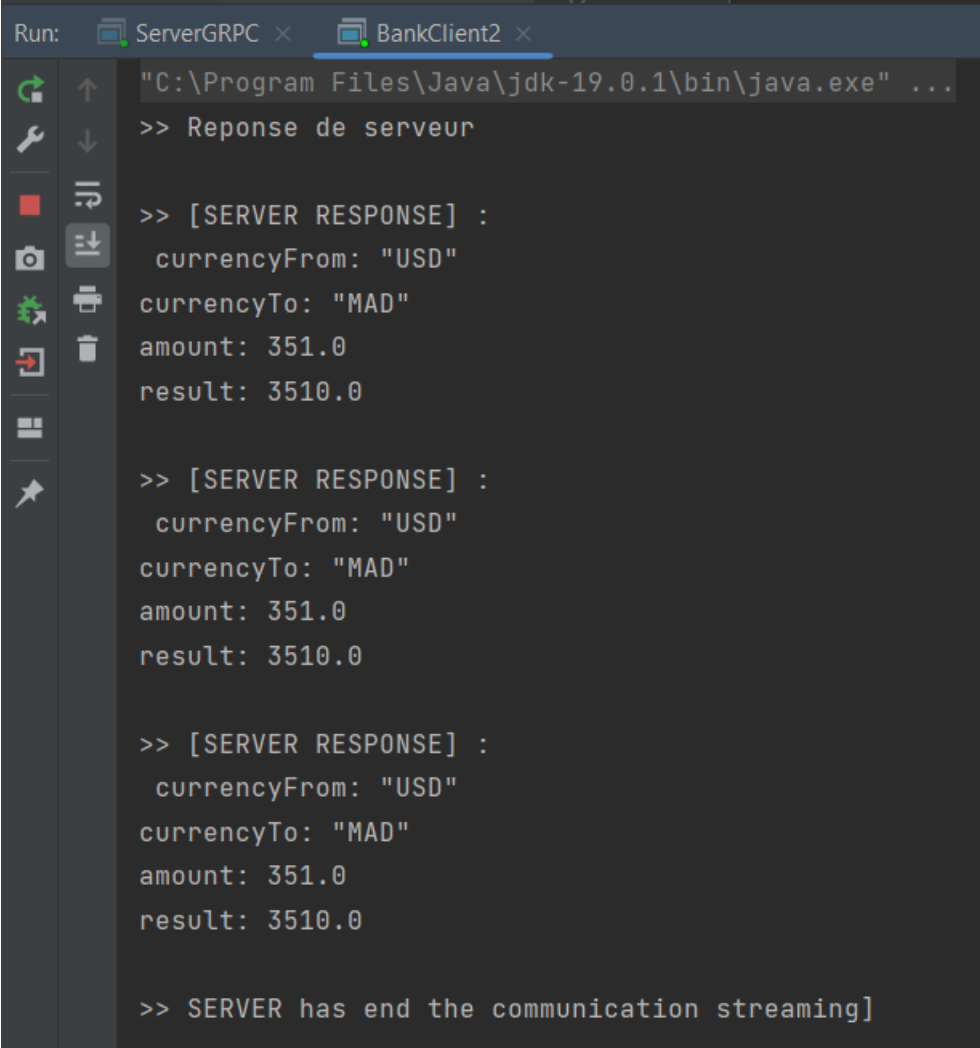
    7 usages
    @Override
    public void onCompleted() { System.out.println(">> SERVER has end the communication streaming"); }
});

System.out.println(">> Reponse de serveur \n");
System.in.read();
}

```

Figure 1 : Définition de la classe Client en mode Server Streaming Model (2)

2.3. Exécution :



```
Run: ServerGRPC x BankClient2 x
"C:\Program Files\Java\jdk-19.0.1\bin\java.exe" ...
>> Reponse de serveur
>> [SERVER RESPONSE] :
    currencyFrom: "USD"
    currencyTo: "MAD"
    amount: 351.0
    result: 3510.0
>> [SERVER RESPONSE] :
    currencyFrom: "USD"
    currencyTo: "MAD"
    amount: 351.0
    result: 3510.0
>> [SERVER RESPONSE] :
    currencyFrom: "USD"
    currencyTo: "MAD"
    amount: 351.0
    result: 3510.0
>> SERVER has end the communication streaming]
```

Figure 11: Résultat d'exécution- pour le cas Serveur Streaming Model

3. Client Streaming Model

3.1. Coté serveur :

```
public StreamObserver<Bank.ConvertCurrencyRequest> convertStreamClient(StreamObserver<Bank.ConvertCurrencyResponse> responseObserver) {  
    return new StreamObserver<Bank.ConvertCurrencyRequest>() {  
        3 usages  
        double total_amount=0;  
        @Override  
        public void onNext(Bank.ConvertCurrencyRequest convertCurrencyRequest) {  
            System.out.println(">> MSG [" +total_amount+"] --> "+convertCurrencyRequest.getAmount()+"$");  
            total_amount+=convertCurrencyRequest.getAmount();  
        }  
  
        @Override  
        public void onError(Throwable throwable) {  
  
        }  
  
        7 usages  
        @Override  
        public void onCompleted() {  
            Bank.ConvertCurrencyResponse response = Bank.ConvertCurrencyResponse.newBuilder()  
                .setAmount(total_amount)  
                .build();  
  
            responseObserver.onNext(response);  
            responseObserver.onCompleted();  
            System.out.println(">> FIN ");  
        }  
    };  
};
```

Figure 12 : Définition d'un service de type Client Streaming Model.

3.2. Coté client :

```
public static void main(String[] args) throws IOException {  
    //Déclarer un canal de communication  
    ManagedChannel managedChannel = ManagedChannelBuilder.forAddress( name: "localhost", port: 9999)  
        .usePlaintext() capture of ?  
        .build();  
  
    BankServiceGrpc.BankServiceStub asyncStub = BankServiceGrpc.newStub(managedChannel); //communic  
    Bank.ConvertCurrencyRequest currencyRequest = Bank.ConvertCurrencyRequest.newBuilder()  
        .setCurrencyTo("MAD")  
        .setCurrencyFrom("USD")  
        .setAmount(351)  
        .build();  
}
```

Figure 13: Définition de la classe Client en mode Client Streaming Model (1)


```

StreamObserver<Bank.ConvertCurrencyRequest> requestObservable =
    asyncStub.convertStreamClient(new StreamObserver<Bank.ConvertCurrencyResponse>() {
        //traitement sur les réponses envoyées par le serveur en mode streaming

        @Override
        public void onNext(Bank.ConvertCurrencyResponse convertCurrencyResponse) {
            System.out.println(">> Server response --> total Amount = " + convertCurrencyResponse);
        }

        @Override
        public void onError(Throwable throwable) {

        }

        7 usages
        @Override
        public void onCompleted() { System.out.println(">> Server has ended the communication"); }
    });

```

Figure 15: Définition de la classe Client en mode Client Streaming Model (2)

```

Timer timer = new Timer();
timer.schedule(new TimerTask() {
    4 usages
    int total_request=0;

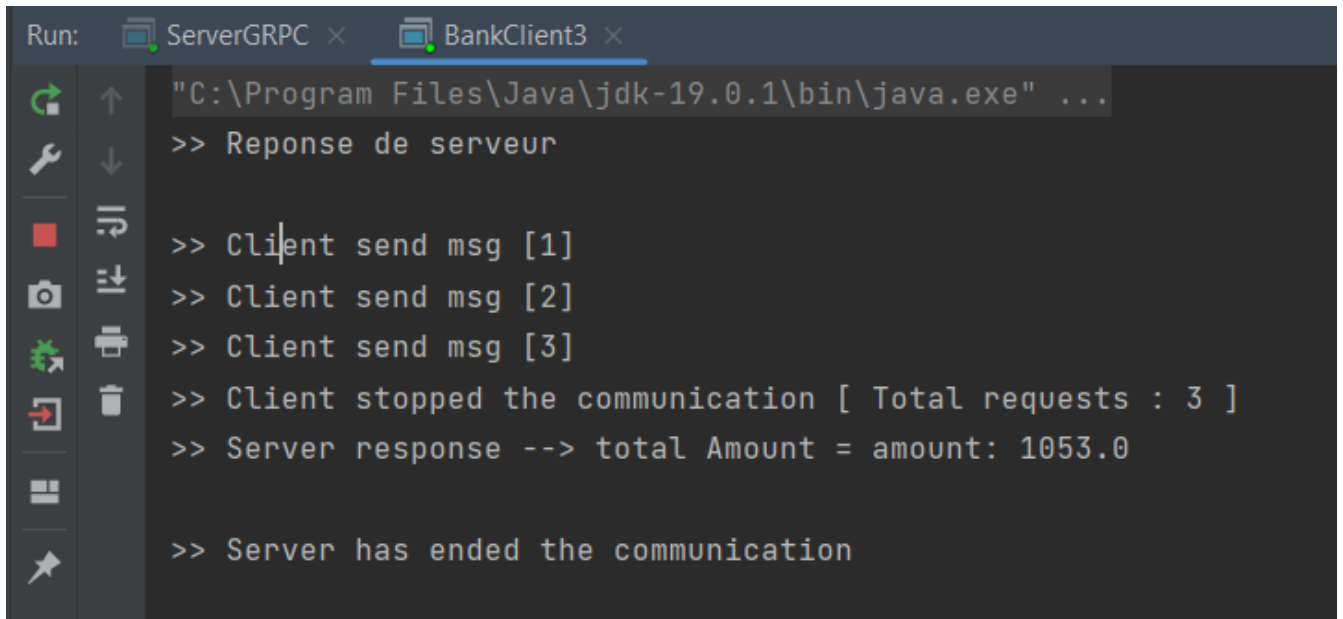
    @Override
    public void run() {
        requestObservable.onNext(currencyRequest); //send the request to the server
        total_request++;
        System.out.println(">> Client send msg [" + total_request + "]);
        if (total_request == 5) {
            requestObservable.onCompleted();
            System.out.println(">> Client stopped the communication [ Total requests : " + total_request-- + " ]");
            timer.cancel();
        }
    }
}, delay: 2000, period: 2000);

System.out.println(">> Reponse de serveur \n");
System.in.read(); //permet de bloquer le thread jusqu'à qu'il lit la méthode Next() ou Complete()

```

Figure 14 : Définition de la classe Client en mode Client Streaming Model (3)

3.3. Exécution :



```
Run: ServerGRPC x BankClient3 x
"C:\Program Files\Java\jdk-19.0.1\bin\java.exe" ...
>> Reponse de serveur
>> Client send msg [1]
>> Client send msg [2]
>> Client send msg [3]
>> Client stopped the communication [ Total requests : 3 ]
>> Server response --> total Amount = amount: 1053.0
>> Server has ended the communication
```

Figure 16 : Résultat d'exécution pour le cas Client Streaming Model

4. Bi-directional Streaming Model :

4.1. Coté serveur :

```
public StreamObserver<Bank.ConvertCurrencyRequest> convertStreamCltSrv(StreamObserver<Bank.ConvertCurrencyResponse> responseObserver) {  
    return new StreamObserver<Bank.ConvertCurrencyRequest>() {  
        2 usages  
        double somme;  
        2 usages  
        int total_request;  
        @Override  
        public void onNext(Bank.ConvertCurrencyRequest convertCurrencyRequest) {  
            total_request++;  
            somme += convertCurrencyRequest.getAmount();  
            Bank.ConvertCurrencyResponse response = Bank.ConvertCurrencyResponse.newBuilder()  
                .setResult(somme)  
                .build();  
  
            System.out.println(">> Client request [" + total_request + "]");  
            responseObserver.onNext(response);  
        }  
  
        @Override  
        public void onError(Throwable throwable) {  
        }  
  
        7 usages  
        @Override  
        public void onCompleted() { responseObserver.onCompleted(); }  
    };  
}
```

Figure 17 : Définition de service en mode Bi-directional Streaming Model

4.2. Coté client :

```
public static void main(String[] args) throws IOException {  
    //Déclarer un canal de communication  
    ManagedChannel managedChannel = ManagedChannelBuilder.forAddress( name: "localhost", port: 9999)  
        .usePlaintext() capture of ?  
        .build();  
  
    BankServiceGrpc.BankServiceStub asyncStub = BankServiceGrpc.newStub(managedChannel); //communic  
    Bank.ConvertCurrencyRequest currencyRequest = Bank.ConvertCurrencyRequest.newBuilder()  
        .setCurrencyTo("MAD")  
        .setCurrencyFrom("USD")  
        .setAmount(351)  
        .build();  
}
```

Figure 18: Définition de la classe Client en mode Bi-directional Streaming Model (1)

```

StreamObserver<Bank.ConvertCurrencyRequest> requestObservable =
    asyncStub.convertStreamClient(new StreamObserver<Bank.ConvertCurrencyResponse>() {
        //traitement sur les réponses envoyées par le serveur en mode streaming

        @Override
        public void onNext(Bank.ConvertCurrencyResponse convertCurrencyResponse) {
            System.out.println(">> Server response --> total Amount = " + convertCurrencyResponse);
        }

        @Override
        public void onError(Throwable throwable) {

        }

        7 usages
        @Override
        public void onCompleted() { System.out.println(">> Server has ended the communication"); }
    });

```

Figure 20: Définition de la classe Client en mode Bi-directional Streaming Model (2)

```

Timer timer = new Timer();
timer.schedule(new TimerTask() {
    4 usages
    int total_request=0;

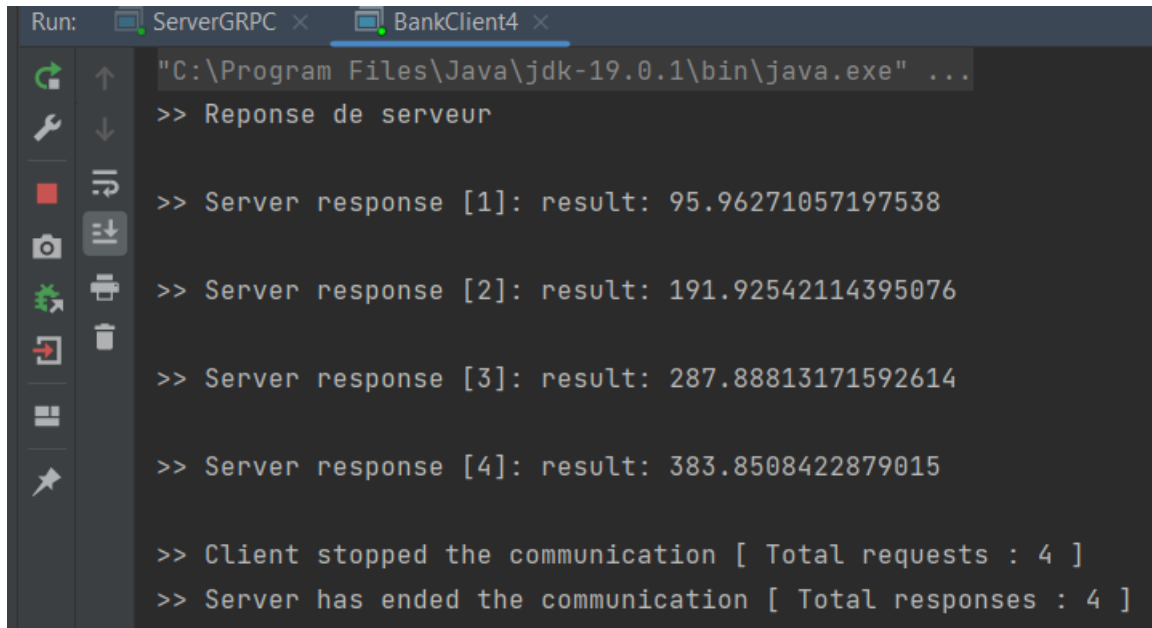
    @Override
    public void run() {
        requestObservable.onNext(currencyRequest); //send the request to the server
        total_request++;
        System.out.println(">> Client send msg [" + total_request + "]);
        if (total_request == 5) {
            requestObservable.onCompleted();
            System.out.println(">> Client stopped the communication [ Total requests : " + total_request-- + " ]");
            timer.cancel();
        }
    }
}, delay: 2000, period: 2000);

System.out.println(">> Reponse de serveur \n");
System.in.read(); //permet de bloquer le thread jusqu'à qu'il lit la méthode Next() ou Complete()

```

Figure 19 : Définition de la classe Client en mode Bi-directional Streaming Model (3)

4.3. Exécution :



```
Run: ServerGRPC x BankClient4 x
"C:\Program Files\Java\jdk-19.0.1\bin\java.exe" ...
>> Reponse de serveur
>> Server response [1]: result: 95.96271057197538
>> Server response [2]: result: 191.92542114395076
>> Server response [3]: result: 287.88813171592614
>> Server response [4]: result: 383.8508422879015
>> Client stopped the communication [ Total requests : 4 ]
>> Server has ended the communication [ Total responses : 4 ]
```

Figure 21: Résultat d'exécution pour le cas Bi-directional Streaming model

FIN DE RAPPORT.