ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Compt Rendu

5ème année
Ingénieur Informatique & Réseaux

TP5

GRPC

--

Réalisé par : *IMANE Chakrellah*
YASSINE Ech-chaoui

Prof : *Pr.Oussama*
Classe: *5IIR-11*

2025-2026

Table des matières

Tout les TPs Réalisés ce trouve dans le lien suivant qui contient le projet + CR :

https://github.com/CHAKRELLAH44/JEE-ARCHITECTURE.git

Objectif du TP

L'objectif de ce TP est de découvrir le fonctionnement de gRPC, une technologie de communication hautement performante basée sur HTTP/2 et Protocol Buffers.
Nous avons appris à :

- définir un contrat d'échange de données via un fichier .proto,
- générer automatiquement les classes Client/Serveur,
- implémenter quatre types de communication : Unary, Server-Streaming, Client-Streaming et Bidirectional-Streaming,
- créer un serveur gRPC et le consommer via un client Java,
- tester les services avec BloomRPC.

Ce TP permet de comprendre comment développer des services distribués performants, plus rapides et plus efficaces que REST.

1. Architecture du projet

1.1 Structure du projet

Le projet est organisé en plusieurs dossiers principaux :

- **src/main/proto/** → contient le fichier calculator.proto (contrat gRPC).
- **src/main/java/service/** → implémentations des services gRPC.
- **src/main/java/server/** → démarrage du serveur gRPC.
- **src/main/java/client/** → code client pour consommer les services.
- **target/generated-sources/protobuf/** → classes Java générées automatiquement (STUBs).

Cette structure sépare clairement :

- le contrat .proto,
- le service serveur,
- le client,
- le code généré automatiquement.

1.2 Description des composants

- **Fichier .proto** : décrit les messages échangés et les méthodes du service.
- **Classes générées** : contiennent les stubs pour le client et le serveur.
- **Service gRPC** : implémente la logique des méthodes (addition, streaming…).
- **Serveur gRPC** : application qui écoute sur un port et expose les services.
- **Client gRPC** : application Java consommant les méthodes du service.
- **BloomRPC** : outil pour tester les appels gRPC.

1.3 Rôle de chaque couche / module

- **.proto** : sert de contrat universel entre tous les clients et serveurs.
- **gRPC Server** : exécute les méthodes définies dans le contrat.
- **gRPC Client** : appelle directement les méthodes du serveur via les stubs.

- **Protocol Buffers** : encode/décode les messages en binaire (très rapide).
- **HTTP/2** : permet streaming, multiplexage et faible latence.

2. Notions importantes

2.1 Concepts théoriques liés au TP

- **gRPC** : Framework RPC moderne basé sur HTTP/2.
- **Protocol Buffers (protobuf)** : format binaire compact et très rapide.
- **Unary RPC** : 1 demande → 1 réponse (équivalent REST).
- **Server Streaming** : 1 demande → plusieurs réponses.
- **Client Streaming** : plusieurs demandes → 1 réponse.
- **Bidirectional Streaming** : les deux côtés échangent plusieurs messages.
- **Stub** : classe générée automatiquement permettant d'appeler un service.

2.2 Technologies et outils utilisés

- **Java 17**
- **gRPC Java**
- **Protocol Buffers compiler**
- **Maven + plugin protobuf**
- **BloomRPC** (équivalent Postman pour gRPC)
- **HTTP/2** pour le transport

2.3 Principes ou notions clés

- Communication basée sur **contrat** (fichier .proto).
- Messages **binaires** → performances élevées.
- Support naturel du **streaming** bidirectionnel.
- Génération automatique du code client/serveur.
- Robustesse et faible consommation réseau.

3. Déroulement du TP

3.1 Description des étapes réalisées

1. Création du fichier .proto contenant la définition du service Calculator.
2. Ajout du plugin Maven pour compiler le fichier .proto en Java.
3. Génération des classes gRPC (stub client + stub serveur).
4. Implémentation du service Unary (addition simple).
5. Implémentation du Server-Streaming (opérations successives).
6. Implémentation du Client-Streaming (somme des valeurs envoyées).
7. Implémentation du Bidirectional Streaming (renvoi du carré de chaque valeur).
8. Démarrage du serveur gRPC sur un port dédié.
9. Test avec BloomRPC en important le fichier .proto.
10. Développement d’un client Java pour consommer tous les modèles de streaming.

Ce déroulement montre l'utilisation complète de gRPC sous ses quatre formes principales.

4. Code source (si applicable)

4.1 Extraits de code essentiels

Exemple d'une méthode Unary :

```
@Override
public void sum(UnaryRequest request, StreamObserver<UnaryResponse> responseObserver) {
    double result = request.getA() + request.getB();
    UnaryResponse response = UnaryResponse.newBuilder()
        .setResult(result)
        .build();
    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

Exemple Server-Streaming :

```
responseObserver.onNext(OperationResponse.newBuilder().setResult(a + b).build());
Thread.sleep(1000);
responseObserver.onNext(OperationResponse.newBuilder().setResult(a - b).build());
...
responseObserver.onCompleted();
```

Exemple Client-Streaming :

```
public StreamObserver<StreamRequest> sumStream(StreamObserver<StreamResponse> responseObserver) {
    return new StreamObserver<>() {
        double total = 0;

        @Override
        public void onNext(StreamRequest request) {
            total += request.getValue();
        }

        @Override
        public void onCompleted() {
            responseObserver.onNext(StreamResponse.newBuilder().setTotal(total).build());
            responseObserver.onCompleted();
        }
    };
}
```

4.2 Explication du workflow

Workflow Unary :

Client → envoie message → serveur calcule → renvoie réponse → fin.

Workflow Server-Streaming :

Client → envoie 1 message → serveur renvoie plusieurs réponses → fin.

Workflow Client-Streaming :

Client → envoie plusieurs messages → serveur calcule → renvoie réponse unique.

Workflow Bidirectional Streaming :

Client ↔ Serveur → échanges multiples simultanés (temps réel).

4.3 Rôle des principales classes/fonctions

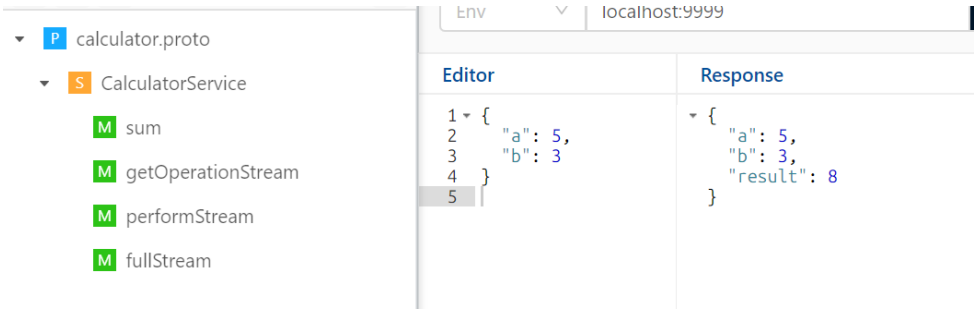
- **CalculatorServiceGrpc.CalculatorServiceImplBase** : classe générée à étendre pour implémenter le service.
- **UnaryRequest / UnaryResponse** : messages protobuf.
- **StreamObserver** : interface permettant d'envoyer et recevoir des messages.
- **GrpcServer** : démarre le serveur sur un port donné.
- **GrpcClient** : consomme les méthodes du service.

5. Tests (JUnit, BloomRPC, Console)

5.1 Tests des appels gRPC (avec explication)

Test 1 — Unary RPC

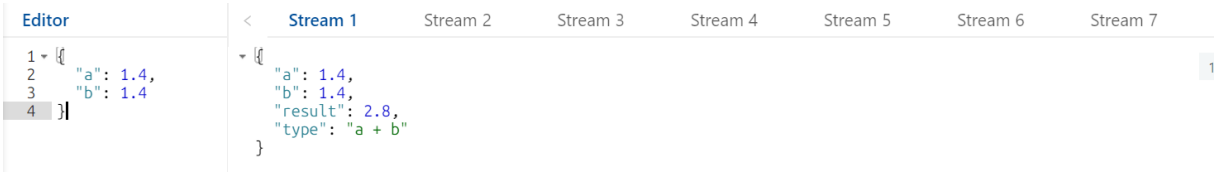
Méthode : sum()
Objectif : Un seul message envoyé → une seule réponse.
Requête (BloomRPC) :

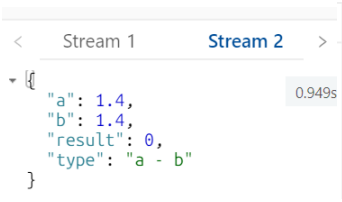


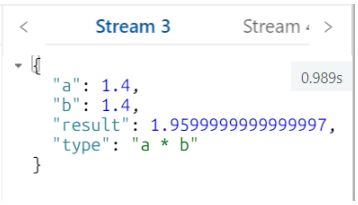
Résultat attendu : { result: 8 }

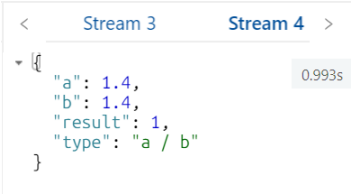
Test 2 — Server Streaming

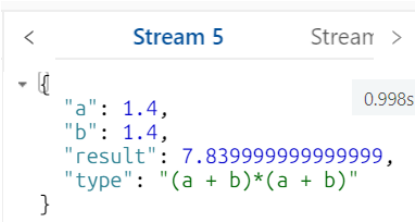
Méthode : getOperationStream()
Objectif : Une seule requête → plusieurs réponses envoyées par le serveur pendant 10 secondes.
Résultat attendu : Le serveur t'envoie une série de calculs chaque seconde : addition, soustraction, multiplication, division, etc

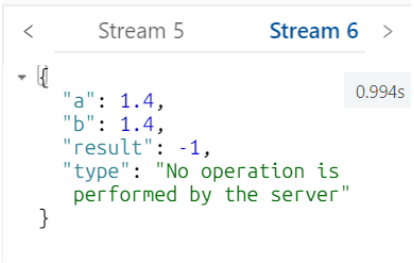






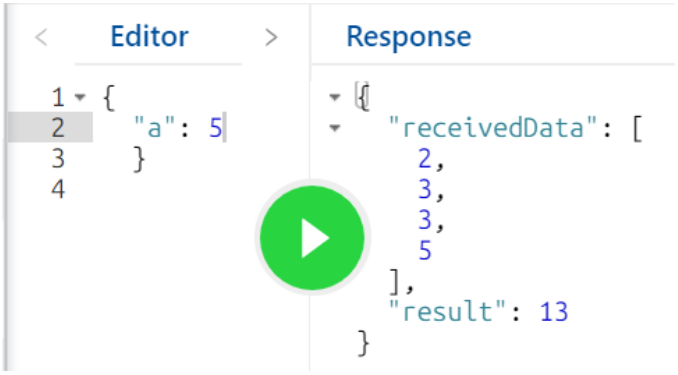






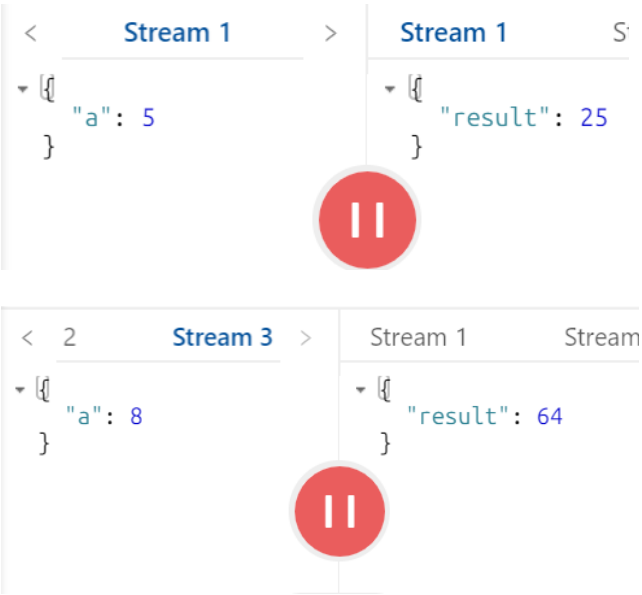
Test 3 — Client Streaming

Méthode : performStream()
Objectif : Plusieurs requêtes envoyées par le client → une seule réponse finale.
Résultat attendu : Tu envoies plusieurs valeurs **une par une**.
Le serveur les additionne toutes et renvoie la somme + la liste des valeurs reçues.



Test 4 — Bidirectional Streaming

Méthode : fullStream()
Objectif : *Le client envoie plusieurs valeurs → le serveur répond à chaque message.*
Résultat attendu : Pour chaque message envoyé, le serveur renvoie **le carré du nombre**.



7. Outils de visualisation

- **BloomRPC** : permet de charger le fichier .proto, d'exécuter toutes les méthodes, d'observer les flux streaming.
- **Console Java** : observe l'ordre d'arrivée des messages et leur traitement.
- **Logs gRPC** : permettent de vérifier que les streams se ferment correctement.

8. Conclusion

Ce TP nous a permis d'explorer en profondeur la technologie gRPC : définition du contrat dans le fichier .proto, génération automatique du code client/serveur, utilisation des quatre modes de communication, tests via BloomRPC, et implémentation d'un client Java.
Nous avons compris en quoi gRPC est plus rapide, plus compact et mieux adapté au streaming que REST ou GraphQL.

Limites du TP

- Pas d'authentification ni de sécurisation TLS.
- Pas de gestion d'erreurs avancée côté client.
- Tests limités à un seul type de service (Calculator).
- Absence d'interopérabilité avec d'autres langages (Python, Node, etc.) dans le TP.